

Foundations of Python Network Programming

*The comprehensive guide to building network
applications with Python*

John Goerzen

Apress®



Author of
Linux Programming Beginner's Guide
Debian GNU/Linux: Guide to Installation and Usage
Debian GNU/Linux 2.1 Unleashed
Linux Unleashed

Foundations of Python Network Programming

Dear Reader,

One reason for the Python programming language's popularity is its vast assortment of convenient and flexible features. Python's networking capabilities are no different, offering unfettered access to your operating system's networking library, not to mention many different modules that provide complete protocol implementations such as HTTP, FTP, SMTP, IMAP, DNS, and others. Whether you want to write network-enabled applications that communicate with the computer down the hall or the server across the ocean, Python is the language for you. And I want to help you get going fast, so I've included over 175 example programs.

The first part of this book discusses client and server design, which are the building blocks of every network-oriented application. You'll learn how to write applications that take advantage of IPv6, how to communicate with domain name servers, how to fetch web pages, and more.

In the second part of the book, I'll show you how to write applications that are capable of sending and receiving e-mail, serving static and dynamic web pages, transferring files, and executing other commonplace network-oriented tasks. You'll also learn how to build web-based Python applications using the mod_python Apache module.

In the final part of the book, I focus on more advanced topics, teaching you how to use multitasking and non-blocking I/O to create servers that are capable of handling thousands of clients. Along the way, you'll learn many secrets to successful network programming. And most important, with over 6,600 lines of self-contained, ready-to-run code, you'll be able to immediately put the concepts you learn into practice.

John Goerzen

THE APRESS PYTHON ROADMAP

Practical Python

The Definitive Guide
to Python

Dive Into Python

Foundations of Python
Network Programming

Join online discussions at
forums.apress.com

APRESS™
SOURCE CODE ONLINE
www.apress.com

US \$44.99

Shelves in Programming
Languages/Python

User Level:
Intermediate-Advanced

ISBN 1-59059-371-5

54499



6 89253 15715 2 9 781590 593716

Foundations of Python Network Programming

JOHN GOERZEN

Apress®

Foundations of Python Network Programming
Copyright © 2004 by John Goerzen

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN (pbk): 1-59059-371-5

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Lead Editor: Jason Gilmore

Technical Reviewer: Magnus Lie Hetland

Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, John Franklin, Jason Gilmore, Chris Mills, Steve Rycroft, Dominic Shakeshaft, Jim Sumser, Karen Watterson, Gavin Wray, John Zukowski

Project Manager: Beth Christmas

Copy Edit Manager: Nicole LeClerc

Copy Editor: Mark Nigara

Production Manager: Kari Brooks

Production Editor: Ellie Fountain

Composer: Susan Glinert Stevens

Proofreader: Elizabeth Berry

Indexer: Kevin Broccoli

Cover Designer: Kurt Krames

Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, e-mail orders@springer-ny.com, or visit <http://www.springer-ny.com>. Outside the United States: fax +49 6221 345229, e-mail orders@springer.de, or visit <http://www.springer.de>.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit <http://www.apress.com>.

The information in this book is distributed on an "as is" basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at <http://www.apress.com> in the Downloads section.

*To my wife Terah: Thank you for tolerating the tall grass,
late evenings, and short weekends while I worked on this book.
Thank you for supporting me every step of the way with
encouragement, love, and chocolate cake.*

Contents at a Glance

<i>About the Author</i>	<i>xv</i>
<i>About the Technical Reviewer</i>	<i>xvii</i>
<i>Acknowledgments</i>	<i>xix</i>
<i>Introduction</i>	<i>xxi</i>
Part One Low-Level Networking	1
<i>Chapter 1 Introduction to Client/Server Networking</i>	3
<i>Chapter 2 Network Clients</i>	19
<i>Chapter 3 Network Servers</i>	35
<i>Chapter 4 Domain Name System</i>	65
<i>Chapter 5 Advanced Network Operations</i>	87
Part Two Web Services	111
<i>Chapter 6 Web Client Access</i>	113
<i>Chapter 7 Parsing HTML and XHTML</i>	127
<i>Chapter 8 XML and XML-RPC</i>	145
Part Three E-mail Services	167
<i>Chapter 9 E-Mail Composition and Decoding</i>	169
<i>Chapter 10 Simple Message Transport Protocol</i>	197
<i>Chapter 11 POP</i>	211
<i>Chapter 12 IMAP</i>	223
Part Four General-Purpose Client Protocols	273
<i>Chapter 13 FTP</i>	275
<i>Chapter 14 Database Clients</i>	295
<i>Chapter 15 SSL</i>	321

Part Five	Server-Side Frameworks	339
<i>Chapter 16</i>	<i>SocketServer</i>	341
<i>Chapter 17</i>	<i>SimpleXMLRPCServer</i>	355
<i>Chapter 18</i>	<i>CGI</i>	369
<i>Chapter 19</i>	<i>mod_python</i>	393
Part Six	Multitasking.....	417
<i>Chapter 20</i>	<i>Forking</i>	419
<i>Chapter 21</i>	<i>Threading</i>	443
<i>Chapter 22</i>	<i>Asynchronous Communication</i>	469
<i>Index</i>		491

Contents

<i>About the Author</i>	<i>xv</i>
<i>About the Technical Reviewer</i>	<i>xvii</i>
<i>Acknowledgments</i>	<i>xi</i>
<i>Introduction</i>	<i>xxi</i>
Part One Low-Level Networking	1
Chapter 1 <i>Introduction to Client/Server Networking</i>	3
<i>Understanding TCP Basics</i>	3
<i>Using the Client/Server Model</i>	6
<i>Understanding User Datagram Protocol</i>	7
<i>Understanding Physical Transports and Ethernet</i>	9
<i>Networking in Python</i>	9
<i>Summary</i>	17
Chapter 2 <i>Network Clients</i>	19
<i>Understanding Sockets</i>	19
<i>Creating Sockets</i>	20
<i>Communicating with Sockets</i>	23
<i>Handling Errors</i>	23
<i>Using User Datagram Protocol</i>	31
<i>Summary</i>	34

Chapter 3 Network Servers	35
<i>Preparing for Connections</i>	35
<i>Accepting Connections</i>	40
<i>Handling Errors</i>	41
<i>Using User Datagram Protocol</i>	43
<i>Using inetd or xinetd</i>	45
<i>Logging with syslog</i>	55
<i>Avoiding Deadlock</i>	60
<i>Summary</i>	63
Chapter 4 Domain Name System	65
<i>Making DNS Queries</i>	65
<i>Using Operating System Lookup Services</i>	66
<i>Using PyDNS for Advanced Lookups</i>	76
<i>Summary</i>	85
Chapter 5 Advanced Network Operations	87
<i>Half-Open Sockets</i>	87
<i>Timeouts</i>	89
<i>Transmitting Strings</i>	90
<i>Understanding Network Byte Order</i>	93
<i>Using Broadcast Data</i>	95
<i>Working with IPv6</i>	97
<i>Binding to Specific Addresses</i>	102
<i>Using Event Notification with poll() or select()</i>	104
<i>Summary</i>	109
Part Two Web Services	111
Chapter 6 Web Client Access	113
<i>Fetching Web Pages</i>	114
<i>Authenticating</i>	115
<i>Submitting Form Data</i>	118
<i>Handling Errors</i>	121
<i>Using Non-HTTP Protocols</i>	125
<i>Summary</i>	125

<i>Chapter 7 Parsing HTML and XHTML</i>	127
<i>Understanding Basic HTML Parsing</i>	128
<i>Handling Real-World HTML</i>	130
<i>A Working Example</i>	137
<i>Summary</i>	143
<i>Chapter 8 XML and XML-RPC</i>	145
<i>Understanding XML Documents</i>	147
<i>Using DOM</i>	148
<i>Using XML-RPC</i>	159
<i>Summary</i>	166
<i>Part Three E-mail Services.....</i>	167
<i>Chapter 9 E-Mail Composition and Decoding</i>	169
<i>Understanding Traditional Messages</i>	169
<i>Composing Traditional Messages</i>	173
<i>Parsing Traditional Messages</i>	176
<i>Understanding MIME</i>	180
<i>Composing MIME Attachments</i>	182
<i>Composing MIME Alternatives</i>	185
<i>Composing Non-English Headers</i>	187
<i>Composing Nested Multiparts</i>	188
<i>Parsing MIME Messages</i>	190
<i>Summary</i>	195
<i>Chapter 10 Simple Message Transport Protocol</i>	197
<i>Introducing the SMTP Library</i>	197
<i>Error Handling and Conversation Debugging</i>	199
<i>Getting Information from EHLO</i>	202
<i>Using Secure Sockets Layer and Transport Layer Security</i>	205
<i>Authenticating</i>	208
<i>SMTP Tips</i>	209
<i>Summary</i>	210

Chapter 11 POP	211
<i>Connecting and Authenticating</i>	212
<i>Obtaining Mailbox Information</i>	215
<i>Downloading Messages</i>	216
<i>Deleting Messages</i>	218
<i>Summary</i>	221
Chapter 12 IMAP	223
<i>Understanding IMAP in Python</i>	224
<i>Introducing IMAP in Twisted</i>	225
<i>Understanding Twisted Basics</i>	226
<i>Scanning the Folder List</i>	236
<i>Examining Folders</i>	239
<i>Basic Downloading</i>	243
<i>Flagging and Deleting Messages</i>	249
<i>Retrieving Message Parts</i>	255
<i>Finding Messages</i>	262
<i>Adding Messages</i>	268
<i>Creating and Deleting Folders</i>	270
<i>Moving Messages Between Folders</i>	270
<i>Summary</i>	271
Part Four General-Purpose Client Protocols	273
Chapter 13 FTP	275
<i>Understanding FTP</i>	275
<i>Using FTP in Python</i>	277
<i>Downloading ASCII Files</i>	278
<i>Downloading Binary Files</i>	279
<i>Uploading Data</i>	281
<i>Handling Errors</i>	283
<i>Scanning Directories</i>	284
<i>Downloading Recursively</i>	290
<i>Manipulating Server Files and Directories</i>	293
<i>Summary</i>	294

Chapter 14 Database Clients	295
<i>SQL and Networking</i>	295
<i>SQL in Python</i>	296
<i>Connecting</i>	297
<i>Executing Commands</i>	301
<i>Transactions</i>	302
<i>Repeating Commands</i>	305
<i>Retrieving Data</i>	310
<i>Reading Metadata</i>	313
<i>Using Data Types</i>	317
<i>Summary</i>	319
Chapter 15 SSL	321
<i>Understanding Network Vulnerabilities</i>	322
<i>Reducing Vulnerabilities with SSL</i>	324
<i>Understanding SSL in Python</i>	326
<i>Using Built-in SSL</i>	326
<i>Using OpenSSL</i>	330
<i>Verifying Server Certificates with OpenSSL</i>	331
<i>Summary</i>	338
Part Five Server-Side Frameworks.....	339
Chapter 16 SocketServer	341
<i>Using BaseHTTPServer</i>	341
<i>SimpleHTTPServer</i>	348
<i>CGIHTTPServer</i>	349
<i>Implementing New Protocols</i>	350
<i>IPv6</i>	352
<i>Summary</i>	353
Chapter 17 SimpleXMLRPCServer	355
<i>SimpleXMLRPCServer Basics</i>	356
<i>Serving Functions</i>	359
<i>Exploiting Class Features</i>	361
<i>Using DocXMLRPCServer</i>	364
<i>Using CGIXMLRPCRequestHandler</i>	365
<i>Supporting Multicall Functions</i>	367
<i>Summary</i>	367

Chapter 18 CGI	369
<i>Setting Up CGI</i>	370
<i>Understanding CGI</i>	370
<i>Understanding CGI in Python</i>	371
<i>Retrieving Environment Information</i>	373
<i>Getting Input</i>	375
<i>Escaping Special Characters</i>	383
<i>Handling Multiple Inputs per Field</i>	385
<i>Uploading Files</i>	386
<i>Using Cookies</i>	388
<i>Summary</i>	392
Chapter 19 mod_python	393
<i>Understanding the Need for mod_python</i>	393
<i>Installing and Configuring mod_python</i>	394
<i>Understanding mod_python Basics</i>	399
<i>Dispatching Requests</i>	402
<i>Handling Input</i>	405
<i>Escaping</i>	412
<i>Understanding Interpreter Instances</i>	413
<i>Prebuilt Handlers in mod_python</i>	415
<i>Summary</i>	415
Part Six Multitasking	417
Chapter 20 Forking	419
<i>Understanding Processes</i>	419
<i>Understanding fork()</i>	421
<i>Forking First Steps</i>	424
<i>Forking Servers</i>	430
<i>Locking</i>	433
<i>Error Handling</i>	438
<i>Summary</i>	441
Chapter 21 Threading	443
<i>Threading in Python</i>	444
<i>Writing Threaded Servers</i>	455
<i>Writing Threaded Clients</i>	463
<i>Summary</i>	467

Chapter 22 Asynchronous Communication	469
<i>Deciding Whether or Not to Use</i>	
<i>Asynchronous Communication</i>	470
<i>Using Asynchronous Communication</i>	471
<i>Advanced Server-Side Use</i>	476
<i>Monitoring Multiple Master Sockets</i>	480
<i>Using Twisted for Servers</i>	485
<i>Summary</i>	489
Index	491

About the Author

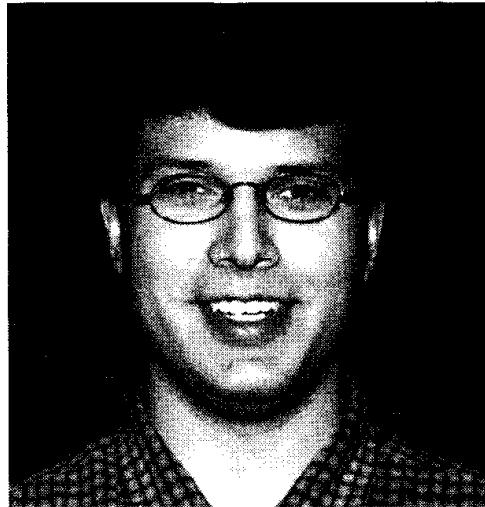
John Goerzen has been a member of the Debian GNU/Linux operating system development team since 1996 and has been writing software for the last 15 years. He is interested in operating systems, programming languages, and networking, and has worked with diverse sets of each. He works as a programmer and UNIX administrator for a mid-sized manufacturing firm, and has developed Python interfaces to many of the company's data systems.

As part of his involvement with Debian, John has maintained several dozen different programs for the system. Since 1998, he has been involved with starting or enhancing ports of Debian to new architectures, and has worked with the Alpha, PowerPC, AMD64, and NetBSD i386 porting efforts.

In 2003, John was elected to the Board of Directors for Software in the Public Interest, Inc., the organization that manages Debian's legal and financial affairs. Later that year, he was named vice president of SPI.

Aside from Debian, John spends a lot of time writing software. Two of his most well-known works are network-enabled programs written in Python. OfflineIMAP is a bidirectional mail synchronization program written with laptop users in mind, and PyGopherd is a multiprotocol web and Gopher server. Both use Python's unique features to support a modular architecture, and both offer extreme flexibility.

John has written many books, including the 800-page *Linux Programming Bible*. His articles have appeared in various magazines, and he has served as technical editor for several books. He also founded the Air Capital Linux Users group and was a frequent speaker there.



About the Technical Reviewer

Magnus Lie Hetland is an associate professor of algorithms at the Norwegian University of Science and Technology (NTNU). He has been using Python since 1997 and is the author of the popular online tutorials “Instant Hacking” and “Instant Python.” His publications include *Practical Python* (Apress, 2004) as well as several scientific papers.



Acknowledgments

THIS BOOK HAS BENEFITED tremendously from the insight, experience, dedication, and encouragement of many people. Without them, this project would not have happened. I'd like to especially thank the following people:

- Magnus Lie Hetland, technical reviewer. Magnus has a level of Python knowledge that I hope to acquire someday. He has a keen eye for detail, and a great knack for explaining things in a clear way. He found more bugs than I ever thought possible, and I thank him for it.
- Ellie Fountain, production editor. Ellie was extremely patient with my struggles to find a machine capable of running the software I needed to review your files.
- Beth Christmas, project manager. Just when I was losing track of which project to work on next, an e-mail from Beth would pop up, conveniently summarizing exactly what the next steps should be. She did a phenomenal job managing the schedule for this project. I have no idea how she made things always work out, even when I was slower than expected.
- Jason Gilmore, lead editor. Jason's comments on each chapter made the text far stronger than it was originally. By the time the book was half done, I'd read some text and think, "Jason would tell me to do that differently." His words of encouragement helped get me through the frantic days when things were happening at the last minute.
- Mark Nigara, copy editor. Thanks to Mark's reviews, the spelling is correct, the text flows well, and the book is easy to read. Reading over Mark's edits, I was often amazed at how much improvement he could bring to the text.

Many people contributed to the technologies used in this book. I'd like to highlight a few of them:

- To Guido van Rossum: Thank you for providing us with a language so versatile, yet so readable.
- To Richard M. Stallman: Thank you for showing us all the value of collaboration and sharing of technology.

Acknowledgments

And finally, to Vint Cerf, Robert Kahn, the late Jonathan Postel, and the many other Internet pioneers: Thank you for letting us all send a few packets on your network.

Introduction

TWENTY-FIVE YEARS AGO, the world was a different place. Talking to someone across the ocean was, for most people, an exceptionally rare experience. Sending a letter could take weeks. Listening to a shortwave newscast from a foreign land required technical skill, patience, and precise atmospheric conditions.

Today, we think nothing of receiving an e-mail from South Korea, checking the weather in California, and reading the day's headlines in Germany—all in less than five minutes. Files zip across the global electronic network, allowing us to do everything from managing investment accounts to seeing pictures of distant relatives.

And yet, despite all that has happened in the last 25 years, the Internet is still in its infancy. It's a new technology, still growing.

I wrote this book because the Internet is *exciting*. In the past few years, we've seen the rise of an entire industry that did not exist before. It's a place where inventors thrive.

And that is what I hope you get from this book. I want this to be your lab manual—your guide for inventing things that make the Internet better.

Organization

This book is divided into five parts. The first part explains how the Internet works. You'll learn about the fundamentals of Internet communication, and the examples provide you with the basic tools that you'll use to assemble your programs.

Part Two covers web-based services and Part Three covers e-mail services. Many new technologies are a form of web or e-mail communication, and these parts explain how to write programs that take advantage of them.

Part Four covers other technologies, such as databases and file transfer, that are often used behind the scenes of network programs. Your users may never know you use them, but they're still key components of your toolbox.

Part Five shows you how to write servers, the applications that answer requests and give out information. You may never need to write a server, but if you're designing a new protocol from scratch, you'll need the techniques here.

Finally, Part Six shows you how to do more than one thing at a time. Some network programmers never need to use these techniques. Others, such as server designers, couldn't survive without them.

Assumptions

If you're reading this book, I assume that you already know Python. If not, I recommend the Apress book *Practical Python* by Magnus Lie Hetland.

I also assume that you have Python 2.3 or above installed on your system with networking support compiled in. If you don't already have Python installed, check first with your operating system's packaging or installation system; a package is available for many systems. Otherwise, source code and installers can be downloaded directly from www.python.org.

Finally, I assume that you have a network connection configured on your system (even if you didn't personally configure it). You don't need to have any knowledge of Internet protocols to read this book.

Examples

There are many examples in these pages. You can download all of them from the Apress website at www.apress.com. I encourage you to try the examples included here, and to experiment with them. Many are suitable as a starting point for your own programs, and almost every one is a fully contained, ready-to-run program. You can see for yourself how things work.

I'll often show you an example of an interaction with a program. When you see such an example, text in bold represents keyboard input. When you see commands, the \$ character represents your operating system's command prompt.

Networking relies heavily on the network support in your operating system. The operating system provides all of the basic infrastructure necessary for communicating with other computers. Here are some notes related to specific operating systems.

TIP If you have a problem with any example in this book, check these notes or the text in the chapter for an explanation. Some examples cannot run on all operating systems.

Linux, FreeBSD, Mac OS X, Solaris, and Other UNIX Platforms

Each example program assumes that you have the Python interpreter, named python, installed on your path. They also assume you have the /usr/bin/env program available. The first line of each executable Python script begins with #!,

which tells the operating system where to find the interpreter to run your programs. If none of these examples work, you either don't have Python installed or your operating system can't find it. Try replacing the text after the `#!` with the full path to your interpreter.

Also, scripts must be marked executable. If your operating system complains of permissions problems when you attempt to run a script, you can run a command such as `chmod 0755 scriptname.py` to mark it executable.

If all else fails, you can start the Python interpreter manually. You can run `python scriptname.py` instead of `./scriptname.py` to invoke a particular script.

In this book, when I refer to Linux/UNIX platforms, I loosely mean any UNIX-like operating system. Mac OS X is included in this description. Versions of Mac OS prior to OS X are not considered in this book.

Windows

Windows doesn't support the special `#!` notion that Linux/UNIX platforms use to start scripts. Therefore, you cannot use `./scriptname.py` to start your scripts in Windows. Whenever you see a command of the form `./scriptname.py` in this book, replace it with a command such as `python scriptname.py`.

Windows also does not support certain features that Linux/UNIX platforms typically do (or, in some cases, Python cannot present the same interface to them). The most prominent examples of this are the `inetd`-style server in Chapter 3, forking in Chapter 20, and certain parts of asynchronous communication in Chapter 22. However, in each of these cases, the technologies being described are one choice among several that you can use to solve a problem. You'll still be able to select a different method to achieve the same end result. Comments in the chapters themselves will direct you to these alternatives.

Several different versions of Python are available from Windows. The standard version is available from www.python.org. However, it doesn't support Windows methods of accomplishing certain tasks and certain versions are known to have some bugs related to network programming. An alternative is ActivePython, which is available from www.activestate.com. Whichever version you're using, if you encounter unexpected trouble, you may wish to try the other version of Python.

Those with a Linux/UNIX background may wish to try Cygwin, which is available from www.cygwin.com. It provides a Linux-like environment for Windows, and Python can be compiled from sources on www.python.org in this environment.

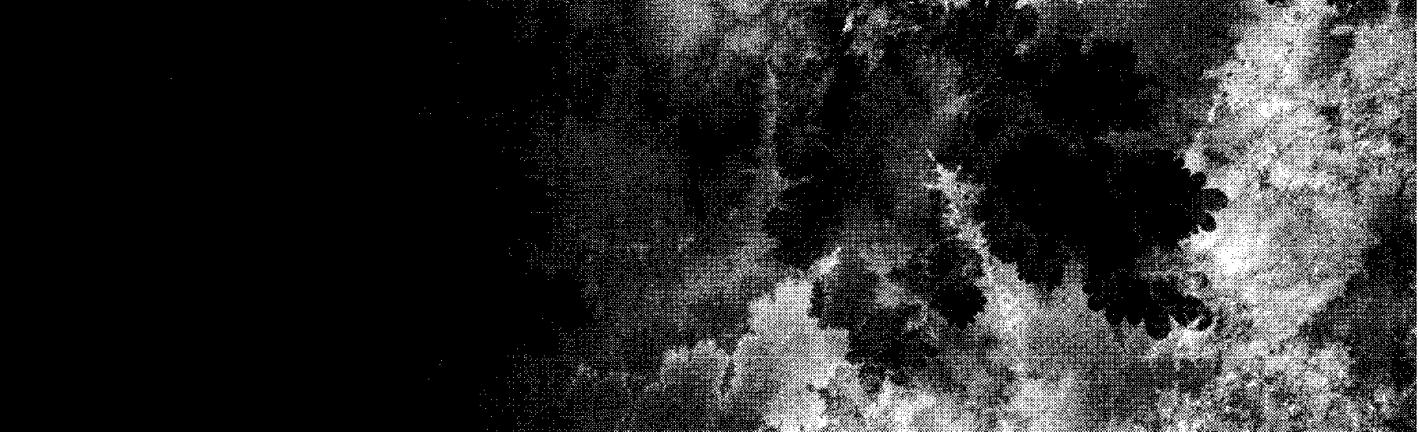
Other Helpful Resources

Should you run into trouble about a particular topic, here's a list of online resources that can help you out:

- Formal standards for Internet protocols are known as Request for Comments (RFC) documents. You can find RFCs at www.rfc-editor.org and www.faqs.org.
- The Python module reference at www.python.org/doc is a good tool for Python-specific information.
- The `comp.lang.python` newsgroup is a good place to search for answers or post questions if you get stuck with a particular problem. You can find details on this group at www.faqs.org/faqs/python-faq/python-newsgroup-faq/. That page also has information on how to participate via e-mail if you don't have Usenet news access.
- Your operating system's developer documentation can provide information on low-level network operations and basic network configuration.

Your Comments

I want to hear from you. Feel free to send me your comments about this book. You may e-mail me at jgoerzen+pynet@complete.org. Although I do read all mail sent to me, please understand that I don't have the time to reply to all of it, so please don't be offended if you don't receive a reply.



Part One

Low-Level Networking

CHAPTER 1

Introduction to Client/Server Networking

FOR MANY YEARS, people have been interested in making computers communicate with each other. Many communication methods have come and gone, and many still remain active today. However, the most popular by far is known as Transmission Control Protocol/Internet Protocol (TCP/IP). TCP/IP is the standard protocol that allows computers worldwide to communicate over the Internet or to someone three feet away.

Everything in the book you hold in your hands is based upon TCP/IP. Although an understanding of TCP/IP isn't strictly necessary to be able to write basic network services, it's essential if you expect to write more advanced services, or if you want to make your programs more secure and robust.

This chapter will give you a high-level overview of how TCP/IP works, starting with the basic principles of TCP/IP and how it travels across networks, then covering the differences between two Internet protocols: TCP and UDP. Finally, you'll see some simple Python code.

More details on TCP/IP are presented in Chapter 2 (from a client perspective) and Chapter 3 (from a server perspective). Chapter 4 provides greater detail on the operation of the Domain Name System (DNS), which translates textual computer names into numeric addresses. Finally, Chapter 5 discusses some more advanced low-level operations that can apply to both clients and servers.

Understanding TCP Basics

TCP/IP is actually a collection of protocols. The majority of communications in use today use the TCP protocol.

The Internet works by sending lots of traffic over shared lines. For instance, you might have several applications open on your PC such as a web browser, instant messenger, and e-mail program. Yet you use a single modem or DSL line to communicate. All those applications share that connection, to the point where it's often not even noticeable that this sharing is happening.

If you think about chatting with four or five people plus sending a fax using a telephone, you realize that you would need to have one line for your fax machine, plus several more lines for your telephone conversations.

TCP permits sharing by breaking down your data stream into small *packets*, which are sent out across the Internet (possibly interspersed with packets from other applications), and then reassembling them at the other end. By making the packets small, the Internet connection is only tied up with sending each bit of data for a small amount of time, and other applications' packets can get sent, too.

Addressing

In order to make the packet scheme work, there are several details that TCP must take care of. First, it needs a way to identify the remote machine. With TCP/IP networking, each machine has a unique *IP address* that looks something like 192.168.1.1. Once you know the IP address of the recipient's machine, you can send information to it.

The second thing TCP needs to know is which application on the remote machine to communicate with. For example, if you're trying to send information to Jane's computer in Houston, and she is running two chat clients and a web browser, her computer will need some way to know which application should receive the incoming data. For this purpose, TCP uses a *port number*. Each application will have a unique port number. As you'll see later, those numbers are sometimes known in advance and sometimes randomly assigned. Therefore, each endpoint of a TCP connection is uniquely identified by an IP address and a port number.

Although TCP is quite capable of working on this level, Internet architects realized early on that it's difficult for people to remember strings of numbers like 65.215.221.149. For that reason, there's the DNS today. When you want to establish a connection to a remote machine, you can ask DNS for the IP address corresponding to something like www.apress.com. The DNS will give you an IP address, and with that you can proceed to establish your connection. Python often hides the DNS layer from applications, so in some cases you don't even need to be aware of its existence. However, this isn't always sufficient, so the DNS system will be described in detail in Chapter 4.

Reliability

Just as you sometimes get static on the line during a telephone conversation, you can also have your data corrupted as it travels across the Internet. There are many different things that could happen: a modem could change a few bytes of data,

a router might drop a packet or two, the system may receive packets out of order, a packet may be duplicated, or a major network line might be cut by a backhoe.

TCP is a *reliable* protocol, which means that, except in the case of a total network outage, your data should get through to the other end intact, complete, unmodified, and in the correct order. This is accomplished with the use of several different algorithms.

To handle situations in which data is corrupted during transmission, each packet includes a *checksum*. This checksum is a code used to ensure that the data in the packet has not been modified during transmission. When the packet reaches its destination, the receiver compares the checksum with the data received in the packet. If the checksum doesn't match, the packet is ignored.

To handle situations in which packets get dropped, TCP expects the remote to acknowledge the receipt of each packet. If the recipient doesn't acknowledge a given packet, the sender automatically resends it. You, the application programmer, aren't even aware that a problem developed because the system handles this automatically. TCP will continue its attempts to resend the packet until the recipient finally receives it, or it decides that the network connection is down and it returns an error to your program.

To handle situations in which packets get duplicated or handled out of order, TCP transmits a sequence number with each packet. The recipient will look at this number to make sure it receives and reassembles packets in order. Additionally, if it sees a sequence number corresponding to a packet it's already seen, it will discard the duplicate packet.

Routing

In order for your packets to get from your machine to the remote server, they will typically pass over a dozen or more different lines. They might go across your DSL to the phone company, then to an Internet provider in your city, then through Dallas, Chicago, New York, and London on their way to the final destination. At every stop, packets from thousands of other computers are also traveling across the link.

Devices on the Internet called routers receive packets and decide how to get them to their intended destination. You can use programs such as traceroute and mtr to show you exactly whose routers your packets cross as they travel over the Internet.

When routers fail, your applications will notice. Links might get congested, causing packets to be dropped and performance to slow to a crawl. Or your connection might get severed entirely.

Security

One of the important functions of routers and local networks is security. As your packets travel around the Internet, anyone with access to network hardware (and, sometimes, anyone with a laptop and an Ethernet card) can watch your packets as they travel across the lines—and even insert or modify data as it goes by. In some cases, this might not be a big problem. If you’re one of two thousand people reading the homepage at www.CNN.com, a random attacker probably isn’t going to go to the work to intercept your packets just to see a website that anyone could access already.

But sometimes there’s more sensitive data traveling across the Internet. If you go to an online shopping site, you wouldn’t want random strangers to be able to find your credit card number and address. If you’re logged in to a remote computer, you wouldn’t want someone to be able to get your password or insert `rm -rf *` or `del *.*` in the middle of commands you’re typing.

Another potential security risk is that of an interceptor that diverts your connections to another machine. Thus, when you think you’re connected to www.Borders.com and ordering a great book, you might really be talking to a computer in Russia that’s carefully noting your credit card number.

Programmers have devised several approaches to this problem. The most popular in use today are known as Secure Sockets Layer (SSL) and Transport Layer Security (TLS). SSL is a layer that’s normally incorporated in application code above a TCP connection. It provides authentication of the server (so you know you’re talking to the machine you think you should be), encryption (so nobody else can read your communications), and data integrity (so no packets can be modified en route without being detected). TLS is very similar to SSL, but is included as a part of the protocol stack.

Security is becoming more and more important, and it’s vital to realize the weaknesses of traditional unencrypted connections—and to know how to effectively deal with them.

Using the Client/Server Model

TCP/IP lends itself to the *client/server* way of communicating. With this architecture, a *server* continually listens for requests from *clients*, and establishes connections to handle each one.

For example, if you open a web browser to view www.google.com, your web browser connects to the server on www.google.com and requests the / page; a single slash identifies the home page for the site. The server, in turn, retrieves that page and transmits it back to your client, which then formats it for display onscreen. A key here is that the client is always the side that initiates the connection; the server just sits and waits until clients connect.

Server-Side Port Numbers

You'll recall from the earlier discussion that, in order to communicate with a remote application, you must know its IP address and port number. Finding the IP address in this case is straightforward (just ask DNS for the IP address of www.google.com), but you might be wondering how to find out the port number of the web server.

With the client/server model, the server typically listens on a well-known port. In this case, web servers listen on port 80. So, a web browser knows to connect to port 80 on www.google.com to retrieve information.

There's actually an official list of assigned port numbers maintained by the Internet Assigned Numbers Authority (IANA) at www.iana.org. On Linux or UNIX systems, you can often find a version of this list in `/etc/services`.

If you're writing a server for a service that isn't on that list, you should pick a port number that's greater than 1024 and doesn't occur on your list. That will make it least likely to conflict with any other server. Port numbers can be as high as 65535.

Traditionally, on Linux or UNIX systems, only the root user is able to request a port number that's less than 1024. Although this certainly isn't foolproof, it does provide some level of security, and you should be aware that your servers must at least start as root if they want a port that's less than 1024.

Client-Side Port Numbers

In general, the port number of a client is unimportant. Usually, the client will let the operating system pick a random port number. The client's system assigns a so-called ephemeral (short-lived) port number that's guaranteed to be unused on that machine. When the server receives a connection request, it can find out the client's port number from that request. Data is sent to that port number. Therefore, the server can work with whatever port number the client chooses.

Understanding User Datagram Protocol

Thus far, we've been discussing TCP, but there is actually another common protocol in use: UDP, which is used for sending very short messages from one system to another. It provides at most one guarantee: that the data that you receive will be intact. It doesn't guarantee that data will actually be received, that it will be received only once, or that different messages will be received in the order that they were sent. But data integrity is usually assured unless an attacker is attempting to bypass security.

The advantage of UDP is that, because it doesn't have to provide all those guarantees, it has a lower overhead than TCP, which takes some time to establish and shut down a connection, whereas UDP has no notion of a connection, so that problem doesn't exist.

UDP is typically used for situations in which a client asks a server for one bit of information and is capable of resending its request if it doesn't receive an answer. The most popular UDP application is the DNS system. Because clients typically need to send a single brief request and receive a single brief answer, UDP is well suited for the task. UDP is also used for streaming audio and video applications, because TCP's overhead in dealing with dropped packets may actually make the audio sound worse than it would when UDP drops an occasional packet. Games and networked file systems such as NFS and Samba are also heavy users of UDP.

Here are some guidelines to help you choose whether to use TCP or UDP for your protocols. They may not be applicable in all situations, but they're good starting points.

You should use TCP if:

- You need a reliable data transport that ensures that your data arrives complete and intact.
- Your protocol requires more than just a single request and a single response from the server.
- You need to send more than a small amount of data across the network.
- A small delay establishing the initial connection is tolerable.

You should use UDP if:

- You're unconcerned if some packets don't arrive or arrive out of order, or you can detect and handle this condition yourself.
- Your protocol consists of brief requests and responses.
- You need the conversation setup to be as quick as possible.
- Only a small amount of data is being sent. UDP is limited to 64 KB of data in a single packet, but people often stay below 1 KB when using UDP.

Except for DNS, none of the network protocols covered in this book use UDP; however, Chapters 2 and 3 provide information on writing code for UDP services.

Understanding Physical Transports and Ethernet

One of the strengths of TCP/IP is that it can send data over so many different types of physical network hardware. Some common ones include Ethernet, dial-up serial links with PPP, token-ring networks, DSL lines, cable modems, satellite links, cellular phones, and leased lines such as a T1.

Each of these different types of communication have their own unique properties, and there are certain properties that they all share. Some types of connections, such as PPP, are typically used to connect one machine to another machine. Other connections, such as Ethernet, are used to make it possible for many computers at a single location to communicate.

Sometimes developers need to take advantage of special features of a specific network transport. These features are sometimes available in the TCP/IP layer.

Ethernet is one of the most common physical transports in use today. Many different protocols can be run over an Ethernet network; TCP/IP is but one family that it supports. There are some unique features of Ethernet networks that are of interest to application developers. The primary one is the ability to broadcast a packet to all workstations on the local network. This can be used to advertise the existence of a service to systems designed to listen for such broadcasts, or to do things like broadcast warning messages to PCs.

A computer running TCP/IP on an Ethernet network has an IP address associated with each network interface. To communicate with another machine on the same network, the computer can just send the information directly to the other machine. To communicate with an outside machine on the Internet, the computer has to send the information to a router on the local network, which determines where the packet should be sent next.

In order to know which machines are local and which are remote, the networking software checks to see whether the leading bits (most significant parts) of the source and destination IP addresses are the same. There is a *netmask* configured on each interface that describes how many bits must be compared. If the comparison fails, and the leading bits are different, the packet must go through the router. Anything else within that range is reachable directly, both by broadcast packets and by normal (directed) transmissions.

Networking in Python

As you write network-aware programs in Python, you'll find that they generally fall into one of two categories: programs that can use a protocol module (such as HTTP or FTP) already available in Python, and programs that require you to write the protocol from scratch. Even if your programs all use existing modules, you'll find it helpful to have an understanding of what is going on under the hood. Let's take a look.

Low-Level Interface

Python provides a full interface to the underlying operating system's socket interface, which gives you a tremendous amount of flexibility and power if you need it. Python also provides access to services such as SSL/TLS for encrypted and authenticated communication channels. If you're accustomed to network programming in C, you'll find Python's socket services very familiar. Part I of this book covers this low-level network programming.

Basic Client Operation

Here's a simple client in Python:

```
#!/usr/bin/env python
# Simple Gopher Client - Chapter 1 - gopherclient.py

import socket, sys

port = 70                      # Gopher uses port 70
host = sys.argv[1]
filename = sys.argv[2]

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((host, port))

s.sendall(filename + "\r\n")

while 1:
    buf = s.recv(2048)
    if not len(buf):
        break
    sys.stdout.write(buf)
```

This is one of the simplest examples of an actual, real-world network protocol implementation that you'll find. It implements the Gopher Protocol, a simple one that was in widespread use on the Internet prior to the introduction of the Web. This program takes two command-line arguments: a hostname and a filename, and requests the appropriate document from the host.

The operation is simple. It creates a socket with the call to `socket.socket()`. The arguments tell the system that it wants an Internet socket for stream (TCP) communication. Then, the program connects to the remote host and transmits the filename. Finally, it reads back the response and prints it to the screen.

Try it out. Save the previous example as `gopherclient.py`, then run a command such as `./gopherclient.py quux.org /` and you'll get back a listing of the root directory on a Gopher server.

Errors and Exceptions

People familiar with network programming in C may at first think that this program has no error detection at all. That's actually not the case. Python takes care of error detection for you and raises an exception if something bad happens. Try giving it an invalid hostname, as follows:

```
$ ./gopherclient.py nonexistant.example.com /
Traceback (most recent call last):
  File "./gopherclient.py", line 11, in ?
    s.connect((host, port))
  File "<string>", line 1, in connect
socket.gaierror: (-2, 'Name or service not known')
```

Python detected the error and generated a `socket.gaierror` exception (though the associated text and number might vary). Because the program doesn't specifically handle that exception, the program terminated and printed out information about where the error occurred and what the problem was. You can make it a little friendlier like this:

```
#!/usr/bin/env python
# Simple Gopher Client with basic error handling - Chapter 1
# gopherclient2.py

import socket, sys

port = 70
host = sys.argv[1]
filename = sys.argv[2]

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
try:
    s.connect((host, port))
except socket.gaierror, e:
    print "Error connecting to server: %s" % e
    sys.exit(1)
```

```
s.sendall(filename + "\r\n")

while 1:
    buf = s.recv(2048)
    if not len(buf):
        break
    sys.stdout.write(buf)
```

If you try to connect to a nonexistent server this time, you'll get a more friendly error message and the program will terminate.

Of course, other parts of the program could generate exceptions, and it's up to you exactly how many of them you're going to care about.

C programmers might also take note of the `s.sendall()` call. You may be familiar with the C `send()` call, which doesn't guarantee that all of your data gets sent. C programmers typically will have a loop to make sure all the data goes out. Python does provide a `send()` function, but `sendall()` is frequently more convenient. If an error occurs, it will raise an exception; otherwise, you'll know your message was sent.

File-like Objects

Python programmers will be familiar with methods of file objects: `readline()`, `write()`, `read()`, and other similar ones. The Python library has support all over for file and file-like objects. Socket objects don't implement the same interface, and you might at first be concerned that you'll lose a lot of convenience because of that. However, Python socket objects do have a `makefile()` call that generates a file-like object for you to use. Here is the same program as before, rewritten to use that file-like interface:

```
#!/usr/bin/env python
# Simple Gopher Client with file-like interface - Chapter 1
# gopherclient3.py

import socket, sys

port = 70
host = sys.argv[1]
filename = sys.argv[2]
```

```

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((host, port))
fd = s.makefile('rw', 0)

fd.write(filename + "\r\n")

for line in fd.readlines():
    sys.stdout.write(line)

```

This program operates exactly the same way as the previous version as far as the user is concerned. Looking at the source, the first difference occurs with the call to `makefile()`. That function takes two optional arguments: a mode and a buffering mode. The mode indicates whether you'll be reading, writing, or doing both; in this case, the program does both, so it says '`'rw'`'. Buffering is often used with disk files, but for interactive network programs, it frequently can get in your way, so it's a good idea to always specify 0 here and disable it.

Now that a file-like object is at hand, you can use more familiar methods. Two shown here are `write()` and `readlines()`. They work just like they would with regular file objects.

Basic Server Operation

The first example of a Python network client took only 20 lines. It's also quite easy to write a basic Python server, as follows:

```

#!/usr/bin/env python
# Simple Server - Chapter 1 - server.py
import socket

host = ''                      # Bind to all interfaces
port = 51423

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind((host, port))
s.listen(1)

print "Server is running on port %d; press Ctrl-C to terminate." \
      % port

```

```

while 1:
    clientsock, clientaddr = s.accept()
    clientfile = clientsock.makefile('rw', 0)
    clientfile.write("Welcome, " + str(clientaddr) + "\n")
    clientfile.write("Please enter a string: ")
    line = clientfile.readline().strip()
    clientfile.write("You entered %d characters.\n" % len(line))
    clientfile.close()
    clientsock.close()

```

Let's take a quick look at the code. Like the client example, the first thing to do is to create a socket with a call to `socket.socket()`. Then, for the ease of running this example, the socket is marked reusable. This is optional, and you can read more about socket options in later chapters. The next step is to bind to a port. For this example, I picked port 51423, but you could pick any port above 1024. The host is set to the empty string, which means that the program accepts connections from anywhere. Then, the call to `listen()` tells it to start looking for connections from clients, and to let at most one connection wait to be processed. Real servers will probably use a higher number.

The main loop starts with a call to `accept()`. The program will stop right there until a client connects to it. When a client connects, `accept()` returns two pieces of information: a new socket connected to the client and the IP address and port number of the client. For this example, file-like objects are used, so one is made in the same manner as it was for the earlier example. The server then writes out some introductory messages, reads a string from the client, writes out a response, and finally closes the client socket. It's important to close that socket here because otherwise the client might not know the server is done talking to it, and your server might accumulate lots of old client connections. When using file-like objects, you must close *both* the file object *and* the socket object.

You can try this out. First, you need to fire up the server. To do that, you can just save the example as `server.py`, then run a command such as `./server.py`. Now open another terminal or telnet application and connect to port 51423 on localhost. On Linux or UNIX-like platforms, your telnet session might look like this:

```

$ telnet localhost 51423
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^>'.
Welcome, ('127.0.0.1', 48665)
Please enter a string: Hello
You entered 5 characters.
Connection closed by foreign host.

```

You might note that I didn't actually implement the Telnet Protocol here, but telnet clients are often capable of communicating anyway. Also, though this basic server does work, it isn't yet particularly useful; you'll learn much more about writing servers in Chapter 3 and Parts V and VI.

High-Level Interface

Although the examples you've just seen show how to implement your own protocols in Python, you might be using a common protocol such as HTTP or IMAP often. You might not need to work with the network at such a low level. Python also provides many protocol modules that can dramatically simplify your programming task. For instance, rather than having to write code to parse and understand HTTP headers, the `httplib` module in Python will do it for you. In effect, you're starting with half the job already done.

The earlier examples in this chapter showed how you would write a module from scratch to communicate with a Gopher server. Now let's take a look at how you might use a higher-level module to do the same thing:

```
#!/usr/bin/env python
# High-Level Gopher Client - Chapter 1 - gopherlibclient.py

import gopherlib, sys
host = sys.argv[1]
file = sys.argv[2]

f = gopherlib.send_selector(file, host)
for line in f.readlines():
    sys.stdout.write(line)
```

That saved some work! The `gopherlib` module actually takes care of the details of creating the socket and establishing the connection. In fact, it even takes care of calling `makefile()` for you before `send_selector()` returns.

You can get to an even higher level with Python modules. For handling URLs, Python provides some modules that actually let you write code that works with several protocols. Here's an example:

```

#!/usr/bin/env python
# High-Level Gopher Client with urllib - Chapter 1 - urlclient.py

import urllib, sys
host = sys.argv[1]
file = sys.argv[2]

f = urllib.urlopen('gopher://%s%s' % (host, file))
for line in f.readlines():
    sys.stdout.write(line)

```

This example simply takes the command-line arguments, builds a URL, and passes it on to `urllib`. You could use `urllib` to make a general-purpose file downloader with just a couple of lines of code. Try the following:

```

#!/usr/bin/env python
# Chapter 1 - Download Example - download.py

import urllib, sys

f = urllib.urlopen(sys.argv[1])
while 1:
    buf = f.read(2048)
    if not len(buf):
        break
    sys.stdout.write(buf)

```

Now you can save this example as `download.py` and run a command such as `./download.py gopher://quux.org/` to get the same output as before. Or, you could use it for entirely new purposes. For instance, perhaps you'd like to view a compressed file from the Web. On Linux and UNIX systems, you might run the following:

```
./download.py http://http.us.debian.org/debian/ls-lR.gz | gunzip | more
```

This command will decompress the data and send it on. And you wrote only 11 lines of code.

Summary

TCP/IP networking can occur over many different types of transport, such as modems and Ethernet. Each endpoint is identified uniquely by an IP address and a port number.

Servers listen for connections on port numbers that are known in advance. When a client connects, it usually lets the operating system choose a port number instead of explicitly selecting one.

There are two common protocols for transmitting data: TCP, which provides reliability and a full conversation, and UDP, which is faster for small, brief conversations.

Most people writing network programs in Python will either design their own protocol from scratch or use one of the many built-in modules that implement existing protocols. For those who design a protocol from scratch, Python provides a full-featured socket interface that C network programmers will find familiar.

Network Clients

AS YOU WRITE programs to use network services, you'll most frequently find yourself writing network clients. In this chapter, you'll learn how to implement an application protocol on the client side. This information will be useful if Python doesn't already have a module implementing your protocol or if you want to modify or extend an existing Python module.

Understanding Sockets

Sockets are an extension to the operating system's I/O system that enable communication between processes and machines. To fully understand how sockets work on a modern system, a bit of historical perspective will help.

Sockets as they are typically used today originated with the BSD UNIX-like operating system. On a UNIX system such as BSD, there are certain existing system calls that work with file descriptors. Such system calls include `open()`, `read()`, `write()`, and `close()`. A file descriptor typically referred to a file or some file-like entity.

When support for networking was to be added to the operating system, it was done in a way that extended the existing file descriptor architecture. New system calls were added to work with sockets, and many existing system calls were now able to work with sockets as well. A socket, then, lets you use standard operating-system calls to talk to other machines or other processes on your own machine.

In some ways, a socket can be treated the same as a standard file descriptor. On UNIX-like platforms, system calls such as `read()`, `write()`, `dup()`, `dup2()`, and `close()` will work for sockets as well as for standard file descriptors. In many ways, a program need not know whether it's writing data to a file, the terminal, or a TCP connection.

However, there are ways that a socket is different. The most obvious is the way in which a socket is created. While many files are opened with the `open()` call, sockets are created with the `socket()` call and additional calls are needed to connect and activate them. The system calls `recv()` and `send()` exist as counterparts to `read()` and `write()`. The `send()` and `recv()` calls provide additional socket-specific functionality.

Python provides an interface to the operating system's socket library through the `socket` module. You'll typically make calls to functions and constants in this module when you set up your sockets.

Creating Sockets

For a client program, creating a socket is generally a two-step process. First, you need to create the actual socket object. Second, you need to connect it to the remote server.

When you create a socket object, you need to tell the system two things: the communication type and the protocol family. The communication type specifies the underlying protocol used to transmit data. Examples of protocols include IPv4 (current Internet standard), IPv6 (future Internet standard), IPX/SPX (NetWare), and AFP (Apple file sharing). By far the most common is IPv4. The protocol family defines how data is transmitted.

For Internet communications, which make up the bulk of this book, the communication type is almost always `AF_INET` (corresponding to IPv4). The protocol family is typically either `SOCK_STREAM` for TCP communications or `SOCK_DGRAM` for UDP communications. For a TCP connection, creating a socket generally uses code like this:

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

To connect the socket, you'll generally need to provide a tuple containing the remote hostname or IP address and the remote port. Connecting a socket typically looks like this:

```
s.connect(("www.example.com", 80))
```

The following program will establish a connection and then terminate. It's not very useful yet, but it's a fully functional example, as shown here:

```
#!/usr/bin/env python
# Basic Connection Example - Chapter 2 - connect.py

import socket

print "Creating socket...",
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
print "done."

print "Connecting to remote host...",
s.connect(("www.google.com", 80))
print "done."
```

This example connects to the web server on `www.google.com`, prints some status messages along the way, and then terminates.

NOTE The C `connect()` function requires an IP address for the remote.

In Python, the `connect()` method of a socket object will, if necessary, automatically resolve a hostname to an IP address using DNS. However, it will not do the same for the port number.

Finding the Port Number

In Chapter 1, you learned that there's a list of well-known server port numbers. Most operating systems will ship with a version of that list, which you can query. The Python socket library includes a function `getservbyname()` that will let you query this in an automated fashion. On UNIX systems, you can often find this list in `/etc/services`.

To query the list, you need two parameters: a protocol name and a port name. A port name is a string such as `http` that can be converted into a port number. Here's a modified example of the previous program that uses a port name instead of the number.

```
#!/usr/bin/env python
# Revised Connection Example - Chapter 2 - connect2.py

import socket

print "Creating socket...",
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
print "done."

print "Looking up port number...",
port = socket.getservbyname('http', 'tcp')
print "done."

print "Connecting to remote host on port %d..." % port,
s.connect(("www.google.com", port))
print "done."
```

As you can see, in this example, you didn't have to know in advance that HTTP uses port 80. Although you could have looked it up and hard-coded it in your program, it's often nice for interactive programs to be able to read textual port descriptions from users.

In this example, TCP is being used, so the string "tcp" is passed to `socket.getservbyname()`. If you were using UDP, you would instead pass "udp".

Getting Information from a Socket

Once you've established a socket connection, you can find out some useful information from it. Here's an example showing off some of these capabilities:

```
#!/usr/bin/env python
# Information Example - Chapter 2 - connect3.py

import socket

print "Creating socket...",
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
print "done."

print "Looking up port number...",
port = socket.getservbyname('http', 'tcp')
print "done."

print "Connecting to remote host on port %d..." % port,
s.connect(("www.google.com", port))
print "done.

print "Connected from", s.getsockname()
print "Connected to", s.getpeername()
```

When you run this program, you'll see two new lines of information. The first one will show your own IP address and port number; the second will show the IP address and port number of the remote machine. As you recall from Chapter 1, for clients, your port number is assigned by the operating system (and may be random), so you'll likely observe it to be different each time you run the program.

Communicating with Sockets

Now that the connection has been established, it's time to send and receive data across it. Python provides two ways to do that: socket objects and file-like objects.

Socket objects provide interfaces to the operating system's `send()`, `sendto()`, `recv()`, and `recvfrom()` calls. File-like objects provide a more traditional Python interface with calls such as `read()`, `write()`, and `readline()`.

Socket objects are typically useful if you have special requirements. Examples of these requirements may be working with protocols that require you to have fine-grained control over when data is read and written, binary protocols where fixed-size chunks of data are transmitted, instances where timeouts for data require special handling, or anything else that involves more than simple reading and writing. Socket objects are also preferable when you're working with UDP programs.

File-like objects typically are more useful if you're working with line-oriented protocols, since they handle much of that parsing for you automatically by providing `readline()` functions. However, file-like objects tend to work best with TCP connections only; they don't fare well with UDP communication. That's because TCP connections behave more like standard files—they guarantee accuracy of data received and behave as a stream of bytes just like a file. UDP doesn't behave as a stream of bytes like a file; instead, it's a packet-based communication. File-like objects provide no way to operate on a per-packet basis. Therefore, the normal mechanisms for building, sending, and receiving UDP packets will not work, and error detection will be very difficult.

Basic examples of both types of communication were provided in Chapter 1. Throughout the rest of this chapter, you'll see both types of communication employed.

Handling Errors

In Python, the socket code raises exceptions when network errors occur. There are so many conditions in which network communications can cause program errors that networking programs can't afford to ignore them. Virtually every function call that touches the network in any way can and does raise exceptions for various reasons—servers being down, connections dropped, and so on.

The proper response to errors depends on your application. For instance, if the connection gets dropped in the middle of a file download, the proper action may be to attempt to restart the download at that point.

Socket Exceptions

Different network calls can raise different exceptions. The following example shows how to catch every common type of exception when dealing with socket objects. The example program takes three command-line arguments: a host to which it will connect, a port number or name on the server, and a file to request from the server. The program will connect to the server, send a simple HTTP request for the given filename, and display the result. Along the way, it exercises care to handle various types of potential errors.

```
#!/usr/bin/env python
# Error Handling Example - Chapter 2 - socketerrors.py

import socket, sys

host = sys.argv[1]
textport = sys.argv[2]
filename = sys.argv[3]

try:
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
except socket.error, e:
    print "Strange error creating socket: %s" % e
    sys.exit(1)

# Try parsing it as a numeric port number.

try:
    port = int(textport)
except ValueError:
    # That didn't work, so it's probably a protocol name.
    # Look it up instead.
    try:
        port = socket.getservbyname(textport, 'tcp')
    except socket.error, e:
        print "Couldn't find your port: %s" % e
        sys.exit(1)
```

```

try:
    s.connect((host, port))
except socket.gaierror, e:
    print "Address-related error connecting to server: %s" % e
    sys.exit(1)
except socket.error, e:
    print "Connection error: %s" % e
    sys.exit(1)

try:
    s.sendall("GET %s HTTP/1.0\r\n\r\n" % filename)
except socket.error, e:
    print "Error sending data: %s" % e
    sys.exit(1)

while 1:
    try:
        buf = s.recv(2048)
    except socket.error, e:
        print "Error receiving data: %s" % e
        sys.exit(1)
    if not len(buf):
        break
    sys.stdout.write(buf)

```

In this program, the exception handlers simply print out a nice message and then terminate. It catches all the network-related exceptions that can possibly be raised in this program. Python's socket module actually defines four possible exceptions:

- `socket.error` for general I/O and communication problems
- `socket.gaierror` for errors looking up address information
- `socket.herror` for other addressing errors (corresponding to `h_errno` in C)
- `socket.timeout` for handling timeouts that occur after `settimeout()` has been called on a socket (Python 2.3 and higher)

In the previous example, you should take special note of the call to `connect()`. Since the program lets it resolve the hostname into an IP address, you could actually see two different errors raised: `socket.gaierror` if the hostname was bad, and `socket.error` if there is a problem connecting to the remote hostname.

Missed Errors

There's a problem with the error handling in this program. There are certain situations in which communication problems could occur but no exception would be raised because no error was passed back from the operating system.

One such problem could occur if the remote server drops the connection between the time the client connects and the time it writes out its request. In this case, the later call to `recv()` will receive no data (since the server closed its connection) and the program will terminate successfully. That's misleading at best.

For many operating systems, calls that send data across the network will sometimes return before the remote server is guaranteed to have received the information. Thus, it's possible that data from a successful call to `sendall()` was, in fact, never actually received.

To correct this problem, once you're completely done writing, you can use the `shutdown()` call. This will force buffers to be flushed and will raise an exception if there was an error at any point.

More details about `shutdown()` will be presented in Chapter 5. The following example expands on the earlier one and shows a simple way to use `shutdown()` in order to ensure that the server properly received the request:

```
#!/usr/bin/env python
# Error Handling Example With Shutdown - Chapter 2 - shutdown.py

import socket, sys, time

host = sys.argv[1]
textport = sys.argv[2]
filename = sys.argv[3]

try:
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
except socket.error, e:
    print "Strange error creating socket: %s" % e
    sys.exit(1)

# Try parsing it as a numeric port number.
```

```

try:
    port = int(textport)
except ValueError:
    # That didn't work. Look it up instead.
    try:
        port = socket.getservbyname(textport, 'tcp')
    except socket.error, e:
        print "Couldn't find your port: %s" % e
        sys.exit(1)

try:
    s.connect((host, port))
except socket.gaierror, e:
    print "Address-related error connecting to server: %s" % e
    sys.exit(1)
except socket.error, e:
    print "Connection error: %s" % e
    sys.exit(1)

print "sleeping..."
time.sleep(10)
print "Continuing."

try:
    s.sendall("GET %s HTTP/1.0\r\n\r\n" % filename)
except socket.error, e:
    print "Error sending data: %s" % e
    sys.exit(1)

try:
    s.shutdown(1)
except socket.error, e:
    print "Error sending data (detected by shutdown): %s" % e
    sys.exit(1)

while 1:
    try:
        buf = s.recv(2048)
    except socket.error, e:
        print "Error receiving data: %s" % e
        sys.exit(1)
    if not len(buf):
        break
    sys.stdout.write(buf)

```

If you run your own server of some type (such as a web server or one of the examples in Chapter 3), this program will let you see for yourself exactly where write errors are detected. Different operating systems and different servers may behave differently. It's important to remember that you must continue to be prepared to handle exceptions raised by `write()` even if, during your own testing, those exceptions are always raised by `shutdown()`, since some operating systems cause exceptions at different times.

Run it and connect it to your server, then kill your server as soon as the client connects. (It's best to use a server example from this book, such as `basicttp.py` from Chapter 16; some others, such as popular web servers, don't actually die within ten seconds.) After ten seconds, the client will try to write its request to the server, but will get an error. Here's the output I received:

```
$ ./shutdown.py localhost 8765 /test-ignore.html
sleeping...
Continuing.
Error sending data (detected by shutdown): (107, 'Transport endpoint
is not connected')
```

Indeed, the exception was not raised by the `sendall()` but by `shutdown()`! `sendall()` returned immediately, but `shutdown()` waited until it could return an accurate exit code to you.

NOTE Variations between operating systems may affect the result of this example. Some operating systems may not cause an error at all. Unfortunately, Python programs are at the mercy of the underlying system's implementation.

For some protocols—those in which you do more than one write at the beginning—it's not practical to `shutdown()` after each write. That's usually not a problem, though; you'll get an exception for the error eventually, just not immediately. You should still perform a shutdown after your last write, though, in order to make sure that all your writes have succeeded up to that point. And always remember that data isn't guaranteed to be sent until you call `shutdown()`. If you do need to know about problems earlier, Python socket timeouts may be useful; see Chapter 5 for information on timeouts.

Errors with File-like Objects

As you saw in Chapter 1, it's possible to use the `makefile()` function to get a file-like object from a socket. This file-like object actually makes calls to the real socket, so the exceptions raised by the file-like object are the same as the ones raised by the socket's own `send()` and `recv()` functions.

Here's the latest `shutdown()` example modified to use file-like objects:

```
#!/usr/bin/env python
# Error Handling Example With Shutdown and File-like Objects - Chapter 2
# shutdownfile.py

import socket, sys, time

host = sys.argv[1]
textport = sys.argv[2]
filename = sys.argv[3]

try:
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
except socket.error, e:
    print "Strange error creating socket: %s" % e
    sys.exit(1)

# Try parsing it as a numeric port number.

try:
    port = int(textport)
except ValueError:
    # That didn't work. Look it up instead.
    try:
        port = socket.getservbyname(textport, 'tcp')
    except socket.error, e:
        print "Couldn't find your port: %s" % e
        sys.exit(1)

try:
    s.connect((host, port))
except socket.gaierror, e:
    print "Address-related error connecting to server: %s" % e
    sys.exit(1)
except socket.error, e:
    print "Connection error: %s" % e
    sys.exit(1)
```

```
fd = s.makefile('rw', 0)

print "sleeping..."
time.sleep(10)
print "Continuing.

try:
    fd.write("GET %s HTTP/1.0\r\n\r\n" % filename)
except socket.error, e:
    print "Error sending data: %s" % e
    sys.exit(1)

try:
    fd.flush()
except socket.error, e:
    print "Error sending data (detected by flush): %s" % e
    sys.exit(1)

try:
    s.shutdown(1)
    s.close()
except socket.error, e:
    print "Error sending data (detected by shutdown): %s" % e
    sys.exit(1)

while 1:
    try:
        buf = fd.read(2048)
    except socket.error, e:
        print "Error receiving data: %s" % e
        sys.exit(1)
    if not len(buf):
        break
    sys.stdout.write(buf)
```

There are two new things to point out about this example. First, notice the call to `flush()`. Technically, since the call to `makefile()` wisely specified no buffer, this isn't necessary, but if you're using buffering for some reason, it may be. Since it will call `send()` underneath, it can still raise an exception.

NOTE Handling errors with buffering on file-like objects can be even trickier, since you don't have control over exactly when the attempt to send data is made. I recommend avoiding buffering for file-like objects.

Second, notice that the code preserves the socket object `s` even after calling `makefile()`. The object returned by `makefile()` doesn't provide a `shutdown()` call, so you must preserve the original socket object and use it.

Using User Datagram Protocol

Thus far, this chapter has focused on TCP communications, but you can also communicate using UDP.

UDP communications almost never use file-like objects because they tend not to provide sufficient control over how data is sent and received. Let's first introduce a basic UDP client:

```
#!/usr/bin/env python
# UDP Example - Chapter 2 - udp.py

import socket, sys

host = sys.argv[1]
textport = sys.argv[2]

s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
try:
    port = int(textport)
except ValueError:
    # That didn't work. Look it up instead.
    port = socket.getservbyname(textport, 'udp')

s.connect((host, port))
print "Enter data to transmit: "
data = sys.stdin.readline().strip()
s.sendall(data)
print "Looking for replies; press Ctrl-C or Ctrl-Break to stop."
while 1:
    buf = s.recv(2048)
    if not len(buf):
        break
    sys.stdout.write(buf)
```

This example program takes two command-line arguments—a hostname and a port number for a server. It will connect to the server, then prompt you for a line of text to send. The data is sent, and then it enters an infinite loop looking for replies. You'll need to terminate the program with Ctrl-C or Ctrl-Break since it has no way of knowing when the server is done sending replies. Here's an example of how to use this program to connect to the UDP echo server from Chapter 3:

```
$ ./udp.py localhost 51423
Enter data to transmit:
Hello, echo server. How are you today?
Looking for replies; press Ctrl-C to stop.
Received: Hello, echo server. How are you today?
Traceback (most recent call last):
  File "./udp.py", line 23, in ?
    buf = s.recv(2048)
KeyboardInterrupt
```

The program sent one UDP packet, received one UDP answer, and continued waiting for others (which never arrived). Finally, it was terminated with Ctrl-C, which caused the KeyboardInterrupt.

Let's take a look at the differences between this example and the TCP client.

First, notice that when the socket is created, the program asks for SOCK_DGRAM instead of SOCK_STREAM; this indicates to the operating system that the socket will be used for UDP instead of TCP communications.

Second, the call to `socket.getservbyname()` looks for UDP port numbers rather than TCP port numbers. A port number is specific to a protocol; so if TCP uses port 119, a completely different UDP application could use the same port.

Third, the program has no way of detecting when the server is done sending data. That is because there's no actual connection here. The call to `connect()` did nothing but initialize some internal parameters. Also, the server may not send back any data or the data may be lost in transit; this program doesn't have the intelligence to figure that out. Therefore, you have to press Ctrl-C when you're done waiting for incoming packets.

You can test the preceding code by starting `udpechoserver.py` from Chapter 3. Then call `./udp.py localhost 51423` to connect to the UDP echo server on your own machine.

It's possible to use UDP without ever calling `connect()` at all. Here's a program that demonstrates that:

```

#!/usr/bin/env python
# UDP Connectionless Example - Chapter 2 - udptime.py

import socket, sys, struct, time

hostname = 'time.nist.gov'
port = 37

host = socket.gethostbyname(hostname)

s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.sendto('', (host, port))

print "Looking for replies; press Ctrl-C to stop."
buf = s.recvfrom(2048)[0]
if len(buf) != 4:
    print "Wrong-sized reply %d: %s" % (len(buf), buf)
    sys.exit(1)

secs = struct.unpack("!I", buf)[0]
secs -= 2208988800
print time.ctime(int(secs))

```

This program is a demonstration of the simple time protocol documented in RFC868. With the `sendto()` call, the program sends an empty string to the server on `time.nist.gov`. Note that no call to `connect()` ever occurred.

The reply can be received like normal, but with `recvfrom()` instead of `recv()`. In general, when using unconnected UDP communication, `recv()` doesn't provide enough information for communication. The `recvfrom()` call actually returns a tuple with two pieces of data: the actual data received and the address of the machine that sent it. In this case, we don't care about the sending address, so only the string is saved out of that reply. Servers will care about it, and can actually use this mechanism to handle requests from multiple clients simultaneously. Using `recvfrom()`, the program receives the reply, which is a binary-encoded number of seconds since January 1, 1900 (see Chapter 5 for more information about decoding this type of data). It then unpacks it, converts it to a UNIX time format by subtracting the number of seconds between 1900 and 1970, and then finally prints it to your screen. If you run this program, it should show you the correct time. Since UDP doesn't guarantee that a packet gets delivered successfully, and this program doesn't make any attempt to check that, you may need to run it more than once before you receive a response. Also, some firewalls may block UDP communication; in that case, you won't receive a reply at all.

Summary

The basic interface for network communication is the socket, which extends the operating system's basic I/O system to handle network communications. A socket can be created with `socket()` and connected with `connect()`. Given a socket, you can determine the IP address and port number of both the local and remote endpoints. Sockets are a generic interface to different protocols and can handle both TCP and UDP communication.

When communicating over many thousands of miles, many different things can go wrong, so error checking is important. Most networking-related calls can raise exceptions, though sometimes not right away. Use `shutdown()` to make sure you're informed of write errors.

Python provides two interfaces for working with sockets: the standard socket interface, used for UDP and advanced TCP purposes, and the file-like interface, used for simple TCP communication.

Many of the concepts covered in this chapter also will apply to Chapter 3, which covers network servers. These two chapters, taken together, will provide you with enough information to design your own complete client/server protocol system.

CHAPTER 3

Network Servers

IN CHAPTER 2, you learned how to write network clients. You created a socket, connected it to a server, and then communicated with it.

In this chapter, you'll learn how to write network servers. Servers typically wait for requests from clients and then transmit answers back. In general, servers do everything from handing out web pages (sometimes generating them on the fly) to exchanging e-mail. There are several examples of server programs in this chapter. You'll see how to implement a server that performs a simple calculation as well as one that acts as a network time server.

In some respects, server programming is similar to client programming. Many commands that you're familiar with in network-client programming are available for servers as well because servers use the same socket interface that clients do.

However, there are some important details that differ, most notably with setting up the socket. Also, there are some other issues to consider that crop up more infrequently on the client side. This chapter will show you how to build a server from scratch, how to find out information about the clients, how to log activity, and how to run your server in different ways. You'll be able to write a fully functional server by the time you reach the end of the chapter.

Preparing for Connections

For a client, the process of establishing a TCP connection is a two-step process that includes the creation of the socket object and a call to `connect()` to establish a connection to the server.

For a server, the process requires the following four steps:

1. Create the socket object.
2. Set the socket options (optional).
3. Bind to a port (and, optionally, a specific network card).
4. Listen for connections.

Here's a code snippet that does those things:

```
host = ''                                # Bind to all interfaces
port = 51423

# Step 1 (Create the socket object)
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Step 2 (Set the socket options)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

# Step 3 (Bind to a port and interface)
s.bind((host, port))

# Step 4 (Listen for connections)
s.listen(5)
```

Your server socket will now be ready to use. Let's take a look at what this code does.

Creating a Socket Object

To create your socket object, use exactly the same command (shown here) as you did for a client:

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

You'll be using the same socket objects that were in use with the client.

Setting and Getting Socket Options

There are many different options that can be set for a socket. For general-purpose servers, the socket option of greatest interest is called `SO_REUSEADDR`. Normally, after a server process terminates, the operating system reserves its port for a few minutes, thereby preventing any other process (even another instance of your server itself) from opening it until the timeout expires. If you set the `SO_REUSEADDR` flag to true, the operating system releases the server port as soon as the server socket is closed or the server process terminates. Doing this makes it much easier to debug and test programs, so all the examples in this book set that option.

The `SO_REUSEADDR` option is set like this:

```
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
```

Python defines `setsockopt()` and `getsockopt()` like this:

```
setsockopt(level, optname, value)
getsockopt(level, optname[, buflen])
```

The `value` parameter has a meaning that's determined by the `level` and `optname` parameters. The `level` defines which set of options is to be used. Normally, it's `SOL_SOCKET`, which means that you're working with socket options. It can also be set to a specific protocol number to set protocol options. However, most protocol options are specific to a given operating system, so they're rarely used in applications designed to be portable. For more details on the protocol options available for your operating system, please consult your operating system's reference for `setsockopt()`.

Assuming you use `SOL_SOCKET` for the `level`, you'll find that the `optname` parameter provides the specific option to use. The set of available options varies somewhat from one operating system to the next and is summarized here. For `getsockopt()`, a `buflen` shouldn't be specified if an integer return value is expected. If a string is expected, `buflen` must be specified and gives the maximum length of a string you're willing to accept. Boolean values are defined as 0 for false and nonzero for true.

Table 3-1 lists the most common options available for `SOL_SOCKET`.

Table 3-1. Option Names for `setsockopt()` and `getsockopt()`

Option	Meaning	Expected Value
<code>SO_BINDTODEVICE</code>	Causes the socket to be valid on a particular network interface (network card) only. May not be portable.	A string giving the device name, or an empty string to return to default behavior
<code>SO_BROADCAST</code>	Permits the transmission and reception of packets to the broadcast address. Only valid for UDP. See Chapter 5 to learn how to send and receive broadcast packets.	Boolean integer
<code>SO_DONTROUTE</code>	Prohibits outgoing packets from crossing any router or gateway. This is most frequently useful for UDP communication on an Ethernet LAN as a security measure. It prevents data from leaving the local network, no matter what IP address is used for the destination.	Boolean integer

Table 3-1. Option Names for `setsockopt()` and `getsockopt()` (Continued)

Option	Meaning	Expected Value
<code>SO_KEEPALIVE</code>	Enables the transmission of keep-alive packets for TCP connections. These packets help both endpoints of the connection verify that the connection remains open even when no data is passing through it.	Boolean integer
<code>SO_OOBINLINE</code>	Causes incoming out-of-band data to be treated as if it were normal data; that is, it will be receivable via a standard call to <code>recv()</code> .	Boolean integer
<code>SO_REUSEADDR</code>	The port number used by the local endpoint of this socket becomes immediately reusable after the socket is closed. Normal behavior prevents it from being reused for a system-defined amount of time.	Boolean integer
<code>SO_TYPE</code>	Retrieves the socket type (such as <code>SOCK_STREAM</code> or <code>SOCK_DGRAM</code>). <code>getsockopt()</code> only.	Integer

Additional options are likely available for your operating system. Please consult your operating system's socket reference for more details. Users of Linux and UNIX systems can often find this information in the `socket(7)` manpage (run `man 7 socket`) or the `setsockopt(2)` manpage. Windows users can find this information at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winsock/winsock/setsockopt_2.asp. The Python reference doesn't contain a list of valid options, since these vary depending on the host system. Be aware that using options not in this list increase the chances of your program failing if it's run on operating systems other than yours.

The following Python program will give you a list of the socket options that your particular Python installation supports:

```
#!/usr/bin/env python
# Get list of available socket options -- Chapter 3 -- sockopts.py

import socket
solist = [x for x in dir(socket) if x.startswith('SO_')]
solist.sort()
for x in solist:
    print x
```

Binding the Socket

The next step is to claim a port number for the server. This process is called *binding*. You'll recall from Chapter 1 that each server requires its own port, and this port number is often well known.

To bind to a port, you'll issue a command such as this:

```
s.bind(('', 80))
```

That requests port 80, the standard HTTP (web) port. However, operating-system constraints generally restrict all port numbers below 1024 so that only the root user can bind to them. The examples in this book will all use higher port numbers.

The first argument to `bind()` specifies the IP address that you wish to bind. It's generally left blank, which means "bind to all interfaces and addresses."

Some machines have multiple network interfaces—for instance, a firewall might have an Ethernet card for the public Internet plus an Ethernet card for the internal network. In such a case, you might wish your server to only be visible to one interface, so you might supply the IP address of the internal interface to `bind()`. In this scenario, to clients connecting to the external interface, it appears as if there's no server on port 80 at all. In fact, you could run a separate server that was bound to port 80 on the external server!

It's also actually possible to bind a *client* socket to a particular IP address and port by calling `bind()`. However, this client-side capability is rarely exercised since the operating system will supply suitable values automatically.

If you want to use only a particular IP address, your call to `bind()` would look like this:

```
s.bind(('192.168.1.1', 80))
```

Listening for Connections

The last step before actually accepting client connections is to call `listen()`. This call tells the operating system to prepare to receive connections. It takes a single parameter, which indicates how many pending connections the operating system should allow to remain in queue before the server actually gets around to processing them. Many people set this to 5 as a matter of convention (some operating systems don't support values larger than that anyway). With modern multithreaded or multitasking servers, this parameter isn't of great significance but is still required. The `listen()` call looks like this:

```
s.listen(5)
```

Accepting Connections

Most servers are designed to run indefinitely (for months or even years) and service multiple connections. Clients, on the other hand, typically make few connections and run until their task is done or the user terminates them.

The usual way to make servers continually run is with a carefully designed infinite loop. Here's an example of a basic server:

```
#!/usr/bin/env python
# Base Server - Chapter 3 - basicserver.py
import socket

host = ''                                # Bind to all interfaces
port = 51423

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind((host, port))
print "Waiting for connections..."
s.listen(1)

while 1:
    clientsock, clientaddr = s.accept()
    print "Got connection from", clientsock.getpeername()
    clientsock.close()
```

The `while 1` means that this program will loop forever. In practical terms, that means it will loop until something raises an exception or terminates the program. In the case of this program, you can press Ctrl-C to terminate it, which generates a `KeyboardInterrupt` exception.

Interrupting on Windows

If you're running Windows, you may find that Ctrl-C doesn't cause a program to terminate. If this is the case, try pressing Ctrl-Break instead. The Break key is the same as the Pause key on most keyboards.

Normally, infinite loops are bad because they eat up your system's CPU resources. However, this loop is different; when you call `accept()`, it will not

return until a client has connected. In the meantime, your program is simply stalled and not using any CPU resources. A program that is stalled waiting for input or output to occur is said to be *blocked*.

To test this and several other server examples, you can use your operating system's telnet command. For this program, first start the server. Then you can run "telnet localhost 51423" from a different command prompt or Run dialog box. Your server process should report a connection and the telnet program should terminate immediately. Depending on your particular version of telnet, you may see a message such as "Connection reset by peer" or "Connection closed" or the program may simply exit without any output. That's normal for this server since it does nothing when the connection is established.

Handling Errors

Any uncaught exception will terminate your Python program. For a client, that's typically acceptable; it's reasonable for a client to exit when a problem occurs in many cases. For a server, this is very bad. It could mean that every time somebody presses Stop on a web browser, the entire server will go down and stop answering requests.

Therefore, you should always take care to handle errors. I suggest putting in a generic error handler to make sure nothing slips through the cracks. In the context of Python-based network programming, an error handler is simply a standard Python exception handler that processes network-related problems.

The following example attempts to catch all possible network errors and handle them in a way that will not terminate the server. Errors at the time of the call to `accept()` are printed but ignored; the `continue` statement returns to the top of the loop for the next `accept()` call. A Ctrl-C or Ctrl-Break is still allowed to terminate the program with the normal exceptions. Other errors are printed but ignored if possible. The following code illustrates this:

```
#!/usr/bin/env python
# Server With Error Handling - Chapter 3 - errorserver.py

import socket, traceback

host = ''                                # Bind to all interfaces
port = 51423

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind((host, port))
s.listen(1)
```

```

while 1:
    try:
        clientsock, clientaddr = s.accept()
    except KeyboardInterrupt:
        raise
    except:
        traceback.print_exc()
        continue

    # Process the connection

    try:
        print "Got connection from", clientsock.getpeername()
        # Process the request here
    except (KeyboardInterrupt, SystemExit):
        raise
    except:
        traceback.print_exc()

    # Close the connection

    try:
        clientsock.close()
    except KeyboardInterrupt:
        raise
    except:
        traceback.print_exc()

```

There are three separate `try` blocks in this program. The first surrounds the call to `accept()`, which can raise exceptions. The program will re-raise the `KeyboardInterrupt`, so if the person running the server presses Ctrl-C, it will continue to terminate as usual. All other exceptions get printed out, but the program doesn't terminate. Rather, it does a `continue`, which skips back to the top of the loop. Otherwise, the code would try to process the nonexistent client connection.

The second block surrounds the code that actually processes the connection. It passes on two exceptions: `KeyboardInterrupt` as before, and `SystemExit`. `SystemExit` is raised by calls to `sys.exit()`, and failing to pass it would lead to situations in which the program doesn't terminate like it should.

The third block surrounds the call to `close()`. This call isn't part of the second `try` block, because if it were there, an earlier exception would cause the call to `close()` and be skipped. This way, `close()` is always called when it should be. If you're using file objects, you should close them here as well.

Using try...finally Blocks to Close Sockets

In large programs, it may make sense to use Python's `try...finally` blocks to ensure that sockets are closed. Immediately after a successful call to `accept()`, you can insert `try` into the code. Before the final call to `close()`, use a `finally` statement to close the socket. In this case, any uncaught exceptions that occur between the `try` and `finally` statements will cause the socket to be closed and then the exception will be raised. Remember, though, that this exception will terminate the program, so you may still wish to handle it in some manner.

In Chapter 2, you saw how it's important for clients to call `shutdown()` to make sure they haven't missed discovering an error. For servers, this is often not necessary. If an error occurs, they typically disconnect the client, then ignore the problem and move on. Therefore, this example doesn't call `shutdown()`.

If you want to test this example, you'll probably find it difficult to actually cause an error that will result in a caught exception. That's OK, and it's how it should be. However, you can note that a Ctrl-C will still cause the `KeyboardInterrupt` exception to occur like before. (Ctrl-Break may not cause an exception.)

Using User Datagram Protocol

From a client perspective, using UDP is often more difficult than TCP, because clients must take care to deal with lost packets themselves. On the other hand, on the server side, UDP is rather easy. A programmer writing a UDP server generally has no need to worry about lost packets. After all, if the packet from a client never arrived, the UDP server has no way to know that a client ever tried to send a request. Since almost all UDP communication involves a client making a brief request and a server sending a brief answer, there's really nothing that a server can do to detect or handle a lost packet situation. The responsibility will lie with the client.

To use UDP on the server, you create a socket, set the options, and `bind()` just like with TCP. However, there's no need for `listen()` or `accept()`—just use `recvfrom()`. This function actually returns two pieces of information: the received data, and the address and port number of the program that sent the data. Because UDP is connectionless, this is all you need to be able to send back a reply. You don't need to have a specific socket connected to the remote as with TCP. Here's a simple UDP echo server that can be useful for testing your UDP clients.

```

#!/usr/bin/env python
# UDP Echo Server - Chapter 3 - udpechoserver.py
import socket, traceback

host = ''                                # Bind to all interfaces
port = 51423

s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind((host, port))

while 1:
    try:
        message, address = s.recvfrom(8192)
        print "Got data from", address
        # Echo it back
        s.sendto(message, address)
    except (KeyboardInterrupt, SystemExit):
        raise
    except:
        traceback.print_exc()

```

This program is significantly shorter than the TCP server with error detection. You can see that if your protocol involves a short request and a short response and nothing else, UDP may well be a better solution. It doesn't have all the overhead associated with establishing connections in TCP.

This UDP server simply starts a loop calling `recvfrom()`. The server can generate a reply and then use `sendto()` to send it back to the client using the endpoint address from `recvfrom()`. There's no need to close the client socket, because there's no separate socket to close. This is a good program to use to test the UDP echo client in Chapter 2.

Another example from Chapter 2 was a UDP network time client. It connected to a server at the National Institute of Standards and Technology to get the exact time. Let's write a server-side program for this protocol. But to add a twist, the program will send back yesterday's date instead of today's. Here's the server:

```

#!/usr/bin/env python
# UDP Wrong Time Server - Chapter 3 - udptimeserver.py
import socket, traceback, time, struct

host = ''                                # Bind to all interfaces
port = 51423

```

```

s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind((host, port))

while 1:
    try:
        message, address = s.recvfrom(8192)
        secs = int(time.time())           # Seconds since 1/1/1970
        secs -= 60 * 60 * 24              # Make it yesterday
        secs += 2208988800                # Convert to secs since 1/1/1900
        reply = struct.pack("!I", secs)
        s.sendto(reply, address)
    except (KeyboardInterrupt, SystemExit):
        raise
    except:
        traceback.print_exc()

```

This server is a simple, yet fully functional and complete, UDP server. You'll recall from Chapter 2 that the client sends an empty packet to the server, so this server just ignores the packet. It gets the current time from the system, calculates the return value, packs it up in binary, and sends the reply. If you modify the UDP time client from Chapter 2 (`udptime.py`) to connect to port 51423 on localhost, you'll find it returns yesterday's time.

Using `inetd` or `xinetd`

All the examples of servers in this chapter have something in common: They all start up a process on the server that waits for connections (or packets) and then handles them as they arrive. If you have many different servers on your machine, and they tend to be infrequently used, you'll wind up with a lot of memory being consumed by mostly idle processes.

The TCP examples in this chapter also share another trait: They can only service a single client at a time. In real production servers, this is rarely appropriate. There are several ways to solve this problem. One way is to use internal methods to handle many clients (see Chapters 20 to 22 for information on how to do that). Another way is to let something start up a new copy of the server for you each time a new client connects.

UNIX and UNIX-like systems provide a program called `inetd` or `xinetd` to solve these problems. The `inetd` or `xinetd` program opens, binds, listens, and accepts connections on each port of interest to a server. When a client connects, `inetd` knows which server the request is for (based on which server port number it arrived on). It then invokes the server and passes the socket to it.

xinetd or inetd?

Most UNIX vendors provide `inetd` with their systems. If you run any commercial UNIX or a BSD-derived UNIX, you'll likely have `inetd`. Some, but not all, Linux vendors such as Red Hat provide `xinetd` instead of `inetd`. The `xinetd` program works the same way as `inetd` and invokes server programs the same way. However, its configuration file takes on a different format. In this chapter, when you see "inetd," the comments will apply to both programs unless otherwise stated.

Microsoft doesn't bundle any `inetd` program with Windows, so this section doesn't pertain to Microsoft platforms unless you've installed an `inetd` program yourself. Programs like `inetd` aren't common on Windows platforms.

If you have no `inetd` program or lack administrator access to your system, Chapter 22 provides a bare-bones TCP-only `inetd` implementation (`inetd.py`) in Python. On UNIX or Linux systems, you can use it to run the TCP `inetd` examples in this chapter.

As it happens, `inetd` passes the socket in two ways: as file descriptors 0 and 1. These correspond to the file descriptors for standard input and standard output. So, the following could be run as an `inetd` server:

```
#!/usr/bin/env python
# Basic inetd server - Chapter 3 - inetdserver.py

import sys
print "Welcome."
print "Please enter a string:"
sys.stdout.flush()
line = sys.stdin.readline().strip()
print "You entered %d characters." % len(line)
```

This looks very much like a traditional Python program. In fact, the only thing you might not normally see is the call to `flush()` to ensure that the output is transmitted immediately. You can even run this program from the command line and interact with it directly—it is indeed a perfectly valid stand-alone program.

The call to `flush()` is required in this example because `sys.stdout` is buffered by default. In Chapter 2, you were able to tell `makefile()` to leave the file descriptor unbuffered, but that option isn't available here.

Configuring inetd

This section pertains to `inetd` only. If your system uses `xinetd` (it likely does if you have no `/etc/inetd.conf` but do have `/etc/xinetd.conf` or `/etc/xinetd.d`), skip to the “Configuring xinetd” section in this chapter.

The `inetd` program has its configuration file in `/etc/inetd.conf`. You’ll generally need to be logged in as root to modify `inetd.conf`.

Each server started by `inetd` contains its own line in `inetd.conf`, which takes this format:

```
port type protocol invocationtype username path programname arguments
```

Table 3-2 describes what these parameters do.

Table 3-2. Required Arguments

Parameter	Description
port	The port specifies the port number (or name as defined in <code>/etc/services</code>) on which the server should be listening. Examples: 80, http.
type	The type should be <code>stream</code> for a TCP server, or <code>dgram</code> for UDP.
protocol	protocol should be either <code>tcp</code> or <code>udp</code> as appropriate.
invocationtype	For TCP servers, the invocationtype should always be <code>nowait</code> . For UDP servers that <code>connect()</code> to the remote and thus require a new process for each packet from a different client, use <code>nowait</code> . If the UDP server handles all incoming packets on its port until it terminates, use <code>wait</code> . Please see “Using UDP with <code>inetd</code> ” later in this chapter for more details.
username	The username parameter specifies the user under which the server should be run.
path	path specifies the full path to the server.
programname	programname gives the name of the server program, as passed to it in <code>sys.argv[0]</code> .
arguments	The arguments are optional, and if present, appear as <code>sys.argv[1:]</code> in the server script.

To configure the example program, you might use this line in `inetd.conf` as follows:

```
51423 stream tcp nowait root /root/example.py /root/example.py
```

Before the changes can take effect, you must either restart `inetd` or send it the `HUP` signal to force it to reread its configuration file. On many systems, you can restart `inetd` by running `/etc/init.d/inetd restart`. If that doesn't work for you, you'll need to find its process ID and send it the `HUP` signal. Here's an example:

```
# ps ax | grep inetd
13628 ? S 0:00 /usr/sbin/inetd
14527 pts/4 R 0:00 grep inetd
# kill -HUP 13628
```

Configuring xinetd

This section pertains to `xinetd` only. If your system uses `inetd`, see the "Configuring `inetd`" section earlier in this chapter.

Depending on your vendor, your `xinetd` configuration will happen in either `/etc/xinetd.d` or `/etc/xinetd.conf`. If you have a `/etc/xinetd.d` directory, your configuration happens there; otherwise, it's in `/etc/xinetd.conf`. The syntax is the same either way. If you use `xinetd.conf`, adding a new server means inserting a server block into that file. If you use `xinetd.d`, adding a new server means creating a new file (you can make up a name) in that directory and adding the server block there. The block itself looks the same either way. You'll generally need to be logged in as root to modify any of these files.

Each server started by `xinetd` contains its own block. The `xinetd` program supports many different options. I'll present the set of options that cause `xinetd` to behave in the same manner as `inetd` so that the examples in this chapter work with both environments.

An `xinetd` block begins with "`service servicename`", where the `servicename` is the port name or number for the server. Following the service declaration is a set of definitions in braces that define attributes for the server. Each definition has a definition name, an equal sign, and the definition attributes. Table 3-3 lists the different options that you'll use.

Table 3-3. Basic xinetd Service Options

Option Name	Description
flags	Various xinetd-specific flags governing the operation of this server. For these examples, you'll specify only NAMEINARGS, which causes the arguments to be passed in the same manner as inetd.
type	If you're defining a service that isn't listed in /etc/services, you should set UNLISTED here. Otherwise, you can omit the type line.
port	If you're defining a type = UNLISTED server, you must specify its port number here.
socket_type	Can be set to stream for TCP servers or dgram for UDP servers.
protocol	Set to tcp for TCP servers or udp for UDP servers.
wait	Set to no for all TCP servers. UDP servers that connect() to a remote and thus require a new process for each packet from a different client should also say no. If the UDP server handles all incoming packets on its port until it terminates, set this to yes. Please see "Using UDP with inetd" later in this chapter for more details.
user	The name of the user under which the program should be run.
server	The full path to the actual program that implements the server.
server_args	The set of arguments to pass to the server. Because you're using the NAMEINARGS flag for inetd compatibility, you must always specify at least one argument—the server name. Other arguments may be specified after the name.

To configure your example program, you might use the following block in /etc/xinetd.conf or in a /etc/xinetd.d file.

```
service pythontestserver
{
    flags      = NAMEINARGS
    type       = UNLISTED
    port        = 51423
    socket_type = stream
    protocol   = tcp
    wait        = no
    user        = root
    server      = /root/example.py
    server_args = /root/example.py
}
```

Before the changes can take effect, you must, logged in as root, either restart xinetd or send it the HUP signal to force it to reread its configuration file.

On many systems, you can restart xinetd by running /etc/init.d/xinetd restart. If that doesn't work for you, you'll need to find its process ID and send it the HUP signal. Here's an example:

```
# ps ax | grep xinetd
13628 ?      S      0:00 /usr/sbin/xinetd
14527 pts/4   R      0:00 grep xinetd
# kill -HUP 13628
```

Running the Example

Now that inedt or xinetd has been configured, you should be able to use the previous example by connecting to port 51423. Here's an example session:

```
$ telnet localhost 51423
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Welcome.
Please enter a string:
Test
You entered 4 characters.
Connection closed by foreign host.
```

Using Socket Objects with inetd

Sometimes, you would like to be able to have access to the socket-specific features you're accustomed to in Python. Normally, socket objects are created by calling `socket.socket()`. When your server is started from `inetd`, though, you don't want to create a new socket. Instead, you call `socket.fromfd()` to create a socket object based on the file descriptor that `inetd` passed to your program. The `fromfd()` call takes a file number and the standard arguments you're familiar with when you call `socket()`. It returns a new socket object. Here's an example:

```
#!/usr/bin/env python
# Socket-based inetd server - Chapter 3 - inetdsocket.py

import sys, socket, time
s = socket.fromfd(sys.stdin.fileno(), socket.AF_INET, socket.SOCK_STREAM)
s.sendall("Welcome.\n")
s.sendall("According to our records, you are connected from %s.\n" % \
          str(s.getpeername()))
s.sendall("The local time is %s.\n" % time.asctime())
```

In this example, there's no equivalent to `getpeername()` without using the socket object itself. So, the socket object is created with `fromfd()`. After that, you can use it just like any other socket object.

Using UDP with inetd

The one process per connection model used with `inetd` is straightforward when applied to TCP connections—each time a new client connects, a new server is started, and continues running until the client disconnects. UDP, however, has no notion of a connection. Therefore, `inetd` has to work with UDP servers to figure out how to best manage UDP communication. There are two different approaches: `wait` and `nowait`.

A `wait` server, once started, *typically* will handle all incoming UDP requests on its port from all clients—just like the earlier UDP server examples in this chapter. While the server is running, `inetd` does nothing with its port. When the server exits, `inetd` resumes listening, and will restart the server when a new client connects. Such a server normally has a timeout—perhaps 60 seconds—after which it will terminate. This type of server will be covered in Chapter 5.

Some `wait` servers will receive one packet, then fork off a child to handle it, and have the parent terminate immediately. This is a workaround with some

inherent limitations in the nowait implementation in `inetd`, and will be demonstrated later in this chapter. It's used to simulate a nowait server with a wait server.

A nowait server will begin by using `recvfrom()` to receive a single UDP packet. It will then use `connect()` so that packets from other clients don't appear on the nowait server. The server will be left to its own devices to determine when it's time to terminate—perhaps after it's done sending its response, or perhaps it will also use a timeout. While a nowait server is running, `inetd` will continue listening for packets from other clients, and when they arrive, it will start a second invocation of your server to handle the new client. Here's an example of such a server:

```
#!/usr/bin/env python
# UDP Inetd Server - Chapter 3 - inetdudpserver.py
import socket, time, sys

s = socket.fromfd(sys.stdin.fileno(), socket.AF_INET, socket.SOCK_DGRAM)
message, address = s.recvfrom(8192)
s.connect(address)
for i in range(10):
    s.send("Reply %d: %s" % (i + 1, message))
    time.sleep(2)
s.send("OK, I'm done sending replies.\n")
```

You can configure it in `inetd.conf` with the following:

```
51423 dgram udp nowait root /home/user/inetdudpserver.py →
/home/user/inetdudpserver.py
```

Or, for `xinetd`, use the following:

```
service pythonnowaitexample
{
    flags = NAMEINARGS
    type = UNLISTED
    port = 51423
    socket_type = dgram
    protocol = udp
    wait = no
    user = root
    server = /home/user/inetdudpserver.py
    server_args = /home/user/inetdudpserver.py
}
```

When you run the previous example (use the UDP echo client `udp.py` from Chapter 2), you'll see it send back ten responses. However, if you use `ps ax` to check to see what processes are running, you'll likely see many copies of this single server! The reason is that there's a race condition: as soon as `inetd` starts up the UDP server, it goes back to checking for incoming connections. However, until your code runs `recvfrom()`, `inetd` still sees the single incoming connection in its queue. Even though it may take only milliseconds for your code to get there, in that time `inetd` could see that single connection a dozen times, and start a dozen copies of your server to handle it! This problem is worse with Python, because Python programs typically take longer to start up than C programs do. Although this time is still generally under one-tenth of a second, it's nevertheless an important difference.

Therefore, I recommend that you *not* use the `nowait` method in your code. However, you can achieve the same effect by combining a `wait` entry in `inetd.conf` (or `wait = yes` in `xinetd.conf`) with a specially designed server like this one:

```
#!/usr/bin/env python
# UDP Inetd Server for Wait - Chapter 3 - inetdwaitserver.py
import socket, time, sys, os

s = socket.fromfd(sys.stdin.fileno(), socket.AF_INET, socket.SOCK_DGRAM)
message, address = s.recvfrom(8192)
localaddr = s.getsockname()
s.close()

pid = os.fork()
if pid:
    sys.exit(0)

s2 = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s2.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s2.bind(localaddr)
s2.connect(address)

for i in range(10):
    s2.send("Reply %d: %s" % (i + 1, message))#
    time.sleep(2)
s2.send("OK, I'm done sending replies.\n")
```

Configure it like this in `inetd.conf`:

```
51423 dgram udp wait root /home/user/inetdwaitserver.py ↵
/home/user/inetdwaitserver.py
```

Or, for `xinetd`, do the following:

```
service pythonnowaitexample
{
    flags = NAMEINARGS
    type = UNLISTED
    port = 51423
    socket_type = dgram
    protocol = udp
    wait = yes
    user = root
    server = /home/user/inetdwaitserver.py
    server_args = /home/user/inetdwaitserver.py
}
```

Since this is a wait service, it can't wait the full 20 seconds to exit. So, after reading the message, it saves off the local address and forks. The `fork()` call creates a second process running the same program as the original (for more details, refer to Chapter 20). The original process then terminates so that `inetd` can continue listening for more incoming connections.

The new process, however, creates a new socket object. It binds it to the *same address* as the server socket `inetd` is using, then calls `connect()` to tell it that it will only communicate with a specific remote address. This socket is allowed to bind to the same address as the server because it's not calling `listen()` but is using `connect()`, so it effectively looks like the same server socket to the client, but on the server side will only communicate with the single client.

If you try this program with your `inetd` server, you'll find it works as expected. Watching the output of `ps ax` will show a copy of it running for 20 seconds, and then it will die.

Handling Errors with `inetd`

Earlier in this chapter, you learned how important it is to detect and properly handle all errors with conventional servers. With `inetd`-based servers, there's no such need.

Because each `inetd` server process handles exactly one client, it's not a huge problem if the server process terminates because of an error. The single client

will lose its connection, but the next time a connection is made, `inetd` will start a new server process to handle it.

That doesn't mean that this is never a problem. Some `inetd` implementations will point `stderr` at the client, so if an uncaught exception does occur, it will actually be sent over the network to the client. This can seriously confuse clients expecting other types of data. Additionally, the server administrator will never learn of the problem. If these will be problems in your case, you should catch errors and log them somewhere an administrator can see. See the section describing logging later in this chapter for more details.

When Not to Use `inetd`

Although there are advantages to using `inetd` in many cases—and it can simplify your server design—there are also disadvantages, which can be serious. The primary one is the overhead of starting a new process and invoking the server program for each connection.

This overhead is serious even for C, and servers that handle many short-lived connections aren't suitable for `inetd`. That's why you rarely see web servers started from `inetd`—they handle such a high volume of connections that the `inetd` overhead could impose a serious load on the server. Chapters 20 through 22 introduce alternatives for your programs.

Another problem is control. If your server needs to decide for some reason to stop listening for connections for a while, it has no way to communicate this to `inetd`. While this is rare, it can still be an issue for some people.

Finally, if your program is going to be run on non-UNIX platforms, `inetd` isn't likely to be available to you, so you'll have to rely on other options.

Logging with `syslog`

An important part of communicating status to a server administrator is making log files. UNIX and UNIX-like systems provide a facility called `syslog`, which helps you do this, and Python provides a convenient interface to it.

NOTE The `syslog` facility will not work on Windows. If you need to log on Windows, you may wish to consider the `logging` module and the `NTEventLogHandler()`. However, that logging handler requires extensions that aren't a standard part of Python. You may, therefore, not be able to use it unless you have installed Python and the extension yourself. The best way to achieve portable logging may be to simply write events to a file.

The syslog facility is a configurable generic infrastructure for logging. That means that it's designed to present a unified interface to any application that wishes to make use of logging. It also means that the system administrator can configure how the logging is done: that is, into what files, and can expect those settings to apply to all the applications on the system. Many UNIX-like systems ship with syslog enabled by default and place the log files in /var/log. The syslog system also permits logging across the network in some situations. This allows servers that have no disk for logging, or that wish to have a secure copy of their logs on another machine, to send logs to a remote syslog for storage.

Entries in a syslog file are typically automatically stamped with a date, time, hostname, and program name. Here is an example entry from a log file:

```
Apr 14 06:45:32 erwin postfix/qmgr[24445]: DF67314E3: from=<root@complete.org>,  
size=711, nrcpt=1 (queue active)
```

This line (it was one long line in the log file) shows some activity from the mail server process postfix/qmgr on the machine erwin at 6:45 a.m. on April 14.

syslog vs. Logging

Python 2.3 introduced a new module named `logging` that provides a generic interface to several different logging methods. If you're running on non-UNIX platforms and have the latest versions of Python available, along with the relevant extensions for your particular platform, you should consider it. Unfortunately, some systems don't yet feature Python 2.3, and the `logging` module doesn't yet provide the same level of flexibility for `syslog` as the specific `syslog` module does.

Using syslog in Python

Python provides a `syslog` module that interfaces with the system's `syslog` program. Before you can log any messages, you must initialize the `syslog` interface by calling `openlog()`. Python defines it like this:

```
openlog(ident[, logopt[, facility]])
```

The first parameter, `ident`, is an identification string that will be automatically added to every log message. It's typically the program name, sometimes with its process ID added as well. In the previous `syslog` line, the `ident` was `postfix/qmgr[24445]`.

The available log options may vary from system to system. They're integers that can be combined with the Python bitwise or operator. The options that may be available include the following ones listed in Table 3-4.

Table 3-4. Syslog Options

Option Name	Description
<code>LOG_CONS</code>	If and only if the machine's <code>syslog</code> process is unreachable or encounters an error logging the message, you should display the message directly on the system's primary physical console.
<code>LOG_NDELAY</code>	Do not delay opening the connection to the <code>syslog</code> program (normal behavior causes the connection to be opened with the first logged message).
<code>LOG_NOWAIT</code>	On systems that create a new process to log a message, do not <code>wait()</code> for the process. Some systems don't create a new process, and on those systems, this option has no effect.
<code>LOG_PID</code>	Automatically include the process ID with each log message.
<code>LOG_PERROR</code>	Print messages on <code>stderr</code> in addition to sending them via normal <code>syslog</code> mechanisms (this option isn't as portable as the others).

The optional `facility` argument is used to identify the type of program that's generating the messages. This is generally used by the system administrator to configure how messages are split out into various files and serves no other purpose. Python supports the following facilities. Their generally understood meaning is documented in Table 3-5, but may vary depending on your system configuration or operating system defaults.

Table 3-5. syslog Facilities

Facility Name	Description
LOG_AUTH	Authentication messages: logging in, logging out.
LOG_CRON	Messages from the automated command scheduler.
LOG_DAEMON	Any system server that doesn't fall under a different logging category.
LOG_KERN	Operating-system kernel messages (should rarely be used in Python programs).
LOG_LOCALx	LOG_LOCAL0 through LOG_LOCAL7 are defined for "local" use and customized by each system administrator. Your application should only use these facilities if the application is designed for internal use only, because the meanings of the LOG_LOCALx facilities will likely differ at other sites.
LOG_LPR	Print server messages.
LOG_MAIL	Mail-related messages.
LOG_NEWS	Usenet news messages.
LOG_USER	Generic nonspecific user-defined messages. This is the default facility if None is specified.
LOG_UUCP	UNIX-to-UNIX Copy Protocol (UUCP) messages (rarely seen today).

Now that the logging system is initialized, if you want to actually log a message, you call the `syslog()` function. Python defines it like this:

```
syslog([priority,] message)
```

The message is simply a string to be logged. The priority describes how important a given message is. It's used by the `syslog` configuration to determine what to do with a given message. For instance, high-priority messages may be displayed on the system's console or may cause an operator to be paged, though low-priority messages may simply be discarded. The default priority is `LOG_INFO`. Table 3-6 lists the available priorities along with their generally accepted meanings, with the highest-priority items listed first.

Table 3-6. syslog Priorities

Priority Name	Description
LOG_EMERG	Emergency; the entire system is down or unusable.
LOG_ALERT	Alert the administrators; immediate action is required.
LOG_CRIT	A critical error has occurred.
LOG_ERR	A general error has occurred.
LOG_WARNING	A warning is being logged (often used for things that don't constitute an error but may merit administrator actions).
LOG_NOTICE	Notice of an important normal situation.
LOG_INFO	Informational message.
LOG_DEBUG	Debugging message only; often discarded.

Here's an example program that demonstrates the use of `syslog`:

```
#!/usr/bin/env python
# Syslog example - Chapter 3 - syslogsample.py

import syslog, sys, StringIO, traceback, os

def logexception(includetraceback = 0):
    exctype, exception, exctraceback = sys.exc_info()
    excclass = str(exception.__class__)
    message = str(exception)

    if not includetraceback:
        syslog.syslog(syslog.LOG_ERR, "%s: %s" % (excclass, message))
    else:
        excfd = StringIO.StringIO()
        traceback.print_exception(exctype, exception, exctraceback, None,
                                  excfd)
        for line in excfd.getvalue().split("\n"):
            syslog.syslog(syslog.LOG_ERR, line)

def init syslog():
    syslog.openlog("%s[%d]" % (os.path.basename(sys.argv[0]), os.getpid()),
                  0, syslog.LOG_DAEMON)
    syslog.syslog("Started.")
```

```

initsyslog()

try:
    raise RuntimeError, "Exception 1"
except:
    logexception(0)

try:
    raise RuntimeError, "Exception 2"
except:
    logexception(1)

syslog.syslog("I'm terminating.")

```

This program first calls `initsyslog()`, which initializes the `syslog` system. It sets the program's name to be the name it was called with plus its process ID, and configures the messages to go into the daemon log file, which is typical for a server. It then logs a message saying the server has started.

The server then raises and catches two exceptions: The first is logged without a traceback, and the second is logged with a full traceback.

The `logexception()` function is responsible for logging exceptions. It first gathers information about the exception by calling `sys.exc_info()`. If tracebacks aren't enabled, it simply logs the exception type and its message. Otherwise, it generates the traceback and logs it.

You may find it useful to wrap your entire server program in a `try...except` clause. You have the `except` clause call `logexception(1)` followed by `sys.exit(1)` to log the exception and then terminate with an error.

If you run this example, you won't see anything at your console. However, some data will be logged to your system's log files. If you check your `/var/log/messages` or `/var/log/syslog` files, you'll see information logged there from `syslogsample.py`. It will even log an exception traceback from the `RuntimeError` exceptions that were raised.

Avoiding Deadlock

One common problem plaguing server designers is *deadlock*. Deadlock occurs when a server and client are both trying to write to a connection simultaneously, and when they're both trying to read from the connection simultaneously. In these situations, neither process will ever get any data back (if they're both reading). Consequently, if they're writing, outgoing buffers will fill. The result will be that they just get stuck doing nothing.

To illustrate the problem, you'll see a basic TCP echo server and a custom TCP echo client. Here's the server:

```
#!/usr/bin/env python
# Echo Server - Chapter 3 - echoserver.py
import socket, traceback

host = ''                                # Bind to all interfaces
port = 51423

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind((host, port))
s.listen(1)

while 1:
    try:
        clientsock, clientaddr = s.accept()
    except KeyboardInterrupt:
        raise
    except:
        traceback.print_exc()
        continue

    # Process the connection

    try:
        print "Got connection from", clientsock.getpeername()
        while 1:
            data = clientsock.recv(4096)
            if not len(data):
                break
            clientsock.sendall(data)
    except (KeyboardInterrupt, SystemExit):
        raise
    except:
        traceback.print_exc()

    # Close the connection
```

```

try:
    clientsock.close()
except KeyboardInterrupt:
    raise
except:
    traceback.print_exc()

```

This is a basic echo server. If you fire it up and telnet to port 51423 on the local server, you'll see it echo back everything you type. Now here's a TCP echo client:

```

#!/usr/bin/env python
# Echo client with deadlock - Chapter 3 - echoclient.py

import socket, sys
port = 51423
host = 'localhost'

data = "x" * 10485760          # 10MB of data

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((host, port))

byteswritten = 0
while byteswritten < len(data):
    startpos = byteswritten
    endpos = min(byteswritten + 1024, len(data))
    byteswritten += s.send(data[startpos:endpos])
    sys.stdout.write("Wrote %d bytes\r" % byteswritten)
    sys.stdout.flush()

s.shutdown(1)

print "All data sent."
while 1:
    buf = s.recv(1024)
    if not len(buf):
        break
    sys.stdout.write(buf)

```

To see the problem, start up the echo server, and then the echo client. The server will show the connection from the client. The client will attempt to transmit 10 MB of data to the server in 1 KB chunks, then it will read back the result. While it's sending data, it will update you on its status.

It will never get all 10 MB sent, though. The server will read the first 4 KB, try to write it, and repeat the process. Because the client doesn't do any reading at all until all the data has been sent, the operating system's transmission buffers at some point will fill and both processes will be stuck in the `send()`. On my system, the client reported that it had sent approximately 350 KB before getting stuck. That number will likely vary from system to system, and possibly even from attempt to attempt.

There are several approaches you could take to fix this problem. The most direct would be to make sure that the client does a `recv()` after each `send()`. Another approach would be to simply have the client send less data (if you modify the value 10485760 to 1024, you'll see no problem at all.) A third approach is to use multitasking or some other method to be able to send and receive simultaneously on the client side. This option is covered in chapters 20 through 22.

The problem can often be trickier to detect than in this example. As a server designer, you have to be aware that misbehaving clients may sometimes connect, and you should always be prepared for deadlock and leave yourself a way out. This is generally done by using a timeout. For more information on timeouts, see Chapter 5.

Summary

Servers typically wait for requests from clients and send responses back. Like client programming, servers use the socket interface, but setting up the socket is a more involved four-step process.

Socket options can be used to alter the behavior of the networking system for a particular connection. The most frequently used option for a server, `SO_REUSEADDR`, will allow its port to be reused immediately after the socket is closed.

Both TCP and UDP servers are possible. TCP servers will normally use `accept()` to create a new socket for each client that connects. UDP servers typically use only a single socket and rely on the return value from `recvfrom()` to know where to send a response.

The `inetd` or `xinetd` programs provide a convenient way to listen for connections and hand them off to server processes. Both programs serve the same basic purpose, though they're configured differently. Servers that work with `inetd` or `xinetd` will have their standard input and standard output set to the socket by default.

Because server programs often aren't run interactively, you need to find a method of communicating information to the operators. UNIX systems provide the syslog interface and Python has a module for it.

Deadlock can occur when both the server and client are stuck waiting for an action to occur. Careful protocol design and the use of timeouts can help minimize the frequency and impact of deadlock conditions.

CHAPTER 4

Domain Name System

THE DOMAIN NAME SYSTEM (DNS) is a distributed database that's typically used to resolve hostnames into IP addresses. DNS and related systems exist for two main reasons:

- They make it easier for humans to remember names such as www.apress.com than IP addresses such as 65.215.221.149.
- They allow servers to change IP addresses and still be reached using the same name.

In this chapter, you'll learn about two ways of accessing the DNS: using the operating system's built-in support via Python's `socket` module, and querying DNS directly using PyDNS from <http://pydns.sf.net/>.

Python DNS Libraries

There are several different DNS modules available for Python. PyDNS, covered in this chapter, is one of them. Another option is `dnspython`, available from www.dnspython.org. You can obtain `dnslook` from <http://pilcrow.madison.wi.us/>. Finally, there is `adns`, an asynchronous DNS library from <http://dustman.net/andy/python/adns-python>.

Making DNS Queries

DNS is a massive, globally distributed database. It provides a series of referrals, each giving a more specific answer, until the final answer is obtained.

As an example, let's consider the case for looking up `www.external.example.com`. First, your program will communicate with the local name server designated by your operating system's configuration. This server is a *recursive* name server; it receives the request and passes it on as appropriate. It will do much of the work for you.

The first thing the recursive server does is to ask about the `.com` domain. It will have a built-in list of the top-levels—those servers that can hand out information about the world's top-level names such as `.com`.

The answer for `.com` will come in the form of a referral to another name server—one that can provide information about names in `.com`. So, a query will be sent to that server. The `.com` server will respond with yet another referral—this one to a server that can provide information about names in `example.com`.

The cycle repeats itself yet again until finally the query is sent to the name server for `external.example.com`. This server will know the IP address in question and return it.

The operating system provides services for performing basic DNS lookups. These services are sufficient for most programs, and spare you the effort of having to deal with all the mechanics of making DNS lookups directly. Python provides an interface to these basic operating-system services in its `socket` module. Third-party modules exist to provide more advanced functionality as well.

Using Operating System Lookup Services

The operating system comes with several functions for DNS lookups (often called *resolver libraries*), which provide everything needed for most applications. Some programs, most notably mail servers and DNS query utilities, will require more sophisticated tools.

When you use the operating system lookup services, the system typically is using a little bit more than pure DNS to answer your queries. Many UNIX systems contain a file called `/etc/hosts` that defines hostnames and IP addresses. Operating-system queries will typically check `/etc/hosts` first, and then resort to DNS if no answer is found. Windows machines don't typically use a `hosts` file.

Also, every operating system will have to provide a way for the administrator to specify the IP addresses of the name servers (DNS servers) that should be used to resolve queries. On UNIX systems that's done in `/etc/resolv.conf`, while Windows systems provide these settings in the registry entries maintained by changing the TCP/IP settings in the Control Panel's *Network* applet. Either way, information is provided about the default name servers as well as a default domain name. For example, if you provide a default domain name of "example.com" and attempt to look up "www", the system will automatically try "www.example.com" for you. All of these service and configuration details are hidden from you; the operating system automatically uses the information when formulating answers to your queries.

Performing Basic Lookups

The most basic lookup is a forward lookup seeking an IP address from a hostname. For instance, if you wish to download a web page from `www.example.com`, you'll need to find its IP address first. A forward lookup will do that for you; it will translate a name into an IP address.

In some cases, Python's socket library will perform the lookup for you. However, you might want to do it yourself. Querying DNS does incur some overhead, so your program will perform better if you keep DNS queries to a minimum. This might be the case if you'll be connecting to the same server several times. In Python, the function you'll use to do this is `socket.getaddrinfo()`. Python defines `socket.getaddrinfo()` like this:

```
getaddrinfo(host, port[, family[, socktype[, proto[, flags]]]])
```

Missing `gethostbyname()`

C programmers will be familiar with the `gethostbyname()` function. Python does provide an analogous `socket.gethostbyname()` function, but this function isn't compatible with IPv6. All code presented in this section will work with both IPv4 and IPv6 to ensure maximum future compatibility. Please see Chapter 5 for more information on IPv6.

The `host` argument is the name you would like to resolve. The remaining arguments are generally useful only if you'll be passing the result directly to `socket.socket()` or `socket.connect()`; they restrict what protocols are displayed in the output and fill in the results with default values for creating sockets. For now, you'll specify `None` for `port` and omit the remaining arguments to do a basic lookup. Python defines the return value of `socket.getaddrinfo()` as a list of tuples. Each tuple looks like this:

```
(family, socktype, proto, canonname, sockaddr)
```

The `sockaddr` is the actual address for the remote and is often the piece of data you'll be looking for when performing queries. Since Python counts tuple elements starting with zero, it's element four in the result. The `socket.getaddrinfo()` function returns a list of tuples because there may be more than one answer for a given query. For instance, if `www.example.com` has several different web servers with each serving the same content, it may return several different IP addresses. You can use any one of them successfully. The `www.example.com` site might do this because it's popular and needs multiple servers to handle the load. Alternatively, `www.example.com` might be reachable by both IPv4 and IPv6. On an IPv6-enabled system, you may get results for both methods of reaching it.

If all you need is a simple IP address to use for connecting, you can just choose the first tuple from the list. Here's an example:

```
#!/usr/bin/env python
# Basic getaddrinfo() basic example - Chapter 4 - getaddrinfo-basic.py

import sys, socket

result = socket.getaddrinfo(sys.argv[1], None)
print result[0][4]
```

You'll recall that the sockaddr data in the tuple is the one of interest. Therefore, this code will obtain the sockaddr result from the first tuple in the result list. Try running it with a few examples as follows:

```
$ ./getaddrinfo-basic.py www.example.com
('192.0.34.166', 0)
$ ./getaddrinfo-basic.py www.yahoo.com
('216.109.118.71', 0)
$ ./getaddrinfo-basic.py www.yahoo.com
('216.109.118.64', 0)
```

Notice that the value printed is a tuple. This is because you can pass a specific port (by name or number) to `getaddrinfo()` and it will then return a ready-to-use address and port tuple for something like `connect()`.

Notice also that the two calls seeking information about `www.yahoo.com` returned different IP addresses. That's perfectly legitimate. It turns out that there are many IP addresses defined for `www.yahoo.com` as shown here:

```
$ host www.yahoo.com
www.yahoo.com          CNAME   www.yahoo.akadns.net
www.yahoo.akadns.net    A       216.109.118.69
www.yahoo.akadns.net    A       216.109.118.74
www.yahoo.akadns.net    A       216.109.118.76
www.yahoo.akadns.net    A       216.109.118.77
www.yahoo.akadns.net    A       216.109.118.78
www.yahoo.akadns.net    A       216.109.118.64
www.yahoo.akadns.net    A       216.109.118.65
www.yahoo.akadns.net    A       216.109.118.68
```

NOTE The `host` command is typically available only on UNIX platforms. Windows and some Linux distributions may not distribute it by default, though Linux users can obtain it from the `bind` package.

It's possible to get all of these entries from `getaddrinfo()`. You might first try this:

```
=!/usr/bin/env python
= Basic getaddrinfo() not quite right list example - Chapter 4
= getaddrinfo-list-broken.py
= Takes a hostname on the command line and prints all resulting
= matches for it. Broken; a given name may occur multiple times.

import sys, socket

= Put the list of results into the "result" variable.
result = socket.getaddrinfo(sys.argv[1], None)

counter = 0
for item in result:
    # Print out the address tuple for each item
    print "%-2d: %s" % (counter, item[4])
    counter += 1
```

However, when you run this program, you'll see each entry appearing several times:

```
$ ./getaddrinfo-list-broken.py www.yahoo.com
0 : ('216.109.118.74', 0)
1 : ('216.109.118.74', 0)
2 : ('216.109.118.74', 0)
3 : ('216.109.118.76', 0)
4 : ('216.109.118.76', 0)
5 : ('216.109.118.76', 0)
...
```

The reason is that `getaddrinfo()` is generating a result for every different protocol type it supports.

NOTE It's possible that you may not see each entry listed several times. On some platforms, such as Windows, `getaddrinfo()` only supports one protocol type by default. Even on platforms that support multiple protocol types, only one result may be given if some of the protocol types aren't configured on the machine running the example. Nevertheless, the principle is important, since these systems may support additional protocol types in the future and other systems may support them now. To restrict the results so that each item is listed only once, you'll specify `socket.SOCK_STREAM` for the protocol and zero for the family to continue allowing it to support all families.

Here's a better version of the last example:

```
#!/usr/bin/env python
# Basic getaddrinfo() list example - Chapter 4 - getaddrinfo-list.py

import sys, socket

# Obtain results for socket.SOCK_STREAM (TCP) only, and put a list
# of them into the "result" variable.
result = socket.getaddrinfo(sys.argv[1], None, 0, socket.SOCK_STREAM)

counter = 0
for item in result:
    # Print out the address tuple for each item
    print "%-2d: %s" % (counter, item[4])
    counter += 1
```

This time, you'll see each item appear only once, as shown here:

```
$ ./getaddrinfo-list.py www.yahoo.com
0 : ('216.109.118.78', 0)
1 : ('216.109.118.65', 0)
2 : ('216.109.118.66', 0)
3 : ('216.109.118.68', 0)
4 : ('216.109.118.69', 0)
5 : ('216.109.118.70', 0)
6 : ('216.109.118.71', 0)
7 : ('216.109.118.77', 0)
```

The `socket.SOCK_STREAM` doesn't make a practical difference on name resolution; it's merely used as part of the full result from `getaddrinfo()` that can be used for creating and connecting sockets. See the IPv6 information in Chapter 5 for more information on using this return value for socket creation.

Performing Reverse Lookups

There are times when you have an IP address and need to determine the hostname for that IP address. Most often, this need occurs in servers that want to know more about the client that's connecting to them, but it may occasionally occur elsewhere as well.

As an example, suppose you're writing a server for an online banking website. Security is going to be critical for this application, and detailed logs may be important so you have as much information as possible should a problem occur.

Although you could just log the remote IP address from the socket itself, you may find it useful to also log the hostname for that IP address. This would let you look for patterns in the log file—for instance, a suspicious number of connection requests from a particular Internet provider at 3 a.m. It could also be useful for tracking down security breaches; you would have an additional clue as to the person's company or provider right away. Of course, this isn't a foolproof method, but it's one way that reverse lookups can be useful.

Reverse Lookup Basics

Before any code is presented, it's critical for you to understand that it's perfectly valid for no reverse mapping to exist for a given IP address. In fact, many don't have such a mapping at all. Internet standards have reverse DNS, like DNS itself, as an optional feature. Therefore, you should make sure to trap and handle `socket.error()` for each attempt to perform a reverse lookup. The following is an example of performing a reverse lookup in Python. It will take a command-line parameter giving an IP address and display the hostname for that address, as shown here:

```
=!/usr/bin/env python
# Basic gethostbyaddr() example - Chapter 4 - gethostbyaddr-basic.py
# This program performs a reverse lookup on the IP address given
# on the command line

import sys, socket

try:
    # Perform the lookup
    result = socket.gethostbyaddr(sys.argv[1])

    # Display the looked-up hostname
    print "Primary hostname:"
    print "  " + result[0]

    # Display the list of available addresses that is also returned
    print "\nAddresses:"
    for item in result[2]:
        print "  " + item

except socket.error, e:
    print "Couldn't look up name:", e
```

Try running this with a few examples:

```
$ ./gethostbyaddr-basic.py 127.0.0.1
Primary hostname:
localhost

Addresses:
127.0.0.1

$ ./gethostbyaddr-basic.py 127.0.0.2
Couldn't look up name: (1, 'Unknown host')
$ ./gethostbyaddr-basic.py 2001:6b0:1:ea:a00:20ff:fe8f:708f
Primary hostname:
renskav.stacken.kth.se

Addresses:
2001:6b0:1:ea:a00:20ff:fe8f:708f

$ ./gethostbyaddr-basic.py 216.109.118.73
Primary hostname:
p10.www.dcn.yahoo.com

Addresses:
216.109.118.73
```

The first example shows an example looking up the localhost address, which should work for everyone. The second example shows the error handling from an address that has no reverse mapping defined. Note that the exact error message received may differ from one system to the next. The third example demonstrates lookup of IPv6 addresses (which will only work if you're running on an IPv6-enabled machine). The fourth example demonstrates lookup of a public IPv4 address.

Sanity Checks on Reverse-Lookup Data

Occasionally, you'll find that attackers may insert bogus reverse-lookup records. For instance, somebody might insert a reverse-lookup record claiming that an IP address is part of, say, `whitehouse.gov`. They may be doing this to attempt to bypass security restrictions (perhaps a site only allows connections from machines whose IP address maps to something under `whitehouse.gov`) or to fool someone into thinking she's from somewhere other than their true location.

This deception is possible due to the way delegation of DNS information happens. For normal, forward queries, the White House is responsible for publishing information about `whitehouse.gov`. When you ask the top-level `.gov` name server where to go for information about `whitehouse.gov`, it will point you to a name server run by the White House.

For reverse lookups, delegation happens based on IP address. For example, if you were performing a reverse lookup on the IP address 192.168.1.2, you might find the name server responsible for handling all reverse lookups starting with 192.168.1. In its entry for 192.168.1.2, it can specify anything it wishes—including whitehouse.gov—even if that entry is misleading. The result is obtained by valid means but generates an invalid answer.

Nothing in the DNS infrastructure can prevent this sort of deception; however, you can add some intelligence to your code to help prevent it. To do that, you first perform a reverse lookup like usual. That will give you a hostname from the IP address. Next, you'll want to perform a forward lookup on that hostname. If all is well, the original IP address should be listed in the forward lookup. Otherwise, someone may be providing fake reverse lookup information.

Here's an example:

```
#!/usr/bin/env python
# Error-checking gethostbyaddr() example - Chapter 4
# gethostbyaddr-paranoid.py
# Performs a reverse lookup on the IP address given on the command line
# and sanity-checks the result.

import sys, socket

def getipaddrs(hostname):
    """Get a list of IP addresses from a given hostname. This is a standard
    (forward) lookup."""
    result = socket.getaddrinfo(hostname, None, 0, socket.SOCK_STREAM)
    return [x[4][0] for x in result]

def gethostname(ipaddr):
    """Get the hostname from a given IP address. This is a reverse
    lookup."""
    return socket.gethostbyaddr(ipaddr)[0]

try:
    # First, do the reverse lookup and get the hostname.
    hostname = gethostname(sys.argv[1]) # could raise socket.error

    # Now, do a forward lookup on the result from the earlier reverse
    # lookup.
    ipaddrs = getipaddrs(hostname)      # could raise socket.gaierror
```

```

except socket.error, e:
    print "No host names available for %s; this may be normal." % sys.argv[1]
    sys.exit(0)
except socket.gaierror, e:
    print "Got hostname %s, but it could not be forward-resolved: %s" % \
        (hostname, str(e))
    sys.exit(1)

# If the forward lookup did not yield the original IP address anywhere,
# someone is playing tricks. Explain the situation and exit.

if not sys.argv[1] in ipaddrs:
    print "Got hostname %s, but on forward lookup," % hostname
    print "original IP %s did not appear in IP address list." % sys.argv[1]
    sys.exit(1)

# Otherwise, show the validated hostname.
print "Validated hostname:", hostname

```

If you run this code with most addresses, you'll either receive a validated hostname message or a message saying there are no hostnames available. Here's an example:

```

$ ./gethostbyaddr-paranoid.py 192.0.34.166
Validated hostname: www.example.com
$ ./gethostbyaddr-paranoid.py 192.168.6.7
No host names available for 192.168.6.7; this may be normal.

```

Obtaining Information About Your Environment

There are some bits of information that you can obtain about the machine your program is running on. The most important is the hostname, which may be required by some applications such as loggers. You can also obtain an IP address, but where possible, you should get this information from the local endpoint of a connected socket as documented in Chapter 2 instead.

To do this, you'll use two new functions. The first is `socket.gethostname()`. It takes no arguments and simply returns a string. This string represents the hostname of the local machine as configured in the operating system. It usually isn't fully qualified; that is, you'll get `erwin` instead of `erwin.example.com`.

The second function is `socket.getfqdn()`. It takes a single parameter, a hostname, and attempts to make it fully qualified. That is, if your machine is `erwin` and is in the DNS zone `example.com`, it will give you `erwin.example.com`.

It does this by querying the operating system for certain configuration information. Depending on the configuration, it may or may not be able to obtain any additional information.

Therefore, to get your own fully qualified name and IP address, you can first use `gethostname()` to get your hostname. Next, use `getfqdn()` to make it fully qualified. Finally, use `getaddrinfo()` to obtain the IP addresses that correspond with the hostname. Here's an example:

```
=!/usr/bin/env python
# Basic gethostbyaddr() example - Chapter 4 - environment.py

import sys, socket

def getipaddrs(hostname):
    """Given a host name, perform a standard (forward) lookup and
    return a list of IP addresses for that host."""
    result = socket.getaddrinfo(hostname, None, 0, socket.SOCK_STREAM)
    return [x[4][0] for x in result]

# Calling gethostname() returns the name of the local machine
hostname = socket.gethostname()
print "Host name:", hostname

# Try to get the fully qualified name.
print "Fully-qualified name:", socket.getfqdn(hostname)
try:
    print "IP addresses:", ", ".join(getipaddrs(hostname))
except socket.gaierror, e:
    print "Couldn't get IP addresses:", e
```

When you run that, you should obtain something like this:

```
$ ./environment.py
Host name: erwin
Fully-qualified name: erwin.example.com
IP addresses: 127.0.0.1
```

In this case, the IP address isn't particularly useful, because it's merely the loopback interface address. Most machines have the loopback and at least one network device (such as Ethernet) configured; the IP address that you receive back may be either of these. So the result of greatest use from this example is the local hostname. Also keep in mind that many systems are on private networks, and neither the hostname nor the fully qualified name may be reachable on the public Internet.

Using PyDNS for Advanced Lookups

Although the operating-system support is suitable for most purposes, there are situations in which you need more information. Examples include SMTP (e-mail) clients, DNS query tools, and diagnostic utilities.

PyDNS provides a much more capable interface to the DNS system, though to use it properly, you'll have to understand some more details about DNS. A special caveat exists: PyDNS does not, in general, query your operating system's own files such as /etc/hosts, so you may not get access to special local names with it.

DNS Records

When you perform any type of lookup, whether through PyDNS or the operating-system query functions described earlier, you're fetching records of a certain type from a name server. Here's a list of the most common records you'll encounter:

- A records give an IP address for a hostname.
- AAAA records give an IPv6 address for a hostname.
- CNAME records name aliases; given a hostname, a CNAME record will point to the hostname of an A record that provides the final IP address. In some cases, these can also point to PTR records rather than A records.
- MX records name the mail exchangers (servers) in order of preference. MX records also have an order; servers with the lowest order number are tried first.
- PTR records provide the hostname for an IP address and are used for reverse lookups.
- NS records name the name servers for a domain.
- TXT records store arbitrary information about a host, such as a description.
- SOA records hold the Start Of Authority information, which has some information about how the record is stored in DNS.

The forward resolving queries provided by the operating system operate solely on A, AAAA, and CNAME records. The operating system's reverse resolving queries operate solely on PTR and CNAME records. Thus, to obtain any other kind of information, you must use PyDNS or a similar DNS library.

NOTE At the time this book was written, PyDNS did not yet support IPv6. If you require advanced IPv6 queries, you may wish to consider `dnspython`. See the “Python DNS Libraries” sidebar at the beginning of this chapter.

Installing PyDNS

PyDNS doesn’t ship as part of the standard Python distribution. Rather, it must be separately installed. You can download it from its website at <http://pydns.sourceforge.net/>; follow the instructions included in the download for installation. Users of Debian GNU/Linux can also install it by running `apt-get install python-dns`. Please install this package before attempting any examples in this chapter.

Simple PyDNS Queries

The PyDNS library provides the Python module `DNS`. The first thing you’ll want to do in your application is call `DNS.DiscoverNameServers()`. This will find the name servers for your system using the registry on Windows or `/etc/resolv.conf` on UNIX systems. These are the servers that all PyDNS queries will be sent to.

CAUTION `DNS.DiscoverNameServers()` will not work on some older platforms, such as Mac OS 9, that provide neither `/etc/resolv.conf` nor registry entries. Fortunately, these platforms are infrequently used with Python today. Also, if you’re running on Windows and use DHCP, `DNS.DiscoverNameServers()` may not work since DHCP on Windows doesn’t publicize the name server information in the registry. If `DNS.DiscoverNameServers()` fails, please contact your ISP or network administrator to obtain your local name servers. You can then remove the call to `DNS.DiscoverNameServers()` in these examples, and instead set your name servers like this:

```
DNS.defaults['server'] = ['192.168.5.6', '192.168.5.7'].
```

After initializing the name servers, you next need to create a request object. The request object is created by calling `DNS.Request()`, and can be used for issuing any DNS lookup requests.

The request object’s `req()` method is used to perform the actual lookup. It typically takes two named arguments: `name`, which gives the actual name to look up; and `qtype`, which specifies one of the record types from the previous list.

When you issue your query using the `request` object, PyDNS returns an answer object with the result. The `answer` object has an attribute named `answers` that contains a list of all answers returned. Here's a simple example:

```
#!/usr/bin/env python
# Basic DNS library example - Chapter 4 - DNS-basic.py

import sys, DNS

query = sys.argv[1]
DNS.DiscoverNameServers()

reqobj = DNS.Request()

answerobj = reqobj.req(name = query, qtype = DNS.Type.ANY)
if not len(answerobj.answers):
    print "Not found."
for item in answerobj.answers:
    print "%-5s %s" % (item['typename'], item['data'])
```

If you take a look at this program, you'll notice a few things. First, if you supply an invalid hostname, rather than getting an exception, you simply get an empty result. Second, there's a special case for the query type ANY in that it sometimes misses MX records (and others) if they aren't requested first. In most regular programs, you will not use ANY, so this should not be a large problem.

When you run this program, you'll get results like this:

```
$ ./DNS-basic.py apress.com
MX      (50, 'mail.uu.net')
MX      (10, 'mailt1.apress.com')
MX      (20, 'maildsl.apress.com')
MX      (30, 'mailt1backup.apress.com')
MX      (40, 'maildslbackup.apress.com')
NS      auth120.ns.uu.net
NS      auth111.ns.uu.net
$ ./DNS-basic.py www.apress.com
A      65.215.221.149
$ ./DNS-basic.py nonexistantexampleasdf.uk
Not found.
$ ./DNS-basic.py www.yahoo.com
CNAME www.yahoo.akadns.net
```

The first lookup, for `apress.com`, shows results that are completely unavailable with the operating system's libraries. In this case, you can see the name servers

and mail servers for the record, but because none of them are IP addresses, it will take a bit more work to get the actual IP addresses.

The second lookup, for `www.apress.com`, shows the lookup for a more specific host. The third shows a nonexistent host, and the fourth duplicates an example used earlier. Note that earlier, the operating system resolved the CNAME for you, but here, you must do that manually.

One problem with a query type of `ANY` is that it only returns information cached by your local name servers, which may be incomplete. For example, with the first query, the `A` record for `apress.com` is omitted.

Querying Specific Name Servers

The way around the previous problem with the `ANY` results is to skip your local name servers and issue the query directly to the name server that's authoritative for the domain. To do that, you'll use the system's default name servers to find the authoritative name servers. That's done by looking for `NS` records as close to the domain in question. For instance, if you're looking up `www.server.external.example.com`, it will first look for `NS` records with that name, then with `server.external.example.com`—all the way up to `.com`.

Once that information is obtained, you'll need to try each name server in turn, and use the results from the first one that answers your query.

Here's code that implements this algorithm:

```
= /usr/bin/env python
= Expanded DNS library example - Chapter 4 - DNSany.py

import sys, DNS

def hierquery(qstring, qtype):
    """Given a query type qtype, returns answers of that type for lookup
    qstring. If no answers are found, removes the most specific component
    (the part before the leftmost period) and retries the query with the
    result. If the topmost query fails, returns None."""
    reqobj = DNS.Request()
    try:
        answerobj = reqobj.req(name = qstring, qtype = qtype)
        answers = [x['data'] for x in answerobj.answers if x['type'] == qtype]
    except DNS.Base.DNSError:
        answers = [] # Fake an empty return
    if len(answers):
        return answers
```

```

else:
    remainder = qstring.split(".", 1)
    if len(remainder) == 1:
        return None
    else:
        return hierquery(remainder[1], qtype)

def findnameservers(hostname):
    """Attempts to determine the authoritative nameservers for a given
hostname. Returns None on failure."""
    return hierquery(hostname, DNS.Type.NS)

def getrecordsfromnameserver(qstring, qtype, nslist):
    """Given a list of nameservers in nslist, executes the query requested
by qstring and qtype on each in order, returning the data from the first
server that returned 1 or more answers. If no server returned any answers,
returns []."""
    for ns in nslist:
        reqobj = DNS.Request(server = ns)
        try:
            answers = reqobj.req(name = qstring, qtype = qtype).answers
            if len(answers):
                return answers
        except DNS.Base.DNSError:
            pass
    return []

def nslookup(qstring, qtype, verbose = 1):
    nslist = findnameservers(qstring)
    if nslist == None:
        raise RuntimeError, "Could not find nameserver to use."
    if verbose:
        print "Using nameservers:", ", ".join(nslist)
    return getrecordsfromnameserver(qstring, qtype, nslist)

if __name__ == '__main__':
    query = sys.argv[1]
    DNS.DiscoverNameServers()

    answers = nslookup(query, DNS.Type.ANY)
    if not len(answers):
        print "Not found."
    for item in answers:
        print "%-5s %s" % (item['typename'], item['data'])

```

This example is split into several functions to make it easier to use and integrate into your own code. When the program starts, it calls `nslookup()` with the hostname given on the command line and a query type of ANY.

The `nslookup()` function first calls `findnameservers()` to obtain the list of authoritative name servers, then, using that list, it calls `getrecordsfromnameserver()`, which tries the query on each name server in turn until it gets an answer or exhausts its list. The `findnameservers()` function simply calls `hierquery()` to find the appropriate name server for the given hostname.

Try this program with some of the examples from before and note the difference.

```
$ ./DNSany.py apress.com
Using nameservers: auth120.ns.uu.net, auth111.ns.uu.net
:
 65.215.221.149
  (10, 'mailt1.apress.com')
  (20, 'maildsl.apress.com')
  (30, 'mailt1backup.apress.com')
  (40, 'maildslbackup.apress.com')
  (50, 'mail.uu.net')
  S auth111.ns.uu.net
  S auth120.ns.uu.net
SOA ('auth111.ns.uu.net', 'hostmaster.uu.net', ('serial', 17L),
('refresh', 21600L, '6 hours'), ('retry', 3600L, '1 hours'),
('expire', 1728000L, '2 weeks'), ('minimum', 21600L, '6 hours'))
$ ./DNSany.py nonexistentexampleasdf.uk
Using nameservers: ns4.nic.uk, ns5.nic.uk, sec-nom.dns.uk.psi.net,
ns1.nic.uk, ns2.nic.uk, ns3.nic.uk
Not found.
$ ./DNSany.py www.yahoo.com
Using nameservers: ns4.yahoo.com, ns5.yahoo.com, ns1.yahoo.com,
ns2.yahoo.com, ns3.yahoo.com
CNAME www.yahoo.akadns.net
```

This time, all of the records for the `apress.com` domain showed up—notice the A and SOA records that were missing previously. For some other interesting information, try the following:

```
$ ./DNSany.py .
```

You'll get a list of the top-level name servers!

Resolving Lookup Results

Some records—NS, PTR, CNAME, and MX in particular—return another hostname as part of their data. To get the final IP address, you’ll need to resolve that returned information. You could do that using the operating system’s built-in facilities, or you could continue using the DNS module to do so.

The following code does just that (and a bit more). It will issue an ANY query on the hostname passed on the command line, then issue A or ANY queries when it makes sense. This example uses functions from the previous one, and assumes you have saved that example in a file named `DNSany.py` in your current directory. Here’s the code:

```

#!/usr/bin/env python
# DNS query program - Example 4 - DNSquery.py

import sys, DNS, DNSany, re

def getreverse(query):
    """Given the query, returns an appropriate reverse lookup string
    under IN-ADDR.ARPA if query is an IP address; otherwise, returns None.
    This function is not IPv6-compatible."""
    if re.search('^\d+\.\d+\.\d+\.\d+$', query):
        octets = query.split('.')
        octets.reverse()
        return '.'.join(octets) + '.IN-ADDR.ARPA'
    return None

def formatline(index, typename, descr, data):
    retval = "%-2s %-5s" % (index, typename)
    data = data.replace("\n", "\n      ")
    if descr != None and len(descr):
        retval += " %-12s" % (descr + ":")
    return retval + " " + data

DNS.DiscoverNameServers()
queries = [(sys.argv[1], DNS.Type.ANY)]
donequeries = []
descriptions = {'A': 'IP address',
                'TXT': 'Data',
                'PTR': 'Host name',
                'CNAME': 'Alias for',
                'NS': 'Name server'}

```

```

while len(queries):
    (query, qtype) = queries.pop(0)
    if query in donequeries:
        # Don't look up the same thing twice
        continue
    donequeries.append(query)
    print "-" * 77
    print "Results for %s (lookup type %s)" % \
          (query, DNS.Type.typestr(qtype))
    print
    rev = getreverse(query)
    if rev:
        print "IP address given; doing reverse lookup using", rev
        query = rev

    answers = DNSany.nslookup(query, qtype, verbose = 0)
    if not len(answers):
        print "Not found."

    count = 0
    for answer in answers:
        count += 1
        if answer['typename'] == 'MX':
            print formatline(count, answer['typename'],
                              'Mail server',
                              "%s, priority %d" % (answer['data'][1],
                                                    answer['data'][0]))
            queries.append((answer['data'][1], DNS.Type.A))
        elif answer['typename'] == 'SOA':
            data = "\n" + "\n".join([str(x) for x in answer['data']])
            print formatline(count, 'SOA', 'Start of authority', data)
        elif answer['typename'] in descriptions:
            print formatline(count, answer['typename'],
                             descriptions[answer['typename']], answer['data'])
        else:
            print formatline(count, answer['typename'], None,
                             str(answer['data']))
        if answer['typename'] in ['CNAME', 'PTR']:
            queries.append((answer['data'], DNS.Type.ANY))
        if answer['typename'] == 'NS':
            queries.append((answer['data'], DNS.Type.A))

```

One thing to note here is how reverse lookups are accomplished with PyDNS (and other low-level DNS toolkits). You must reverse the components of the IP address, and then add IN-ADDR.ARPA to the end. So, if you were given the IP address 10.11.12.13, you would look up 13.12.11.10.IN-ADDR.ARPA. This format is the underlying one used by the DNS protocol and must be used for reverse lookups.

Since this program produces voluminous output, I'll show you the output of only two examples here. The first is a reverse lookup, and the second shows the partial output of a forward lookup.

```
$ ./DNSquery.py 65.215.221.149
-----
Results for 65.215.221.149 (lookup type ANY)

IP address given; doing reverse lookup using 149.221.215.65.IN-ADDR.ARPA
1 CNAME Alias for: 149.144.221.215.65.IN-ADDR.ARPA
-----
Results for 149.144.221.215.65.IN-ADDR.ARPA (lookup type ANY)

1 PTR Host name: www.apress.com
-----
Results for www.apress.com (lookup type ANY)

1 A IP address: 65.215.221.149
$ ./DNSquery.py apress.com
-----
Results for apress.com (lookup type ANY)

1 A IP address: 65.215.221.149
2 MX Mail server: mailt1.apress.com, priority 10
3 MX Mail server: maildsl.apress.com, priority 20
4 MX Mail server: mailt1backup.apress.com, priority 30
5 MX Mail server: maildslbackup.apress.com, priority 40
6 MX Mail server: mail.uu.net, priority 50
7 NS Name server: auth111.ns.uu.net
8 NS Name server: auth120.ns.uu.net
9 SOA Start of authority:
    auth111.ns.uu.net
    hostmaster.uu.net
    ('serial', 17L)
    ('refresh ', 21600L, '6 hours')
    ('retry', 3600L, '1 hours')
    ('expire', 1728000L, '2 weeks')
    ('minimum', 21600L, '6 hours')
```

```
+--> [root@...] ~]# nslookup mailt1.apress.com  
mailt1.apress.com  
+--> [root@...] ~]
```

```
+--> [root@...] ~]# nslookup mailt1.apress.com  
mailt1.apress.com  
+--> [root@...] ~]
```

Summary

The Domain Name System (DNS) is used to translate between textual names and the IP addresses used for low-level communication. Python provides an interface to the operating system's own DNS functionality via the `socket` module. You can also choose to use a third-party add-on DNS package such as PyDNS for greater flexibility.

Standard (forward) lookups translate a textual hostname into a numeric IP address and are a common part of connecting to a remote service. Reverse lookups translate an IP address into a hostname, but you should take care to ensure that invalid data isn't being used for the reverse lookups.

You can obtain information about the computer on which a program is running by calling `gethostname()`. From this, you can learn the computer's own hostname.

CHAPTER 5

Advanced Network Operations

TCP/IP NETWORKING, and Python's support for it, provides a number of useful features for different types of programs. Most of them apply to both client and server programs. This chapter demonstrates several of these features.

The features presented in this chapter are generally independent of each other. You can combine several of these techniques into your programs. Here are the different features you'll learn about in this chapter:

- Half-open sockets, which let you close down one direction of communication
- Timeouts, which raise an exception if no network activity occurs after waiting for a certain amount of time
- Techniques for transmitting strings and marking the end of the strings
- Network byte order, typically used for communicating using C-based protocols
- Broadcasts, which send data to several machines at once
- Using IPv6, the next generation Internet protocol
- Binding to specific addresses or interfaces
- Looking for several different events at once by using `poll()` or `select()`

Half-Open Sockets

Normally, sockets are bidirectional—data can be sent across them in both directions. Sometimes, you may want to make a socket be unidirectional so data can only be sent in one direction. A socket that's unidirectional is said to be a *half-open*

socket. A socket is made half-open by calling `shutdown()`, and that procedure is irreversible for that socket. Half-open sockets are useful when

- You want to ensure that all data written has been transmitted. When `shutdown()` is called to close the output channel of a socket, it will not return until all buffered data has been successfully transmitted.
- You want to have a way to catch potential programming errors that may cause the program to write to a socket that shouldn't be written to, or read from a socket that shouldn't be read from.
- Your program uses `fork()` or multiple threads, and you want to prevent other processes or threads from doing certain operations, or you want to force a socket to be closed immediately.

The `socket.shutdown()` call is used to accomplish all of these tasks. Chapter 2 discusses the first case, showing how `shutdown()` can be used to detect errors. In that situation, the fact that the socket was left half-open afterwards was incidental. The second and third situations are new. One reason you might use `shutdown()` is to help make sure your code is correct. For instance, if you have completed your writing and use `shutdown()` to prevent future writing, you'll get an exception if you try to write to the socket in the future. An exception is usually easier to track down than deadlock or protocol miscommunication, so getting one could be very useful.

Another situation arises when your program uses `fork()` or multiple threads. When using `fork()`, calling `close()` on a socket only makes it unavailable to that particular process. The connection isn't actually closed until all processes that use it have either called `close()`, or had the socket go out of scope or deleted, or terminated. When you're done communicating with the remote, you can force the socket shut by telling `shutdown()` to halt communication in *both* directions.

The call to `shutdown()` requires a single argument that indicates how you want to shut down the socket. Its possible values are as follows:

- 0 to prevent future reads
- 1 to prevent future writes
- 2 to prevent future reads and writes

Once shut down in a given direction, the socket can never be reopened in that direction. Calls to `shutdown()` are cumulative; calling `shutdown(0)` followed by `shutdown(1)` will achieve the same effect as calling `shutdown(2)`.

Timeouts

Python 2.3 introduced a new feature for sockets: timeouts. Timeouts are useful for discovering error conditions or communication problems in some instances.

TCP connections can be held open indefinitely, even if there's no traffic flowing across them. This is often useful; for instance, a user who logged on to a server using telnet might be away from the desk for an hour for lunch. However, a consequence of this is that it can take a very long time for errors in communication to be detected.

A program that's used to active communication (for instance, a web server sending a large file to a client) would rightly be alarmed if no activity had taken place for over a minute. Rather than wait on the operating system to notice the failure, you can instruct Python to notice that nothing is happening, and inform you that there may be a problem.

When reading, timeouts are useful to enforce inactivity timeouts from clients. You can also use timeouts to detect broken links when reading or writing.

To enable timeout detection on a Python socket, you call `settimeout()` on the socket, passing it the number of seconds until a timeout is reached. Later, when you make a socket call and nothing has happened for that amount of time, a `socket.timeout` exception is raised.

Here's an example server program, slightly modified from `echoserver.py` in Chapter 3 that illustrates the usage of timeouts.

```
#!/usr/bin/env python
# Echo Server with Timeouts -- Chapter 5 -- timeoutserver.py
import socket, traceback

host = ''                      # Bind to all interfaces
port = 51423

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind((host, port))
s.listen(1)

while 1:
    try:
        clientsock, clientaddr = s.accept()
    except KeyboardInterrupt:
        raise
    except:
        traceback.print_exc()
        continue
```

```

clientsock.settimeout(5)

# Process the connection

try:
    print "Got connection from", clientsock.getpeername()
    while 1:
        data = clientsock.recv(4096)
        if not len(data):
            break
        clientsock.sendall(data)
    except (KeyboardInterrupt, SystemExit):
        raise
    except socket.timeout:
        pass
    except:
        traceback.print_exc()

# Close the connection

try:
    clientsock.close()
except KeyboardInterrupt:
    raise
except:
    traceback.print_exc()

```

You can test this code quite easily. Start the server, then telnet to port 51423. You can type text and it will be sent back to you. However, if you wait for five seconds before sending anything, you'll be disconnected.

When working with socket timeouts, it's important to remember that both reading and writing operations can cause a timeout to occur. A timeout for reading might happen if the client hasn't sent any data. For writing, the client may not have attempted to read the data yet. In either case, a congested or problematic network connection could also cause a timeout. That's why, in this example, the clause that catches `socket.timeout` is in such a place that it covers both the `recv()` and the `sendall()` calls.

Transmitting Strings

One common problem that arises when sending data across the network is that of transmitting variable-length strings. When you read information from a TCP

stream, you don't know when the sender has finished giving you a piece of data unless you build some sort of indication into your protocol. In many of the examples in Chapters 1 through 3, I made one end close the socket when done. While that works fine for simple protocols, some more advanced systems will require more sophisticated measures.

There are two common approaches to solving this problem: a unique end-of-string identifier and a leading fixed-length size indicator.

Unique End-of-String Identifiers

With this method, the sending end will append an end-of-string identifier after sending some text. This identifier is typically the NULL character ('\0' in Python) or the newline character ('\n' in Python). If you use the newline character, you can conveniently use the `readline()` method on file-like objects to get data from it. Many network protocols, such as HTTP, FTP, and SMTP, are at least partially based on having strings terminated by a newline character. However, newlines occur frequently in text, so that will be a problem if a single piece of data can have multiple lines. In these situations, you may prefer to resort to a NULL character or some other unique identifier.

Another problem is that if your string contains binary data, it may contain a NULL character within itself somewhere. Therefore, for those situations, there's no unique end-of-string identifier available, and you'll have to either fix the problem or resort to a leading size indicator. Some ways to fix the problem include escaping, data encoding, and adjustable end-of-string identifiers.

Escaping

To use escaping, you would devise a way to include the end-of-string identifier within the text. For instance, if your end-of-string identifier is the newline character '\n', then you could add a backslash before sending each newline character that's contained within a string. On the receiving side, you can know that a backslash followed by a newline indicates that the character is a literal newline instead of the end of a string.

However, now you'll have another problem: The backslash itself is now a special character. So, you'll also have to escape it—perhaps just double it—each time you use it on the network. Also, keep in mind that a backslash is special to Python strings, too—you have to double it in a literal string to make it work properly. Table 5-1 illustrates the concept. In the table, I'll use <newline> to represent the newline character.

Table 5-1. Example of Escaping Logic

Data Received	Python String	Meaning
\<newline>	"\\n"	Insert the newline character into the string being read and continue reading.
\\"	"\\\\\"	Insert a single backslash (Python string "\\") into the string being read and continue reading.
<newline>	"\n"	Stop reading.

Data Encoding

You can encode your data so that it's impossible for the end-of-string indicator to occur. One option is base-64 encoding, which represents arbitrary binary data using printable characters. Python's `base64` module will handle this for you. One problem, however, is that this approach will increase the size of data transmitted.

Adjustable End-of-String Indicators

A final approach is to send a unique end-of-string indicator before sending your string. You can often use a random algorithm to generate such an indicator, then ensure that it doesn't occur within the string itself. This end-of-string indicator can often be guaranteed to not contain any newlines or other confusing data, so it can be easily sent. If you don't know exactly what data will be sent in advance, though, this option won't work for you. That's because you'll be unable to verify that your end-of-string indicator really doesn't occur anywhere in the data that will be sent.

Leading Size Indicator

With this method, you first transmit a fixed-width count of the number of bytes to follow. The reading end then simply reads the number of bytes.

The drawbacks are that the transmitting side must know the number of bytes to be sent in advance. Also, you must ensure that the fixed-width count is large enough to accommodate the longest possible string. You could send a fixed-width count as a series of ASCII digits if you make sure they have leading zeros. Most people will use a binary fixed-width count; however, you can store a much larger number in a smaller space that way. The next section describes how to generate a binary leading size indicator.

Understanding Network Byte Order

When you send integer data across the network, there are two common choices for representing it:

- A string of ASCII characters that the receiving side will parse
- A binary word, typically either 16 or 32 bits long

The first option is fairly straightforward. That makes it easy to use and debug, and is the reason why most of the protocols discussed in this book work that way. However, the second option merits some more discussion.

Binary words are especially common with protocols that were first implemented in C, since sending binary data in C is often easier than dealing with string conversions. However, it's not quite that easy, since different platforms have different ways of encoding binary data.

To solve that problem, there's a standard representation of binary data called *network byte order*. Before sending a binary integer, it's converted to network byte order. The receiving side converts it from network byte order to the local representation before using it.

The Python module `struct` provides support for converting between Python and binary data. The `struct` module works based on a format string. This string describes how to process the binary data. Although there are many variations on what's acceptable in that format string, here you'll use two basic formats: `H`, which is used for 16-bit integers, and `I`, which is used for 32-bit integers. To tell Python exactly how to encode an integer, you add a modifier before the basic format. The exclamation point modifier instructs `struct` to use network byte order for encoding and decoding. Here's an example program that encodes and decodes a length word for a string:

```
=!/usr/bin/env python
= Network Byte Order - Chapter 5 - nbo.py
import struct, sys

def htons(num):
    return struct.pack('!H', num)

def htonl(num):
    return struct.pack('!I', num)

def ntohs(data):
    return struct.unpack('!H', data)[0]
```

```

def ntohs(data):
    return struct.unpack('!I', data)[0]

def sendstring(data):
    return htonl(len(data)) + data

print "Enter a string:"
str = sys.stdin.readline().rstrip()

print repr(sendstring(str))

```

The functions in this example convert between Python ints and longs and strings. The strings used contain binary data suitable for writing out to a socket. In this example, I use a 32-bit integer to hold the length of the string; that provides for a maximum string size of approximately 4 GB.

Try running this program. You'll see something like this:

```

$ ./nbo.py
Enter a string:
Test.
'\x00\x00\x00\x05Test.'
$ ./nbo.py
Enter a string:
Hello, I am testing this
'\x00\x00\x00\x18Hello, I am testing this'

```

Because the four binary characters cannot be read on the terminal, the example uses Python's `repr()` function to show them. The `repr()` function simply displays a string using the same format that you would use if you were typing it in to a Python program. You can see how this has worked. For the first example, you see four bytes representing the 32-bit integer. The number shown is 5, which corresponds exactly to the length of "Test.". In the second example, the number is 18 in hexadecimal, or 24 in decimal. Again, that's the correct length.

You might notice that Python's socket module also includes functions such as `htonl()`. However, those functions are of little practical use. They only work once you've already converted binary data to a Python numeric type. Since the `struct` functions provide the exclamation mark format specifier, there's no need for the socket module `htonl()` functions.

Using Broadcast Data

Although most TCP/IP operations can be used on either a local area network (LAN) or the Internet at large, there are a few features that are specific to usage on a LAN. Of these, by far the most prominent and useful is broadcast data.

Broadcast data doesn't work with TCP; it's usually implemented with UDP.

Ordinarily, when you send a UDP message, it's sent from a single machine to a single machine. When you broadcast a UDP packet, it's sent to *all* machines connected to your LAN. The underlying transport, such as Ethernet, will have a special mode that lets you do this without having to repeat the packet for each computer.

On the receiver's side, when a broadcast packet is received, the kernel looks at the destination port number. If it has a process listening to that port, the packet is sent to that process. Otherwise, it's silently discarded. Therefore, simply sending out a broadcast packet will not harm or impact machines that don't have a server listening for it.

Broadcast packets are often used for the following types of activities:

- Automatic service discovery: For instance, a computer might send out a broadcast packet looking for all print servers of a particular type.
- Automatic service announcements: A server providing a service for a LAN might periodically broadcast the availability of that service. Clients would listen for those broadcasts.
- Searching for LAN computers that implement a specific protocol. For instance, a chat program might send out a broadcast packet looking for other people on the LAN with the same chat program. It might then compile a list and present it to the user.

When working with a socket that will be either sending or receiving broadcasts, you must first call `s.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1)` to enable support for broadcasting on the socket. To send a broadcast, you use the special address "`<broadcast>`" instead of a normal IP address or hostname.

There are two sides to working with broadcasts: the sender and the receiver. First, here's the receiver:

```

#!/usr/bin/env python
# UDP Broadcast Server - Chapter 5 - bcastreceiver.py
import socket, traceback

host = ''                                # Bind to all interfaces
port = 51423

s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1)
s.bind((host, port))

while 1:
    try:
        message, address = s.recvfrom(8192)
        print "Got data from", address
        # Acknowledge it.
        s.sendto("I am here", address)
    except (KeyboardInterrupt, SystemExit):
        raise
    except:
        traceback.print_exc()

```

Note that this code is very similar to that of the UDP echo server in Chapter 2. In fact, for many UDP servers, you can make them broadcast-aware by simply adding the second call to `setsockopt()`; they'll then be able to receive and process both broadcast and nonbroadcast data.

The sending application also looks familiar, as shown here:

```

#!/usr/bin/env python
# Broadcast Sender - Chapter 5 - bcastsender.py

import socket, sys
dest = ('<broadcast>', 51423)

s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1)
s.sendto("Hello", dest)
print "Looking for replies; press Ctrl-C to stop."
while 1:
    (buf, address) = s.recvfrom(2048)
    if not len(buf):
        break
    print "Received from %s: %s" % (address, buf)

```

To test this out, you'll need at least two computers on a LAN. (You can simulate it with only a single computer, but you won't see the effect of receiving data from several machines.) Start up the broadcast receiver on as many computers as you would like. Then run the broadcast sender. It should print out one line for each computer that's running the broadcast receiver, as follows:

```
$ ./bcastsender.py
Looking for replies; press Ctrl-C to stop.
Received from ('10.200.0.5', 51423): I am here
Received from ('10.200.0.7', 51423): I am here
```

In this case, two different computers were running the broadcast server when the broadcast sender was invoked. They each replied. Note that the reply is *not* a broadcast; there's no need for it to be, and in fact, doing so would be incorrect. The server simply receives the broadcast and then sends a normal reply to the client.

One final warning about broadcasts: Use them sparingly. Don't try to shove a lot of data through them unless you really have to, and test your code well. Errant code that generates a flood of broadcast packets can literally bring a network to its knees. That said, many common services, such as DHCP, make good use of broadcasts.

Working with IPv6

Most of the examples in this book relate specifically to TCP/IP version 4, also known as IPv4. IPv4 is used by virtually every device on the Internet, LANs, and other networks. Yet it has some problems, the most severe of which relate to lack of address space. IPv4 has a 32-bit address space, thereby providing a theoretical maximum of 4 billion addresses. IPv6 has a 128-bit address space, thereby providing a theoretical maximum of 340 undecillion (or $3.40 * 10^{38}$) addresses. That's 79 octillion times more addresses than are available with IPv4. Note that due to practical restrictions, these theoretical upper bounds will never actually be realized and the shortage of IPv4 addresses is already becoming a problem.

To address these problems, standards organizations are developing the next-generation protocol, IPv6. IPv6 is presently still officially in development; however, all popular modern operating systems already support it either by default or with optional patches, as do modern versions of Python and many other applications.

What About IPv5?

Each packet of data going across the Internet has a version field that indicates which protocol version is in use. IPv4, of course, uses a version number of 4. A family of protocols known as the Stream Protocol (ST, ST+, ST2, and ST2+) was already assigned number 5 back in 1979. Therefore, the new version of IP had to use the next available number, 6.

When you write programs in Python that use IPv6, the difference between it and IPv4 will mostly be felt as you set up a connection. Once a connection is established, IPv6 functions virtually identically to IPv4.

Resolving Addresses

One of the challenges you face when supporting IPv6 is that you may be in a situation where both IPv4 and IPv6 addresses are available for a given service. You'll then have to decide whether to use an IPv4 or IPv6 connection to a server. Alternatively, you'll have to support only one protocol or resolve the address first one way, then the other.

Most of the IPv4 examples use code that follows this general pattern:

```
port = 51423
host = 'localhost'
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((host, port))
```

One thing to note is that `socket.AF_INET` is specifically an IPv4 socket; to use IPv6, you have to use `socket.AF_INET6`. Therefore, you either need to try both, or take a shortcut. That shortcut is `socket.getaddrinfo()`. The `getaddrinfo()` function can search for addresses using one protocol or both. Here's a program that illustrates its use:

```
#!/usr/bin/env python
# getaddrinfo() display - Chapter 5 - getaddrinfo.py
# Give a host and a port on the command line

import socket, sys
host, port = sys.argv[1:]
```

```

: Look up the given data
results = socket.getaddrinfo(host, port, 0, socket.SOCK_STREAM)

: We may get multiple results back. Display each one.
for result in results:
    # Display a separator line to visually segment one result from the next.
    print "-" * 60

    # Print whether we got back an IPv4 or IPv6 result.
    if result[0] == socket.AF_INET:
        print "Family: AF_INET"
    elif result[0] == socket.AF_INET6:
        print "Family: AF_INET6"
    else:
        # It's not IPv4 or IPv6, so we don't know about it. Just print
        # out its protocol number.
        print "Family:", result[0]

    # Indicate whether it's a stream (TCP) or datagram (UDP) result.
    if result[1] == socket.SOCK_STREAM:
        print "Socket Type: SOCK_STREAM"
    elif result[1] == socket.SOCK_DGRAM:
        print "Socket Type: SOCK_DGRAM"

    # Display the final bits of information from getaddrinfo()

    print "Protocol:", result[2]
    print "Canonical Name:", result[3]
    print "Socket Address:", result[4]

```

Try running that against an address that exists only with IPv4, as shown here:

```

$ ./getaddrinfo.py movies.yahoo.com http
-----
Family: AF_INET
Socket Type: SOCK_STREAM
Protocol: 6
Canonical Name:
Socket Address: ('66.218.71.147', 80)

```

You can see it returned all the information that you need in order to establish a socket (AF_INET and SOCK_STREAM) and connect to the remote host (the socket address). However, `getaddrinfo()` can sometimes return multiple results, as follows:

```
$ ./getaddrinfo.py www.ipv6.org http
-----
Family: AF_INET6
Socket Type: SOCK_STREAM
Protocol: 6
Canonical Name:
Socket Address: ('2001:6b0:1:ea:a00:20ff:fe8f:708f', 80, 0, 0)
-----
Family: AF_INET
Socket Type: SOCK_STREAM
Protocol: 6
Canonical Name:
Socket Address: ('130.237.234.41', 80)
```

In this instance, the name server for `www.ipv6.org` defined addresses for both IPv4 and IPv6, and `getaddrinfo()` returned both. The task of the program is now to choose whether to use an IPv4 or IPv6 address.

Why You May Not Get IPv6 Results

You may try the `getaddrinfo.py` example but receive no IPv6 results from it. In order to receive IPv6 results from `getaddrinfo()` and be able to communicate over IPv6, your system must be IPv6-ready. That means that it must support IPv6 results from DNS and IPv6 communication protocols. Some older operating systems don't support IPv6. Other operating systems support it as an optional feature that must be compiled into the kernel or patched on to the system.

If you don't get IPv6 results here, chances are that you need to make sure that your operating system is configured to support IPv6 and that your copy of Python is also configured to support IPv6. If you need to upgrade, you should upgrade your operating system first, then Python, since Python can only support the features it finds in the operating system at compile time.

Handling Family Preferences

When you call `getaddrinfo()`, you tell it the kind of information you're looking for. The example code passes along the host and port information from the command line, as well as `SOCK_STREAM` in order to request a TCP socket (you could also use `SOCK_DGRAM` to request a UDP socket). There's also a way to specify a protocol such

as AF_INET or AF_INET6; however, instead of requesting a specific protocol, the example program used a zero. That tells getaddrinfo() to search both protocols.

In general, you'll probably want to prefer the IPv4 result if one exists, and fall back to IPv6 only if no IPv4 address exists. It's quite possible for a system to look up IPv6 addresses even if it cannot communicate using IPv6. Here's an example that prefers IPv4 but uses IPv6 if IPv4 is unavailable:

```
= /usr/bin/env python
= Connect Example with IPv6 Awareness - Chapter 5 - ipv6connect.py

import socket, sys

def getaddrinfo_pref(host, port, socktype, familypreference = socket.AF_INET):
    """Given a host, port, and socktype (usually socket.SOCK_STREAM or
    socket.SOCK_DGRAM), looks up information with both IPv4 and IPv6.  If
    information is found corresponding to the familypreference, it is returned.
    Otherwise, any information found is returned.  The familypreference
    defaults to IPv4 (socket.AF_INET) but you could also set it to
    socket.AF_INET6 for IPv6.

The return value is the appropriate tuple returned from
socket.getaddrinfo()."""
results = socket.getaddrinfo(host, port, 0, socktype)
for result in results:
    if result[0] == familypreference:
        return result
return results[0]

host = sys.argv[1]
port = 'http'

c = getaddrinfo_pref(host, port, socket.SOCK_STREAM)
print "Connecting to", c[4]
s = socket.socket(c[0], c[1])
s.connect(c[4])
s.sendall("HEAD / HTTP/1.0\r\n\r\n")

while 1:
    buf = s.recv(4096)
    if not len(buf):
        break
    sys.stdout.write(buf)
```

When you run this program, it will attempt to make a connection using IPv4, if available, or IPv6 otherwise. Here are two examples:

```
$ ./ipv6connect.py www.ipv6.org
Connecting to ('130.237.234.41', 80)
HTTP/1.1 200 OK
...
$ ./ipv6connect.py www.ipv6.bieringer.de
Connecting to ('2001:7b0:1101:2::146:6', 80, 0, 0)
HTTP/1.1 200 OK
...
```

Binding to Specific Addresses

Many systems support more than one network interface at once. In fact, it's not uncommon to find a system with Ethernet, a dial-up connection, and a loopback interface. The loopback interface is present almost everywhere, and is a virtual interface that lets you communicate with processes on the local machine. It's addressed as 127.0.0.1 or localhost.

Each interface can have its own IP address, and in some operating systems, it can even have multiple IP addresses. Most of the servers presented in this book bind to any address. That is, you'll see code like the following frequently:

```
host = ''                                # Bind to all interfaces
port = 51423
...
s.bind((host, port))
```

This means that the server listens on port 51423 on all interfaces. Normally, this is fine. However, there are situations in which it isn't, like the following:

- A server may have multiple IP addresses for virtual-hosting purposes, where each IP address corresponds to a different site. Server programs for a specific site should listen only on its IP address.
- A program may want to accept connections only from other programs on the same system for security reasons. In this case, it should bind to 127.0.0.1.

- A firewall or router may have interfaces to both an internal and an external network. For security reasons, programs running on such a system may wish to accept connections from only the internal network.
- A client on a machine that's connected to multiple networks may wish to explicitly choose which network to use for a connection.

Most of these examples involve a server, and that's no coincidence; binding a specific address is almost never used for a client. Normally, a server program will learn what address to bind to from a configuration file; however, to keep the example short, this server just hard-codes the address, as shown here:

```

#!/usr/bin/env python
# Echo Server Bound to Specific Address - Chapter 5 - bindserver.py
# Import socket, traceback

host = '127.0.0.1'
port = 51423

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind((host, port))
s.listen(1)

while 1:
    clientsock, clientaddr = s.accept()
    # Process the connection
    print "Got connection from", clientsock.getpeername()
    while 1:
        data = clientsock.recv(4096)
        if not len(data):
            break
        clientsock.sendall(data)
    # Close the connection

    clientsock.close()

```

After you run this server, you can observe the effects of the more specific bind. If you have a host with an IP address other than 127.0.0.1, you could normally connect to port 51423 on that address; now you cannot. That means that it will be secure against any attempt to infiltrate it from any other machine.

Using Event Notification with `poll()` or `select()`

Ordinarily, I/O on sockets will *block*. That is to say, execution of your program will not continue until an operation is complete. For instance, if you're reading data from a socket, your program will stall at the call to `recv()` until you receive something. Normally this is exactly what you want to have happen. Sometimes, though, you might prefer to simply discover that nothing is ready yet and check back again later, as in the following cases:

- You want your user interface to remain active and responsive even while waiting for network data.
- You want to be able to handle more than one network-related task at once without the need for forking or threads.
- You want to be performing other computation while waiting on the network.

Fundamentally, in nonblocking mode, calls to `send()` and `recv()` will raise a `socket.error` exception if you attempt to send or receive data when the socket isn't yet ready. This is useful, but repeatedly checking to see if the socket is ready is both cumbersome and wasteful of CPU resources. Therefore, that particular feature is rarely used.

Instead, two standard mechanisms are available: `select()` and `poll()`. Both let you inform the operating system which sockets are "interesting" to your program. When an event occurs on a socket, the operating system tells you what happened, and you can then handle it. The `select()` interface is older and more widespread, but it's more cumbersome and can grow slow if you're watching many sockets (as could be the case for server programs; see Chapter 22). Therefore, these examples will start with `poll()`. If you need `select()`, there's an example at the end of the chapter. Please note that Windows doesn't support `poll()`; you must use `select()` on that platform.

First, to help illustrate things, here's a simple server program that will send one line of text to the client every five seconds. This program doesn't use `poll()`, but you'll use it together with a client that does, as shown here:

```
#!/usr/bin/env python
# Delaying Server - Chapter 5 - delayserver.py
import socket, traceback, time

host = ''                                # Bind to all interfaces
port = 51423
```

```
- socket.socket(socket.AF_INET, socket.SOCK_STREAM)
setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
bind((host, port))
listen(1)

while 1:
    try:
        clientsock, clientaddr = s.accept()
    except KeyboardInterrupt:
        raise
    except:
        traceback.print_exc()
        continue

    # Process the connection

    try:
        print "Got connection from", clientsock.getpeername()
        while 1:
            try:
                clientsock.sendall(time.asctime() + "\n")
            except:
                break
            time.sleep(5)
    except (KeyboardInterrupt, SystemExit):
        raise
    except:
        traceback.print_exc()

    # Close the connection

    try:
        clientsock.close()
    except KeyboardInterrupt:
        raise
    except:
        traceback.print_exc()
```

Now, let's say that you have a client in which you would like it to display an interesting little animated pattern on the screen while it's waiting for data to arrive from the network. Normally, this would be impossible, because although you're waiting for `recv()` to return data, you can't do anything else. Now, however, it's easy. Here's a sample program:

```
#!/usr/bin/env python
# Nonblocking I/O - Chapter 5 - pollclient.py

import socket, sys, select
port = 51423
host = 'localhost'

spinsize = 10
spinpos = 0
spindir = 1

def spin():
    global spinsize, spinpos, spindir
    spinstr = '.' * spinpos + \
              '|' + '.' * (spinsize - spinpos - 1)
    sys.stdout.write('\r' + spinstr + ' ')
    sys.stdout.flush()

    spinpos += spindir
    if spinpos < 0:
        spindir = 1
        spinpos = 1
    elif spinpos >= spinsize:
        spinpos -= 2
        spindir = -1

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((host, port))

p = select.poll()
p.register(s.fileno(), select.POLLIN | select.POLLERR | select.POLLHUP)
while 1:
    results = p.poll(50)
    if len(results):
        if results[0][1] == select.POLLIN:
            data = s.recv(4096)
            if not len(data):
                print("\rRemote end closed connection; exiting.")
                break
            # Only one item in here -- if there's anything, it's for us.
            sys.stdout.write("\rReceived: " + data)
            sys.stdout.flush()
```

```

else:
    print "\rProblem occurred; exiting."
    sys.exit(0)
spin()

```

Run this and you'll see output like so:

```

$ ./pollclient.py
--> received: Sat Oct 18 14:01:29 2003
--> received: Sat Oct 18 14:01:34 2003
--> received: Sat Oct 18 14:01:39 2003
.....|...

```

There are a few things to note about this example. First, there are several different things that you can poll for; in this case, you should be interested in incoming data and errors, so the program sets POLLIN, POLLERR, and POLLHUP. You can also use POLLOUT (at least one packet can be sent without delay), POLLPRI (urgent data is ready for reading), and POLLNVAL (invalid request).

The call to `select.poll()` returns a `poll` object, `p`. You then register as many sockets as you wish to watch. In the loop, the program calls `p.poll(50)`. The numeric value is optional and indicates the number of milliseconds to wait for something to happen. If nothing happens, `p.poll()` returns an empty list. This example uses that behavior to make sure the animation is updated at least once every 20th of a second.

The `poll()` objects can also be used for servers. For a detailed discussion of such a use, please refer to Chapter 22.

Using select()

The `select()` method of handling I/O without blocking is implemented as a single function call. Python defines `select()` like this:

```

select(iwtd, owtd, ewtd[, timeout])

```

You can pass three items to `select()`: a list of file objects to watch for input, a list of file objects to watch for output, and a list of file objects to watch for errors. An optional fourth parameter specifies a timeout as a floating-point number of seconds. If it's zero, `select()` always returns immediately. If not given, `select()` will block until an event occurs.

The `select()` call returns a three-tuple. Each component of the tuple is a list of objects that are "ready," in the same order as the preceding parameters. Here's a version of the `poll()` client, rewritten to use `select()` instead:

```
#!/usr/bin/env python
# Nonblocking I/O with select() - Chapter 5 - selectclient.py

import socket, sys, select
port = 51423
host = 'localhost'

spinsize = 10
spinpos = 0
spindir = 1

def spin():
    global spinsize, spinpos, spindir
    spinstr = '.' * spinpos + \
              '|' + '.' * (spinsize - spinpos - 1)
    sys.stdout.write('\r' + spinstr + ' ')
    sys.stdout.flush()

    spinpos += spindir
    if spinpos < 0:
        spindir = 1
        spinpos = 1
    elif spinpos >= spinsize:
        spinpos -= 2
        spindir = -1

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((host, port))

while 1:
    infds, outfds, errfds = select.select([s], [], [s], 0.05)
    if len(infds):
        # Normally, one would use something like "for fd in infds" here.
        # We don't bother since there will only ever be a single file
        # descriptor there.
        data = s.recv(4096)
        if not len(data):
            print("\rRemote end closed connection; exiting.")
            break
        # Only one item in here -- if there's anything, it's for us.
        sys.stdout.write("\rReceived: " + data)
        sys.stdout.flush()
```

```

if len(errfds):
    print "\rProblem occurred; exiting."
    sys.exit(0)
spin()

```

If you run this program, you'll obtain the same result as with the `poll()` client.

Summary

TCP/IP networking is powerful and there are features that apply to both clients and servers. This chapter covers many of them.

You can make sockets half-open by calling `shutdown()`, which can be used to close one direction of communication for error-checking, safeguarding, or buffer-flushing purposes.

Timeouts are new with Python 2.3. They cause your program to receive an exception when nothing happens with a socket for a given amount of time.

Transmitting arbitrary-sized strings can be tricky. The two most common approaches involve sending a special end-of-string marker such as a newline character or sending a binary-length word ahead of the string. If you use an end-of-string marker, you may need to ensure that the string itself will never contain that marker and take steps to resolve ambiguities if it does.

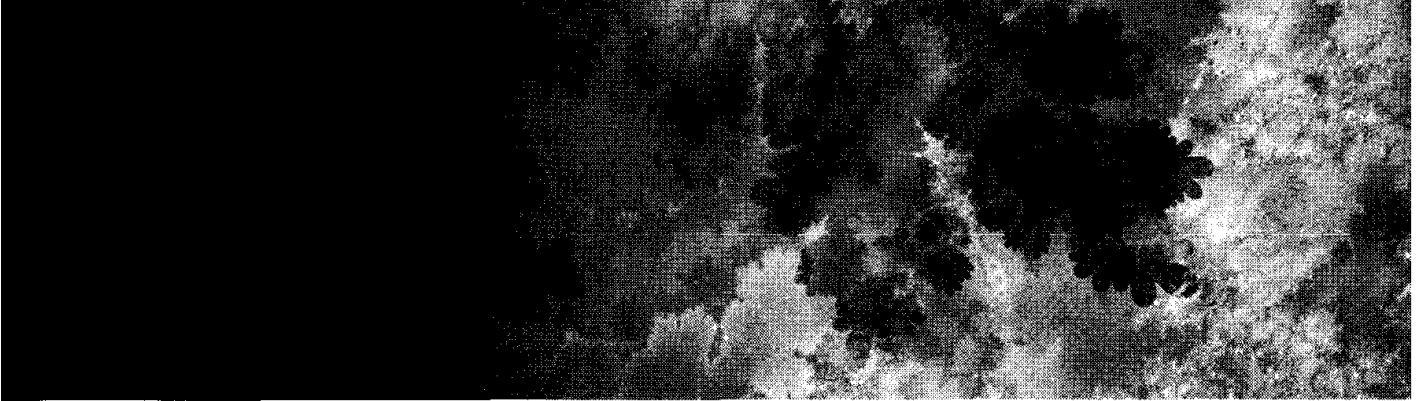
Binary data is sometimes sent across the network. Due to differences in computing platforms, it's usually sent in network byte order. The `Python struct` module helps with the storage and retrieval of binary data.

Broadcast packets can be sent to multiple machines at once and are most often used with UDP. I provided an example that let a client discover the identities of all machines running a particular server on the LAN.

IPv6 is the next-generation Internet protocol. Though already being deployed, it's still technically under development. Python supports IPv6 on platforms that support it. When communicating on IPv6-aware machines, you must often choose between IPv4 and IPv6 for specific endpoints.

Binding to a specific address or interface is one way to ensure that connections are only accepted from a certain set of clients. An example was provided that only accepts connections from the local machine.

You can look for several different events at once by using `select()` or `poll()`. Although `poll()` enjoys a faster and friendlier interface, it isn't as widely supported as `select()`.



Part Two

Web Services

CHAPTER 6

Web Client Access

ONE OF THE INTERNET'S most widely used technologies is the World Wide Web, and that means that its primary protocol, Hypertext Transfer Protocol (HTTP) is something that many programmers must work with. It's no surprise then that Python provides many modules for writing web and HTTP clients.

In this chapter, the `urllib2` module is the focus of discussion. This module actually provides a generic interface to many modules implementing different protocols, though HTTP is clearly the most frequently used protocol for `urllib2`.

urllib vs. urllib2

While the Python modules `urllib` and `urllib2` both provide the same basic functionality, `urllib2` is more extensible and has more built-in features. I recommend that you use `urllib2` for any new code that you write. The `urllib` module does, however, provide a few useful utility functions so you'll see it referenced occasionally in this and other chapters.

In this chapter, you'll learn how to use `urllib2` to do the following:

- Download web pages
- Authenticate to a remote HTTP server
- Submit form data
- Handle errors
- Communicate with protocols other than HTTP

Fetching Web Pages

Downloading a web page from a remote server is a frequent necessity. For instance, you may need to download a weather data page to display a weather conditions icon in an application. Or you might download airline schedules for a travel-planning application.

The following simple program takes a URL on the command line, and simply dumps the page to standard output.

```
#!/usr/bin/env python
# Obtain Web Page - Chapter 6 - dump_page.py
import sys, urllib2

req = urllib2.Request(sys.argv[1])
fd = urllib2.urlopen(req)
while 1:
    data = fd.read(1024)
    if not len(data):
        break
    sys.stdout.write(data)
```

You can run the program and see the HTML code that makes up a web page. Here's an example:

```
$ ./dump_page.py http://www.example.com/
<HTML>
<HEAD>
    <TITLE>Example Web Page</TITLE>
</HEAD>
<body>
<p>You have reached this web page by typing "example.com", "example.net", or "example.org" into your web browser.</p>
<p>These domain names are reserved for use in documentation and are not available for registration. See <a href="http://www.rfc-editor.org/rfc/rfc2606.txt">RFC 2606</a>, Section 3.</p>
</BODY>
</HTML>
```

As you can see, this is a very simple program and works with any protocol supported by `urllib2`—not just HTTP, but also FTP and Gopher. Normally, creating the `urllib2.Request` object is the first thing to do. That object takes the URL, and

You can also set other parameters (such as custom headers to send to a HTTP server) before you open the connection. When the call to `urlopen()` occurs, the object is passed in, and you are handed a file-like object. However, there are a few extra features to that object that let you get a little more detail about the retrieved data. Here's a program that demonstrates that.

```
= /usr/bin/env python
= Obtain Web Page Information - Chapter 6 - dump_info.py
import sys, urllib2

req = urllib2.Request(sys.argv[1])
fd = urllib2.urlopen(req)
print "Retrieved", fd.geturl()
info = fd.info()
for key, value in info.items():
    print "%s = %s" % (key, value)
```

Here's an example result that you'll see after running this program:

```
$ ./dump_info.py http://httpd.apache.org/dev
Retrieved http://httpd.apache.org/dev/
content-length = 9136
accept-ranges = bytes
server = Apache/2.0.48-dev (Unix)
last-modified = Mon, 15 Sep 2003 15:09:09 GMT
connection = close
etag = "e3552a-23b0-a5e1ef40"
date = Wed, 29 Oct 2003 21:17:09 GMT
content-type = text/html; charset=ISO-8859-1
```

Note that the value from `geturl()` is different from the value passed in to the `Request` object; it has a trailing slash. The remote server had issued an HTTP redirect, and `urllib2` was smart enough to automatically follow it. The remaining lines show raw HTTP headers.

Authenticating

Some websites require HTTP authentication to gain access. The most common authentication type is basic authentication, in which the client transmits a username and password to the server. HTTP authentication typically appears as a pop-up window asking for a username and password, and it isn't the same as authentication based on cookies and forms.

Using SSL-Enabled Communication (HTTPS)

HTTP authentication is often combined with encrypted communication using SSL to help secure the transmission of authentication data such as passwords. Python's `urllib2` module has built-in support for `https` URLs. If your Python installation supports SSL, then `https` URLs will be supported automatically. In many cases, you don't need to do anything special to support `https`; it behaves exactly the same way as standard `http` URLs.

If you attempt to access a URL for which authentication is provided, you'll normally get HTTP error 401 (Authorization Required). However, `urllib2` is capable of handling authentication for you. Here's a program that prompts for the authentication information when necessary:

```
#!/usr/bin/env python
# Obtain Web Page Information With Authentication - Chapter 6
# dump_info_auth.py
import sys, urllib2, getpass

class TerminalPassword(urllib2.HTTPPasswordMgr):
    def find_user_password(self, realm, authuri):
        retval = urllib2.HTTPPasswordMgr.find_user_password(self, realm,
                                                          authuri)
        if retval[0] == None and retval[1] == None:
            # Did not find it in stored values; prompt user.
            sys.stdout.write("Login required for %s at %s\n" % \
                             (realm, authuri))
            sys.stdout.write("Username: ")
            username = sys.stdin.readline().rstrip()
            password = getpass.getpass().rstrip()
            return (username, password)
        else:
            return retval

req = urllib2.Request(sys.argv[1])
opener = urllib2.build_opener(urllib2.HTTPBasicAuthHandler(TerminalPassword()))
fd = opener.open(req)
print "Retrieved", fd.geturl()
info = fd.info()
for key, value in info.items():
    print "%s = %s" % (key, value)
```

There are several new things in this program. First, it defines a `TerminalPassword` class that extends the `urllib2.HTTPPasswordMgr` class. The extension simply allows the program to ask the operator for a username and password when necessary. Another new call is `build_opener()`. This function allows additional handlers to be specified. There are certain handlers that are always provided by default (such as basic HTTP and FTP support), and other handlers that can optionally be added. Since this code will support basic authentication, it must add the `HTTPBasicAuthHandler` to the handler chain. In the previous examples, the code simply called `urllib2.urlopen()`, which internally calls `build_opener()` with no arguments. That results in only the default handlers being selected.

Note that once the connection is open, there's no change. If authentication is required, the `HTTPBasicAuthHandler` will call the appropriate function in `TerminalPassword` automatically; no further checks are required. Also note that if you access a normal website that doesn't require authentication, this program behaves identically to the earlier examples.

Here's an example of the behavior of this program. It connects to a page linked from <http://www.unicode.org/mail-arch/>. The username and password are given on that page and, as of the time this text was written, were "unicode-ml" and "unicode", respectively. First, we supply a valid password as follows:

```
$ ./dump_info_auth.py http://www.unicode.org/mail-arch/unicode-ml/
Login required for Unicode-MailList-Archives at www.unicode.org
Username: unicode-ml
Password:
Retrieved http://www.unicode.org/mail-arch/unicode-ml/
Date = Mon, 10 May 2004 14:51:00 GMT
Content-length = 5322
Content-type = text/html; charset=UTF-8
Connection = close
Server = Apache
```

After supplying the username and password, the information was displayed as it was previously. Now, consider what happens if an invalid login is provided:

```
$ ./dump_info_auth.py http://www.unicode.org/mail-arch/unicode-ml/
Login required for Unicode-MailList-Archives at www.unicode.org
Username: badusername
Password:
Traceback (most recent call last):
  File "./dump_info_auth.py", line 23, in ?
    fd = opener.open(req)
  File "/usr/lib/python2.3/urllib2.py", line 326, in open
    '_open', req)
```

```
File "/usr/lib/python2.3/urllib2.py", line 306, in _call_chain
    result = func(*args)
File "/usr/lib/python2.3/urllib2.py", line 901, in http_open
    return self.do_open(httplib.HTTP, req)
File "/usr/lib/python2.3/urllib2.py", line 895, in do_open
    return self.parent.error('http', req, fp, code, msg, hdrs)
File "/usr/lib/python2.3/urllib2.py", line 352, in error
    return self._call_chain(*args)
File "/usr/lib/python2.3/urllib2.py", line 306, in _call_chain
    result = func(*args)
File "/usr/lib/python2.3/urllib2.py", line 412, in http_error_default
    raise HTTPError(req.get_full_url(), code, msg, hdrs, fp)
urllib2.HTTPError: HTTP Error 401: Authorization Required
```

Since `urllib2` wasn't able to log in to the site, it raised `urllib2.HTTPError`. How you wish to handle this is up to you. Some programs may attempt to reconnect and reprompt for a username and password in case the user made a typing error. Others may treat it as a fatal error like this program does.

Submitting Form Data

CGI scripts and other interactive server-side programs often receive data from web clients, usually from forms. Your Python client can send this type of data as well. There are two different ways to submit form data: GET and POST. The method used is normally indicated by the `method` parameter in the `<form>` tag in a HTML document.

Submitting with GET

The GET method of submitting forms encodes form data into the URL. After the path of the page to retrieve is given, a question mark is added, followed by elements of the form. Each key and value pair is separated by an ampersand. Certain characters must be escaped as well. Since all the data is included as part of the URL, the GET method is unsuitable for large amounts of data.

Here's an example of manually constructing a GET request. It uses the `dump_page.py` example from earlier in the chapter to construct a query for "python socket" in the FreeBSD search engine:

```
: ./dump_page.py \
-->http://www.freebsd.org/cgi/search.cgi?words=python+socket&max=25&source=www"
---l>
<head><title>Search Results</title>
<meta name="robots" content="nofollow">
<head>
<body text="#000000" bgcolor="#ffffff">
.
```

In this example, I had to convert the space between "python" and "socket" into a plus character, and there were some other parameters to add as well. Note that in this case, I even had to put quotes around the URL; the ampersands in it can cause trouble for shells. Having to manually generate a suitable search string can be a cumbersome procedure.

Fortunately, Python provides some utilities in `urllib` that help you do this. The following program contains a function to build a URL with GET method data:

```
:usr/bin/env python
# Submit GET Data - Chapter 6 - submit_get.py
import sys, urllib2, urllib

def addGETdata(url, data):
    """Adds data to url. Data should be a list or tuple consisting of 2-item
    lists or tuples of the form: (key, value).

    Items that have no key should have key set to None.

    A given key may occur more than once.
    """
    return url + '?' + urllib.urlencode(data)

zipcode = sys.argv[1]
url = addGETdata('http://www.wunderground.com/cgi-bin/findweather/getForecast',
                  [('query', zipcode)])
print "Using URL", url
req = urllib2.Request(url)
fd = urllib2.urlopen(req)
while 1:
    data = fd.read(1024)
    if not len(data):
        break
    sys.stdout.write(data)
```

The example program takes a single command-line argument—a US zip code—and makes a call to a weather-forecast site to request weather information at that location. The HTML result is sent to standard output for you to use. Here's an example that will get the forecast for New York City:

```
$ ./submit_get.py 10001
Using URL http://www.wunderground.com/cgi-bin/findweather/
getForecast?query=10001

<HTML>
<head>
<META HTTP-EQUIV="Refresh" CONTENT="1800;URL=/cgi-bin/findweather/
getForecast?query=10001">
<title>Weather Underground: New York, New York Forecast</title>
...

```

The function `addGETdata()` is responsible for adding the data, if any, to the end of the URL. Internally, it adds the URL and a question mark to the beginning of the data returned by `urllib.urlencode()`. This function nicely handles all necessary escaping, and its results can just be added to the URL at the proper place.

Submitting with POST

Forms submitted with the POST method work a little differently. Like the GET method, data needs to be encoded. However, unlike GET, the data isn't added on to the URL. Rather, it's sent as a separate part of the request. Therefore, POST is sometimes preferable when large amounts of data are being exchanged. Here's a version of the GET example that uses POST instead. For this particular example, the weather server's script accepts both GET and POST requests. However, be aware that nothing requires servers to accept both types, and many accept only one type.

```
#!/usr/bin/env python
# Submit POST Data - Chapter 6 - submit_post.py
import sys, urllib2, urllib

zipcode = sys.argv[1]
url = 'http://www.wunderground.com/cgi-bin/findweather/getForecast'
data = urllib.urlencode([('query', zipcode)])
req = urllib2.Request(url)
fd = urllib2.urlopen(req, data)
```

```

while 1:
    data = fd.read(1024)
    if not len(data):
        break
    sys.stdout.write(data)

```

This code is very similar to the GET example. Note that the URL is never modified when using POST; instead, the additional information is passed as a second argument to `urlopen()`.

Handling Errors

As you would expect from a good Python module, `urllib2` detects errors and raises exceptions when they occur. Handling errors gracefully, then, is usually a matter of catching the appropriate exceptions.

Catching Connection Errors

The process of establishing a connection to the remote web server is by far the most common place for errors to occur. Several things could go wrong at that point: the supplied URL might be malformed, the URL might use a protocol that isn't supported, the hostname might be nonexistent, the server could be unreachable, or the server might return an error (such as 404, File Not Found) in response to the request.

Any exception raised during the connection process is either an instance of `urllib2.URLError` or of a subclass. Therefore, if you don't really care what went wrong—just the fact that *something* did—all you have to do is specify the superclass in your `catch` statement, like this:

```

#!/usr/bin/env python
# Obtain Web Page Information With Simple Error Handling - Chapter 6
# error_basic.py
import sys, urllib2

req = urllib2.Request(sys.argv[1])

try:
    fd = urllib2.urlopen(req)
except urllib2.URLError, e:
    print "Error retrieving data:", e
    sys.exit(1)

```

```

print "Retrieved", fd.geturl()
info = fd.info()
for key, value in info.items():
    print "%s = %s" % (key, value)

```

Now instead of getting a traceback, you'll get a brief message. The message might be "HTTP Error 404: Not found" or "Name or service not known," depending on exactly what went wrong.

However, that's not quite the end of the story. HTTP error messages actually are accompanied by a document describing what happened. If you use a browser such as Mozilla, which doesn't replace this document with something else (like Internet Explorer does), you'll doubtless have seen many "404 documents" saying that the page couldn't be found.

The `urllib2` module handles this situation in an interesting way. It raises an instance of `urllib2.HTTPError` (a subclass of `urllib2.URLError`). `HTTPError` exceptions are themselves file-like objects that can be read from! To get the HTTP server's error document, you just use `read()` like any other file. Remember, though, that some errors are not `HTTPErrors`, so you still need to handle `URLError`. Here's an example:

```

#!/usr/bin/env python
# Obtain Web Page Information With Error Document Handling - Chapter 6
# error_doc.py
import sys, urllib2

req = urllib2.Request(sys.argv[1])

try:
    fd = urllib2.urlopen(req)
except urllib2.HTTPError, e:
    print "Error retrieving data:", e
    print "Server error document follows:\n"
    print e.read()
    sys.exit(1)
except urllib2.URLError, e:
    print "Error retrieving data:", e
    sys.exit(2)

print "Retrieved", fd.geturl()
info = fd.info()
for key, value in info.items():
    print "%s = %s" % (key, value)

```

If an instance of `HTTPError` is raised, the example will catch the exception and print out the details. Otherwise, if the exception is an instance of the normal `urllib2.URLError` class, the program will act as before and display the plain `URLError` message. Here's what you can expect its output to look like:

```
$ ./error_doc.py http://httpd.apache.org/nonexistent
Error retrieving data: HTTP Error 404: Not Found
Server error document follows:

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>404 Not Found</title>
</head><body>
<h1>Not Found</h1>
<p>The requested URL /nonexistent wasn't found on this server.</p>
<hr />
<address>Apache/2.0.48-dev (Unix) Server at httpd.apache.org Port 80</address>
</body></html>
$ ./error_doc.py http://httpd.apacheblah.org/
Error retrieving data: <urlopen error (-2, 'Name or service not known')>
```

Catching Data Errors

Handling errors while reading data is trickier. There are two different problems that may occur: A communication error could occur, causing the socket module to raise `socket.error` during a call to `read()`; or the document could be truncated without any communication error.

If a communication error occurs, the low-level error is passed straight up through the system layers and you can handle it exactly like you would any other socket error (see Chapter 2). Detecting a truncated document is a bit more difficult, however.

It's possible for you to receive a shorter-than-intended document without any exception at all. This could happen, for instance, if a program on the server crashed while it was transmitting the document. Such a crash results in the remote socket being closed normally, so your client program would simply see a normal end-of-file condition.

The way to detect this problem is to look for a `Content-Length` header in the server response. If that header is present, you can compare the amount of data you received to the amount the header says you should receive. If the numbers don't match, you know something went wrong.

The `Content-Length` header is not always present. In particular, CGI-generated pages often don't use it. Non-HTTP protocols also may not provide that information.

In these situations, there's no way to detect a truncated file. This isn't a Python-specific problem; all web browsers will have that problem as well.

Here's an example program that checks for both socket errors and truncated documents:

```
#!/usr/bin/env python
# Obtain Web Page With Full Error Handling - Chapter 6
# error_all.py
import sys, urllib2, socket

req = urllib2.Request(sys.argv[1])

try:
    fd = urllib2.urlopen(req)
except urllib2.HTTPError, e:
    print "Error retrieving data:", e
    print "Server error document follows:\n"
    print e.read()
    sys.exit(1)
except urllib2.URLError, e:
    print "Error retrieving data:", e
    sys.exit(2)

print "Retrieved", fd.geturl()

bytesread = 0

while 1:
    try:
        data = fd.read(1024)
    except socket.error, e:
        print "Error reading data:", e
        sys.exit(3)

    if not len(data):
        break
    bytesread += len(data)
    sys.stdout.write(data)

if fd.info().has_key('Content-Length') and \
long(fd.info()['Content-Length']) != long(bytesread):
    print "Expected a document of size %d, but read %d bytes" % \
(long(fd.info()['Content-Length']), bytesread)
    sys.exit(4)
```

Different `sys.exit()` Codes

In the `error_all.py` example, you can observe the program calling `sys.exit()` with codes 1, 2, 3, or 4. These are arbitrarily chosen. A program executing your Python script can learn what value was passed to `sys.exit()` by querying the operating system. Returning different codes will allow other programs to detect whether or not an error occurred, and if so, what happened. By convention, an exit code of zero indicates a successful termination; anything else indicates an error. We don't have another program that calls the script in this instance, but it's a useful practice in case you find yourself in that situation.

Using Non-HTTP Protocols

The `urllib2` module does support non-HTTP protocols. By default, it will support HTTP files on your machine's local hard drive, and FTP, but you can also enable the Gopher handler.

The only difference that's noticeable to your program will be the headers returned by `info()`. In some cases, there may be no headers at all (such as when you use an FTP URL that points to a directory). In others, HTTP headers may be simulated. The file handler, for instance, adds a `Content-Length` header with the size of the file being referenced.

If your program is well written, and accepts lack of headers gracefully, you will not need to make any modifications to let it support non-HTTP protocols. As an example, you can see that the `dump_page.py` program from the beginning of the chapter is perfectly capable of retrieving FTP documents, as shown here:

```
$ ./dump_page.py ftp://ftp.ibiblio.org/README
```

```
Welcome to ftp.ibiblio.org, the public ftp server of ibiblio.org. We  
hope you find what you're looking for.
```

Summary

Python's `urllib2` module provides a convenient interface for obtaining data from a variety of sources and supports HTTP with or without Secure Sockets Layer, FTP, and Gopher by default. Its most basic use is to simply download web pages or files from the Internet.

Some websites require authentication before access is granted. By building your own URL opener and defining a password manager class, you can prompt the user for authentication information when necessary.

With `urllib2`, you can also submit data for forms. Two different methods are used for that: GET and POST. Both are supported by `urllib2`.

Like many Python modules, `urllib2` raises exceptions when errors occur. Several different types of exceptions are possible, and some of them provide additional information in the exception object.

If your program gracefully handles the lack of HTTP-specific headers, it will be adaptable enough to support any non-HTTP protocol that `urllib2` supports.

Parsing HTML and XHTML

HYPertext Markup Language (HTML) and its newer cousin Extensible Hypertext Markup Language (XHTML) are the primary languages of the Web. When you view a web page, chances are HTML or XHTML is used to represent that page.

These markup languages are designed primarily to present information to a web browser. However, sometimes you may find yourself needing to extract information from a web page because there's no better way to obtain that information. Here are some examples of information that you may have to extract from web pages in this way:

- Today's weather forecast for a datebook application
- Television schedules for a video-recording program
- Weather conditions for a severe-storm alerting program
- Stock prices for a financial-management application
- Music information for a CD-labeling program

Extracting information from web pages intended for viewing by humans can be difficult. You should always first try to find a data source more suited for computer processing. Sometimes, you may be able to find comma-separated or XML files, which are easier to handle. (See Chapter 8 for details on XML files.)

HTML is hierarchical, meaning that a parser needs to know the context in which tags occur. This makes parsing using ordinary methods, such as regular expressions, difficult. Things are complicated further by the fact that many HTML pages don't actually adhere to the HTML standard. Python provides a module called `HTMLParser` that's designed to let you parse HTML with greater ease. In this chapter, you'll learn how to extract information from both simple and complex sites. Once the information is extracted, you'll be able to take advantage of it in your own programs.

XHTML represents an interesting hybrid data format. XHTML aims to be compatible with HTML while also being parsable as a true XML document. If you have XHTML files, you'll be able to choose whether to parse them as HTML files

with the techniques shown in this chapter, or as XML files with the techniques shown in Chapter 8. In general, given a choice, you should opt for XML processing of these files, since it will usually be easier and more reliable. When I refer to HTML files in this chapter, please remember that many XHTML files are also valid HTML, and could be used as well.

Alternatives to HTMLParser

`HTMLParser` is one of several choices you have for parsing HTML. Python's standard library ships `htmllib`, which is based on Python's more general SGML Framework. This library also features a formatter interface, which may be useful if you'll be presenting HTML as some other data format.

Interfaces to the `Tidy` library, such as `mxTidy` or `uTidylib`, are also available as third-party add-ons. These interfaces attempt to convert HTML into XML, correcting coding errors in the HTML as they go. You can then use XML parsing techniques as described in Chapter 8 to work with the result. For large, complex HTML or XHTML documents that are generally standards-compliant, this may be an easier route for you. You can find `mxTidy` from www.egenix.com/files/python/mxTidy.html or `uTidylib` from <http://utidylib.sourceforge.net/>.

Understanding Basic HTML Parsing

To implement a parser with the `HTMLParser` module, you'll generally subclass `HTMLParser.HTMLParser` and add functions to handle different types of tags. As an example, let's consider an attempt to pull the title out of the following HTML document:

```
<!-- Basic title parsing example, Chapter 7 - basictitle.html -->
<HTML>
<HEAD>
<TITLE>Document Title</TITLE>
</HEAD>
<BODY>
This is my text.
</BODY>
</HTML>
```

Now here's code that will be able to pick out the title:

```

: /usr/bin/env python
: Basic HTML Title Retriever - Chapter 7 - basictitle.py

:::~ HTMLParser import HTMLParser
:::~ import sys

class TitleParser(HTMLParser):
    def __init__(self):
        self.title = ''
        self.readingtitle = 0
        HTMLParser.__init__(self)

    def handle_starttag(self, tag, attrs):
        if tag == 'title':
            self.readingtitle = 1

    def handle_data(self, data):
        if self.readingtitle:
            # Ordinarily, this is slow and a bad practice, but
            # we can get away with it because a title is usually
            # small and simple.
            self.title += data

    def handle_endtag(self, tag):
        if tag == 'title':
            self.readingtitle = 0

    def gettitle(self):
        return self.title

fd = open(sys.argv[1])
tp = TitleParser()
tp.feed(fd.read())
print "Title is:", tp.gettitle()

```

The program creates a class `TitleParser` that is a descendant of the standard `HTMLParser` class. `HTMLParser`'s `feed()` method will call our `handle_starttag()`, `handle_data()`, and `handle_endtag()` methods as appropriate. The `handle_data()` method simply checks whether or not it's receiving data from a `TITLE` element, and if so, saves it off. Running the program produces this result:

```

$ ./basictitle.py basictitle.html
Title is: Document Title

```

You could use the same sort of principle to extract the text from any other opened and closed tags in your document as well. For instance, to extract data from tables, you may be looking at pulling data from `<TR>` or `<TD>` tags.

Handling Real-World HTML

The preceding code works well for the simple, well-formed document at hand. However, not all HTML is quite that easy—and not all documents calling themselves HTML actually contain valid HTML. Today's web browsers are quite lenient given bad HTML, and as an unfortunate consequence, bad HTML is prolific on the Internet today. Your programs will have to deal with it. This section presents ways of dealing with both aspects of legal HTML and invalid HTML.

Translating Entities

Entities in HTML stand in for regular characters. For instance, the HTML entity `&` represents the ampersand. When displaying or working with HTML code, you'll generally want to convert the HTML entities back into plain text. Here's a modified version of the first example that illustrates the problem:

```
<!-- Entity title parsing example, Chapter 7 - etitle.html -->
<HTML>
<HEAD>
<TITLE>Document Title & Intro</TITLE>
</HEAD>
<BODY>
This is my text.
</BODY>
</HTML>
```

If you run the earlier code against this HTML, you don't get the desired result, as follows:

```
$ ./basictitle.py etitle.html
Title is: Document Title  Intro
```

In fact, the entity has completely disappeared. This is because `HTMLParser` calls `handle_entityref()` when it encounters an entity. Since the code didn't define such a method, it does nothing. Here's an example that considers entities:

```

$ usr/bin/env python
$ HTML Title Retriever With Entity Support - Chapter 7 - etitle.py

from htmlentitydefs import entitydefs
from HTMLParser import HTMLParser
import sys

class TitleParser(HTMLParser):
    def __init__(self):
        self.title = ''
        self.readingtitle = 0
        HTMLParser.__init__(self)

    def handle_starttag(self, tag, attrs):
        if tag == 'title':
            self.readingtitle = 1

    def handle_data(self, data):
        if self.readingtitle:
            self.title += data

    def handle_endtag(self, tag):
        if tag == 'title':
            self.readingtitle = 0

    def handle_entityref(self, name):
        if entitydefs.has_key(name):
            self.handle_data(entitydefs[name])
        else:
            self.handle_data('&' + name + ';')

    def gettitle(self):
        return self.title

fd = open(sys.argv[1])
tp = TitleParser()
tp.feed(fd.read())
print "Title is:", tp.gettitle()

```

Running this program over the example produces the following correct output:

```

$ ./etitle.py etitle.html
$ title is: Document Title & Intro

```

Python conveniently provides mappings from HTML entities in its `htmlentitydefs` class. This program simply uses that code to enable the proper conversion. When an entity is encountered, the code checks to see if it's recognized. If so, the translated value is used. Otherwise, the literal value from the input stream is used. This is because people often forget to use `&` instead of an ampersand in HTML, and end up creating invalid entities.

Translating Character References

Besides entities, HTML files can also contain character references. These contain decimal values of characters and look like `®`. Character references like this are used to embed characters that aren't always printable. For instance, non-English documents may contain character references for certain characters. English documents may contain references to special symbols as well. Translating them is straightforward. Here's an example of a file with a character reference representing the registered trademark symbol:

```
<!-- Character reference title parsing example, Chapter 7 - ctitle.html -->
<HTML>
<HEAD>
<TITLE>Document Title &amp; Intro&#174;</TITLE>
</HEAD>
<BODY>
This is my text.
</BODY>
</HTML>
```

The code to handle that simply adds this function to the last example, as shown here:

```
# excerpt from ctitle.py
def handle_charref(self, name):
    # Validate the name.
    try:
        charnum = int(name)
    except ValueError:
        return

    if charnum < 1 or charnum > 255:
        return

    self.handle_data(chr(charnum))
```

The code simply ignores invalid character references—those that cannot be translated into an integer or are outside the range of characters that get handled. If you run this program with the previous example, you should see a registered trademark sign at the end of the output. If you don't, your terminal may be set up to display characters from other than the Latin-1 character set. That's OK and is nothing to worry about.

Handling Unbalanced Tags

One of the most annoying—and prevalent—problems with HTML code is unbalanced tags. For instance, a web page might have a `<TITLE>` start tag but omit any `</TITLE>` end tag. With HTML, some tags aren't required to be closed: `<P>` and `` are two examples. Therefore, you'll likely need to invent mechanisms for dealing with these problems. XHTML mandates that *all* tags are closed; that's why you may sometimes see XHTML code such as `
`—the trailing slash is a shortcut for the code `
</br>`. Here's an example file with unbalanced tags:

Tidy Can Help

In the “Alternatives to HTMLParser” sidebar earlier in this chapter, I mentioned `mxTidy` and `uTidylib`. These libraries can be used to attempt an automated fix of poorly written HTML code. You may still need to manually correct things in some cases, but these interfaces can make life easier for you in many instances.

```
-- Unbalanced tags parsing example, Chapter 7 - utitle.html -->
<HTML>
<HEAD>
<TITLE>Document Title &amp; Intro<#174;
<HEAD>
<BODY>
<is is my text.
<L>
<I>First List Item
<I>Second List Item</LI>
<I>Third List Item
<BODY>
<HTML>
```

You'll notice that the closing </TITLE> tag is missing. Also, two tags aren't closed—in fact, their parent tag isn't closed either. This code handles all of these situations:

```
#!/usr/bin/env python
# HTML Extended Parser - Chapter 7 - utitle.py

from htmlentitydefs import entitydefs
from HTMLParser import HTMLParser
import sys, re

class TitleParser(HTMLParser):
    def __init__(self):
        # The taglevels list keeps track of where we are in the tag
        # hierarchy.
        self.taglevels = []
        self.handledtags = ['title', 'ul', 'li']
        self.processing = None
        HTMLParser.__init__(self)

    def handle_starttag(self, tag, attrs):
        """Called whenever a start tag is encountered."""
        if len(self.taglevels) and self.taglevels[-1] == tag:
            # Processing a previous version of this tag. Close it out
            # and then start anew on this one.
            self.handle_endtag(tag)

        # Note that we're now processing this tag
        self.taglevels.append(tag)

        if tag in self.handledtags:
            # Only bother saving off the data if it's a tag we handle.
            self.data = ''
            self.processing = tag
            if tag == 'ul':
                print "List started."
```

```

def handle_data(self, data):
    """This function simply records incoming data if we are presently
    inside a handled tag."""
    if self.processing:
        # This could be slow for large files. For this example,
        # it's a simple way to save off data.
        self.data += data

def handle_endtag(self, tag):
    if not tag in self.taglevels:
        # We didn't have a start tag for this anyway. Just ignore.
        return

    while len(self.taglevels):
        # Obtain the last tag on the list and remove it
        starttag = self.taglevels.pop()

        # Finish processing it.
        if starttag in self.handledtags:
            self.finishprocessing(starttag)

        # If it's our tag, stop now.
        if starttag == tag:
            break

def cleanse(self):
    """Removes extra whitespace from the document."""
    self.data = re.sub('\s+', ' ', self.data)

def finishprocessing(self, tag):
    self.cleanse()
    if tag == 'title' and tag == self.processing:
        print "Document Title:", self.data
    elif tag == 'ul':
        print "List ended."
    elif tag == 'li' and tag == self.processing:
        print "List item:", self.data

    self.processing = None

```

```

def handle_entityref(self, name):
    if entitydefs.has_key(name):
        self.handle_data(entitydefs[name])
    else:
        self.handle_data('&' + name + ';')

def handle_charref(self, name):
    # Validate the name.
    try:
        charnum = int(name)
    except ValueError:
        return

    if charnum < 1 or charnum > 255:
        return

    self.handle_data(chr(charnum))

def gettitle(self):
    return self.title

fd = open(sys.argv[1])
tp = TitleParser()
tp.feed(fd.read())

```

If you run the program, you'll get this output:

```

$ ./utitle.py utitle.html
Document Title: Document Title & Intro®
List started.
List item: First List Item
List item: Second List Item
List item: Third List Item
List ended.

```

That's the correct output. Let's look at how the program managed to produce it.

In the `handle_starttag()` function, the system notes in `self.taglevels` whenever a new start tag is encountered. If the tag is one of the three that the program processes, it also sets the `self.processing` flag to tell the system to begin recording data. This flag acts similar to the previous title-processing programs.

In `handle_endtag()`, the program first checks to see if it had seen a start tag for the end tag in question. If not, it just ignores the tag—it was probably a typo by the HTML author. If the tag had been seen, it would find the most recent occurrence.

It does that by starting at the end of the `self.taglevels` list and working its way backward. Along the way, if any of the three interesting tags are encountered, `self.finishprocessing()` is called for them.

The `finishprocessing()` function cleans up the white space in the data string and then prints out the appropriate message. The `tag == self.processing` checks are there to ensure that the same data isn't used twice. For instance, if interesting tags were nested three deep, this code would have saved the data from only the most recent.

That illustrates one problem with this code: If you wanted to capture data between the `` and the first `` (or even anywhere inside the `` but outside an ``), it would be lost as soon as the `` begins. That's because `handle_starttag()` overwrites `self.data` whenever it sees a start tag for an interesting tag. This isn't often a problem, however.

A Working Example

`HTMLParser` can do much more than just pick out titles and lists from a document. It can also be used to help parse tables, pick out interesting bits from huge web pages, and other tasks.

To illustrate using `HTMLParser` for tasks, here's an example that connects to a real website (www.wunderground.com) and downloads an HTML page containing the current conditions and forecast for your zip code. The program then parses this page, finding only the interesting parts, and then renders the table in a nice way on the screen. I'll show you this code in pieces, explaining each chunk along the way. Here's the first piece:

```
= /usr/bin/env python
= weather Parser - Chapter 7 - weather.py

from htmlentitydefs import entitydefs
from HTMLParser import HTMLParser
import sys, re, urllib2

= Declare a list of interesting tables.
interesting = ['Day Forecast for ZIP']

class WeatherParser(HTMLParser):
    """Class to parse weather data from www.wunderground.com."""
    def __init__(self):
        # Storage for parse tree
        self.taglevels = []
```

```

# List of tags that are interesting
self.handledtags = ['title', 'table', 'tr', 'td', 'th']

# Set to the interesting tag currently being processed
self.processing = None

# True if currently processing an interesting table
self.interestingtable = 0

# If processing an interesting table, holds cells in current row
self.row = []

# Initialize base class.
HTMLParser.__init__(self)

```

So far, this code is fairly standard. It mainly sets up some variables for later use. Note the list of interesting tables; it's used to help identify which tables to display. If you wanted to display more than one, you could add more names there, as follows:

```

def handle_starttag(self, tag, attrs):
    """Called by base class to handle start tags."""
    if len(self.taglevels) and self.taglevels[-1] == tag:
        # Processing a previous version of this tag. Close it out
        # and then start anew on this one.
        self.handle_endtag(tag)
    self.taglevels.append(tag)
    if tag == 'br':
        # Add a special newline token to the stream.
        self.handle_data("<NEWLINE>")
    elif tag in self.handledtags:
        # Start processing an interesting tag.
        self.data = ''
        self.processing = tag

```

This code is called when a start tag is encountered. Like before, we handle only interesting tags. In this case, a break tag
 is notable, and we add a special token to the data stream for later processing. The program could have just used
 instead of <NEWLINE>, but that's more likely to occur as literal text in the output (for instance, in a HTML page describing how to use HTML).

```

def handle_data(self, data):
    """Called by both HTMLParser and methods in WeatherParser to handle
    plain data."""
    if self.processing:
        self.data += data

def handle_endtag(self, tag):
    """Handle a closing tag."""
    if tag in self.taglevels:
        # We didn't have a start tag for this anyway. Just ignore.
        return

    while len(self.taglevels):
        # Obtain the last tag on the list and remove it
        starttag = self.taglevels.pop()

        # Finish processing it
        if starttag in self.handledtags:
            # If it's interesting, do something with it.
            self.finishprocessing(starttag)
        if starttag == tag:
            # Found the tag; stop processing here.
            break

```

The `handle_data()` and `handle_endtag()` methods are similar to what you've already seen in the title-parsing examples. But the next functions are different.

```

def cleanse(self):
    """Adjusts data stream to convert whitespace."""
    # \xa0 is the non-breaking space (&nbsp; in HTML)
    self.data = re.sub('(\s|\xa0)+', ' ', self.data)
    self.data = self.data.replace('<NEWLINE>', "\n").strip()

```

The `cleanse()` method removes the extra white space in the document as before. However, this time it replaces `<NEWLINE>` with the actual newline character. The special string `<NEWLINE>` is used because otherwise the program would remove an embedded "`\n`" just like any other white space, as shown here:

```
def finishprocessing(self, tag):
    """Called by handle_endtag() to handle an interesting end tag."""
    global interesting
    self.cleanser()
    if tag == 'title' and tag == self.processing:
        # Print out the page's title.
        print " *** %s ***" % self.data
    elif (tag == 'td' or tag == 'th') and tag == self.processing:
        # Got a cell in a table.
        if not self.interestingtable:
            # If we're not already in an interesting table, see if
            # this cell makes the table interesting.
            for item in interesting:
                if re.search(item, self.data, re.I):
                    # Yep, found an interesting table. Note that, then
                    # remove it from the interesting list, print
                    # out a heading, and stop looking at the list.
                    self.interestingtable = 1
                    interesting = [x for x in interesting if x != item]
                    print "\n *** %s\n" % self.data.strip()
                    break
        else:
            # Already in an interesting table; just add this cell to
            # the current row.
            self.row.append(self.data)
    elif tag == 'tr' and self.interestingtable:
        # Print out an interesting row.
        self.writerow()
        self.row = []
    elif tag == 'table':
        # End of table: note that system is no longer processing
        # an interesting table.
        self.interestingtable = 0

    self.processing = None
```

```

def writerow(self):
    """Formats a row for on-screen display."""

    cells = len(self.row)
    if cells < 2:
        # If there are no cells, the row is empty; display nothing.
        # If there is one cell, wunderground.com uses it as a header.
        # We don't want it, so again, display nothing.
        return
    if cells > 2:
        # If it's a table with lots of cells, give each cell
        # the same amount of space, leaving room for a space between
        # cells.
        width = (78 - cells) / cells
        maxwidth = width
    else:
        # If it's a table with two cells, make the left one narrow
        # and the right one wide.
        width = 20
        maxwidth = 58

    # Continue looping while at least one cell has a line of data to print
    while [x for x in self.row if x != '']:
        # Process each cell in the row.
        for i in range(len(self.row)):
            thisline = self.row[i]
            if thisline.find("\n") != -1:
                # If it has multiple lines, we want only the first;
                # save it in thisline, and shove the rest back into
                # the list for processing later.
                (thisline, self.row[i]) = self.row[i].split("\n", 1)
            else:
                # Just one line -- we've already got it in thisline,
                # so put the empty string in the list for later.
                self.row[i] = ''
            thisline = thisline.strip()
            sys.stdout.write("%-*.*s " % (width, maxwidth, thisline))
            sys.stdout.write("\n")

```

Together, the `finishprocessing()` and `writerow()` functions form the heart of the table-processing code. Like before, `finishprocessing()` is called when there's an interesting end tag. However, this time the function does more. If a table is being processed, it looks at the cell contents to determine if the table is an interesting one. If it is, it sets the flag. If the flag is set, then cell contents are saved off or printed. The `writerow()` function does the actual printing, as follows:

```
def handle_charref(self, name):
    """Process character references if possible."""
    # Validate the name.
    try:
        charnum = int(name)
    except ValueError:
        return

    if charnum < 1 or charnum > 255:
        return

    self.handle_data(chr(charnum))

    sys.stdout.write("Enter ZIP code: ")
    zip = sys.stdin.readline().strip()
    url = "http://www.wunderground.com/cgi-bin/findweather/getForecast?query=" + \
          zip

    req = urllib2.Request(url)
    fd = urllib2.urlopen(req)

    parser = WeatherParser()
    data = fd.read()
    data = re.sub(' ([^ =]+)=[^ ="]+=[^ ="]+', ' \\\1=', data)
    data = re.sub('(?s)<!!--.*?-->', '', data)
    parser.feed(data)
```

The remainder of the program is mostly standard. There are two interesting things to note near the bottom of the program, namely, the calls to `re.sub()`. The input document in this instance has some violations of HTML that are so flagrant that they prevent `HTMLParser` from being able to process the document. These regular expressions clear up those problems before the document is sent to `HTMLParser`.

Try running the program. Here's an example:

```

$ ./weather.py
Enter ZIP code: 77002
*** Weather Underground: Houston, Texas Forecast **

*** 5 Day Forecast for ZIP Code 77002

Thu      Fri      Sat      Sun      Mon
66° | 45°   69° | 58°   76° | 68°   80° | 70°   77° | 54°
Rain Showers  Partly Cloudy  Chance of T-st  Chance of T-st  Chance of T-st
Detail      Detail      Detail      Detail      Detail

```

Much of this program is documented with comments. However, there are a few other things to highlight. Pages from `www.wunderground.com` are big and complex and contain many things that aren't pertinent for the purposes of this example. Their page layout tends to put content of a particular type in its own table, so all the program has to do is figure out which tables are interesting. It does this by looking for certain strings in the table; if found, the table is processed.

Also, one final caution: Website maintainers may change their layout at any time. If `www.wunderground.com` has changed their site between the time this example was written and the time you try to run it, the program may not work. Some websites are starting to make it easier to pull down information using formats such as XML (and XHTML). If the site you're interested in has that capability, you should also read Chapter 8.

Summary

HTML and XHTML are the most common ways of representing documents on the Web. XHTML is a newer format that's designed to be both valid HTML and a valid XML document and can be parsed by both of the techniques described in this chapter and those described in Chapter 8.

Parsing HTML can be difficult; other ways to obtain data should be attempted first. Tools such as `HTMLParser` can help make the job easier. However, there's a lot of useful information available in HTML format, and sometimes there's no other way to easily obtain it.

HTML is hierarchical and often contains coding errors. Your program will often need to cope with coding errors. You'll also have to convert entity and character references.

Processing large and complex web pages can require a large and complex program. Finally, I provided an example that obtains the weather forecast for a given location.

CHAPTER 8

XML and XML-RPC

THE INTERNET HAS long been used to transfer documents from one machine to another. These documents may be anything from plain text to HTML, Microsoft Word, or PDF files. It's difficult for a machine to extract data from documents written in a human language. Sometimes it's useful to be able to extract information from documents and use it in new ways. Other times, it may simply be useful to keep the content the same, but adjust the layout—perhaps to permit the incorporation of a document into a larger compilation. For example, consider a large family-history book. You may wish to generate a list of all known addresses of people mentioned in the book, but they're not listed in any standard way. Your program may have trouble retrieving two addresses from text such as the following:

Jeff Washington grew up in Chicago, IL at 132 S. Lake Shore Dr.
After graduating from high school, he met Mary Davis of
100 E. Main St, Dallas, TX.

To help computers be able to process documents better, various ways of marking text have been invented. In this chapter and the next, you'll be concerned with a derivative of the Standard Generalized Markup Language (SGML), which describes a standard way embedding tags in a document. An SGML tag is a special string within a document that carries information for the parser. SGML tags are surrounded by angle brackets and look something like this: <para>. SGML doesn't describe what those tags are or what they mean. Normally, you'll have a *document type definition* (DTD) that describes the structure of the document and the set of available tags. You'll also have processors that do various things with your document. They may, for instance, transform a document into an elegant PDF document that's ready to print or extract certain information from it for storage in a database. Document validators are also available; they ensure that your documents comply with your DTD. One popular system based on SGML is known as DocBook and represents a framework for developing technical documentation.

Here's a hypothetical excerpt from an SGML document that could represent the preceding example text:

```

<para>
  <name id="jeff"><firstname>Jeff</firstname> <surname>Washington</surname>
  </name>grew up at <address nameid="jeff">
    <city>Chicago</city>
    <state>IL</state>
    <street>132 S. Lake Shore Dr.</street>
  </address>. After graduating from high school, he met
  <name id="mary"><firstname>Mary</firstname>
  <surname>Davis</surname></name>
  of <address nameid="mary">
    <street>300 E. Main St.</street>
    <city>Dallas</city>
    <state>TX</state>
  </address>.
</para>

```

A document processor could easily discover the names of all people mentioned. It could also link addresses to particular names and discover the list of addresses. A document processor may produce a result like this:

Davis, Mary
 300 E. Main St.
 Dallas, TX

Washington, Jeff
 132 S. Lake Shore Dr.
 Chicago, IL

Alternatively, your processor may generate text that's suitable for printing. Perhaps you would obtain a result like this:

Jeff Washington grew up at 132 S. Lake Shore Dr.; Chicago, IL.
 After graduating from high school, he met Mary Davis of
 300 E. Main St.; Dallas, TX.

By slightly adjusting your processor, you could cause all addresses in the document to be rendered differently. If your processor knows about state abbreviations, you could easily obtain a result like this:

Jeff Washington grew up at 132 S. Lake Shore Dr., Chicago
 (Illinois). After graduating from high school, he met
 Mary Davis of 300 E. Main St., Dallas (Texas).

You can do all of this because your processor is able to pick out well-defined parts of the document and know that you're always dealing with a particular type of data.

The first example, which generated a name and address list, may discard everything that wasn't of interest for its particular purpose. The other examples rendered all the text, but used different rules to do so.

The Extensible Markup Language (XML) is a derivative of SGML that's designed to be simpler to use and easier to parse. Compared with SGML, XML is often easier to use in your programs and is easier to write. Existing SGML tools are generally compatible with XML as well.

XML was originally designed to be a flexible language to represent the content and structure of documents. However, XML has spread far beyond that. XML can also be used to represent many types of structured data and for transmitting data across a network.

This chapter discusses both XML as a document language and XML-RPC, which is a network transport protocol that uses XML to represent data. If you're only interested in XML-RPC, you can safely skip the following sections on XML documents; an understanding of XML isn't required to use XML-RPC. Likewise, XML-RPC knowledge isn't required to understand XML. This chapter describes XML-RPC from a client perspective. If you're interested in writing XML-RPC servers, consult Chapter 17.

Understanding XML Documents

XML documents are hierarchical and structured. For instance, this could be a well-formed XML document:

```
>xml version="1.0" encoding="ISO-8859-1"?>
!-- Sample XML Document - Chapter 8 - sample.xml -->
<book>
  <title>Sample XML Thing</title>
  <author>
    <name><first>Benjamin</first> <last>Smith</last></name>
    <affiliation>Springy Widgets, Inc.</affiliation>
  </author>

  <chapter number="1">
    <title>First Chapter</title>
    <para>
      I think widgets are great. You should buy lots of them from
      <company>Springy Widgets, Inc</company>.
    </para>
  </chapter>
</book>
```

You can see the hierarchy; most of the document is part of the book. The book has a title, as does the chapter. There's some information about the author as well. You know that the name refers to the author because it's within the `<author>` section.

Unlike HTML's, the XML specification doesn't mention what constitutes a valid tag or which tags you must use. As a document author, you're free to invent your own tags and use them for whatever purpose you like.

However, many standards are built atop XML, and those do define valid tags. Two examples include XHTML (used for representing data on the Web) and DocBook (used for publishing technical material). In fact, by using a DTD, you can specify which tags are valid for the purposes of verifying the correctness of a document. There are tools that can validate your document against its DTD. However, these tools are outside the scope of this chapter.

Usually, when you work with XML documents, you'll use a premade XML parsing library. These libraries generally use one of two ways to represent the XML document to you: *trees* and *events*. An event-based parser will scan a document and inform your program when interesting things are encountered. The `HTMLParser` examples in Chapter 7 are an example of an event-based parser. For an event-based parser, you write hooks that define what to do when certain things are encountered in a document.

A tree-based parser, on the other hand, scans the complete document for you and generates nested data structures that represent the document. For a tree-based parser, you scan the result produced by the parser after it's done, picking out information that you need. Another important benefit of a tree-based parser is that you can often alter the in-memory data structures and then request the system to write out a modified XML document to disk.

I often believe that a tree-based parser is easier to use than an event-based parser. Being able to examine a prebuilt tree can be a time-saver when programming. However, event-based parsers are more flexible and can be optimized to a greater degree. They can also be used with documents that are too large to fit in memory. That said, you could accomplish most any task with either model.

Python provides implementations of both. Its Simple API for XML (SAX) modules implement an event-based parser, and its Document Object Model (DOM) modules implement a tree-based parser. Both SAX and DOM are available in other object-oriented languages as well. In this book, DOM is covered because it tends to be easier and more versatile for most tasks.

Using DOM

The primary blessing of DOM is that it presents you with a full XML document tree to work with, though this can be a problem with very large XML documents.

Thus, you can use commands that effectively say things like “give me a list of all paragraph nodes anywhere below the current one” or “show me all the nodes directly beneath the current one.” This is a powerful feature, and can save you a lot of coding grunt work.

Tools for Searching XML

If you find yourself performing lots of queries like the previous one, you may find the XPath support in the third-party libxml2 library to be useful. You may download that library from www.xmlsoft.org/python.html.

You can find an example XML file earlier in this chapter. Let’s look at a representation of what this looks like to DOM. First, as you saw in the example, XML is hierarchical. DOM represents this hierarchy using a tree. Each item on the tree is a node. A node may have children. For instance, the `<book>` tag in the example has children such as `<title>` and `<author>`.

In the following example, each line represents an object. Indented lines are children of their parent objects. The names (Element, Text, and so forth) indicate the type of the DOM object shown, as shown here:

```
Document
  Comment
  Element, tag: book
    Text
    Element, tag: title
      Text
    Text
    Element, tag: author
      Text
      Element, tag: name
        Element, tag: first
          Text
        Text
        Element, tag: last
          Text
        Text
      Text
      Element, tag: affiliation
        Text
      Text
    Text
```

```
Element, tag: chapter
    Text
    Element, tag: title
        Text
    Text
    Element, tag: para
        Text
    Element, tag: company
        Text
    Text
    Text
    Text
```

You can see the structure of the sample XML document displayed here. Element objects represent the parts of a document between a pair of tags such as <book>. Text objects represent the actual text of the document; you see them frequently because the parser sometimes creates Text objects even if they represent only white space. Comment objects represent comments. Finally, Document objects occur only once in a given tree and represent the entire document.

This listing was actually generated by a Python program. Here's the code for it:

```
#!/usr/bin/env python
# Tree Generation with DOM - Chapter 8 - domtree.py

from xml.dom import minidom, Node

def scanNode(node, level = 0):
    msg = node.__class__.__name__
    if node.nodeType == Node.ELEMENT_NODE:
        msg += ", tag: " + node.tagName
    print " " * level * 4, msg
    if node.childNodes:
        for child in node.childNodes:
            scanNode(child, level + 1)

doc = minidom.parse('sample.xml')
scanNode(doc)
```

This program is fairly simple. It starts with a call to `minidom.parse()`. The `parse()` function loads and parses the XML document and returns the topmost node—the `Document` object. From that object, you can descend the tree and find all the other objects. That's exactly what `scanNode()` does.

The `scanNode()` function begins by generating the message to display for the current node. Then it checks to see if the current node is capable of having child nodes (generally, only `Document` and `Element` nodes have children). If so, it obtains the list of children from `node.childNodes` and recursively calls itself for each one.

Full Parsing with Document Object Model

Let's consider a more thorough example. The previous example displayed information about the document structure only. This example works with the preceding `sample.xml` file and generates a plain text representation of it. I'll show you the example in chunks, and describe each chunk as we go. Here's the first piece:

```
#!/usr/bin/env python
# Parsing Sample with DOM - Chapter 8 - domparsesample.py
# This program requires Python 2.3 for the textwrap module

from xml.dom import minidom, Node
import re, textwrap

class SampleScanner:
    def __init__(self, doc):
        for child in doc.childNodes:
            if child.nodeType == Node.ELEMENT_NODE and \
                child.tagName == 'book':
                self.handleBook(child)
```

This is the start of the implementation of the `SampleScanner` class. This wouldn't really have to be a class—we're not extending any other class—but it's useful to group things together in this way.

At the end of the program, you'll see the code instantiate a `SampleScanner` object, passing it the `Document` node as an argument. That's received as `doc` here. This `__init__()` method scans `doc` looking for the `<book>` tag. It ignores anything else it may encounter, such as comments or other tags.

```

def gettext(self, nodelist):
    """Given a list of one or more nodes, recursively finds all text
    nodes in that list (or children of nodes in that list), concatenates
    them, removes duplicate spaces, and returns the result."""
    retlist = []
    for node in nodelist:
        if node.nodeType == Node.TEXT_NODE:
            retlist.append(node.wholeText)
        elif node.hasChildNodes:
            retlist.append(self.gettext(node.childNodes))

    return re.sub('\s+', ' ', ''.join(retlist))

```

This function scans nodes looking for text and builds a list of strings holding that text. It then combines that list into a single string and replaces any instance of one or more white-space characters with a single space.

```

def handleBook(self, node):
    """Process the book tag. Look for title, author, then chapters."""

    for child in node.childNodes:
        if child.nodeType != Node.ELEMENT_NODE:
            continue
        if child.tagName == 'title':
            print "Book title is:", self.gettext(child.childNodes)
        if child.tagName == 'author':
            self.handleAuthor(child)
        if child.tagName == 'chapter':
            self.handleChapter(child)

```

Here the program processes the direct children of <book>. If it finds a title, it will just print it out. For an author or a chapter, it will call handler functions for those particular tags. Anything else is ignored.

```

def handleAuthor(self, node):
    for child in node.childNodes:
        if child.nodeType != Node.ELEMENT_NODE:
            continue
        if child.tagName == 'name':
            self.handleAuthorName(child)
        elif child.tagName == 'affiliation':
            print "Author affiliation:", self.gettext([child])

```

Recall that an `<author>` tag in `sample.xml` could contain a `<name>` or an `<affiliation>` child. An affiliation is a simple string, but a name can have some child nodes, so it's been split off into a separate function, as follows:

```
def handleAuthorName(self, node):
    surname = selfgettext(node.getElementsByTagName("last"))
    givenname = selfgettext(node.getElementsByTagName("first"))
    print "Author Name: %s, %s" % (surname, givenname)
```

This is the name printing function. It takes a node representing a name and simply gets the text that can be found in the `<first>` and `<last>` tags. Then it prints out the result. You could easily adjust the program to print the name in the opposite order (surname last) by adjusting this one simple format string for `print`, as follows:

```
def handleChapter(self, node):
    print " *** Start of Chapter %s: %s" % \
        (node.getAttribute('number'),
         selfgettext(node.getElementsByTagName('title')))
    for child in node.childNodes:
        if child.nodeType != Node.ELEMENT_NODE:
            continue
        if child.tagName == 'para':
            self.handlePara(child)

def handlePara(self, node):
    partext = selfgettext([node])
    partext = textwrap.fill(partext)
    print partext
    print
```

These two functions handle the text of a chapter. The first function, `handleChapter()`, simply displays a chapter number and title, then handles all the `<para>` child nodes. You'll recall from `sample.xml` that a chapter was defined like this: `<chapter number="1"><title>First Chapter</title>...</chapter>`. That's why `getAttribute()` is used to obtain the number, but `getElementsByTagName()` is used to obtain the title, as shown here:

```
:doc = minidom.parse('sample.xml')
:sampleScanner(doc)
```

These two lines conclude the program. They simply parse the XML file into a DOM tree and pass that tree to an instance of `SampleScanner`.

If you run this program, you'll find output like this:

```
$ ./domparsesample.py
Book title is: Sample XML Thing
Author Name: Smith, Benjamin
Author affiliation: Springy Widgets, Inc.
*** Start of Chapter 1: First Chapter
I think widgets are great. You should buy lots of them from Springy
Widgets, Inc.
```

You can see that the program found the book's title, the author's name (and was even able to reverse the surnames), affiliation, chapter title, and text. Also, even though both `<book>` and `<chapter>` used `<title>` for their respective titles, the parser was able to determine from the context which title was which. It did this by carefully descending the tree.

Using SimpleAPI for XML for This Task

If you prefer using SAX, you could implement a similar parser with it. Since we are generally iterating over the document, directly descending the tree as we go, it could also be easy to implement with SAX.

Generating Documents with DOM

The examples thus far have demonstrated using the DOM library to generate a tree of objects, and then working with that tree. The library also can work in reverse: You can generate a tree of objects, and the library can write it out to an XML document. You can also take a tree made from an existing document, modify it, and write out the result. Here's a program that generates a parse tree corresponding to the sample document, then prints out the XML for it, as follows:

```
#!/usr/bin/env python
# Generating XML with DOM - Chapter 8 - domgensample.py

from xml.dom import minidom, Node

doc = minidom.Document()

doc.appendChild(doc.createComment("Sample XML Document - Chapter 8"))
```

= Generate the book

```
book = doc.createElement('book')
doc.appendChild(book)

# The title

title = doc.createElement('title')
title.appendChild(doc.createTextNode('Sample XML Thing'))
book.appendChild(title)
```

The author section

```
author = doc.createElement('author')
book.appendChild(author)
name = doc.createElement('name')
author.appendChild(name)
firstname = doc.createElement('first')
name.appendChild(firstname)
firstname.appendChild(doc.createTextNode('Benjamin'))
name.appendChild(doc.createTextNode(' '))
lastname = doc.createElement('last')
name.appendChild(lastname)
lastname.appendChild(doc.createTextNode('Smith'))

affiliation = doc.createElement('affiliation')
author.appendChild(affiliation)
affiliation.appendChild(doc.createTextNode('Springy Widgets, Inc.'))
```

The chapter

```
chapter = doc.createElement('chapter')
book.appendChild(chapter)
chapter.setAttribute('number', '1')
title = doc.createElement('title')
chapter.appendChild(title)
title.appendChild(doc.createTextNode('First Chapter'))

para = doc.createElement('para')
chapter.appendChild(para)
para.appendChild(doc.createTextNode("I think widgets are great. You" +
    " should buy lots of them from "))
```

```

company = doc.createElement('company')
para.appendChild(company)
company.appendChild(doc.createTextNode('Springy Widgets, Inc'))

para.appendChild(doc.createTextNode('.'))

print doc.toprettyxml(indent = '    ')

```

Let's step through this code. It starts off by creating the `minidom.Document` object. All other objects in the tree will be children of this object.

New nodes are created by calling a `create` method on the document object. But a created node doesn't actually participate in the document until it's added; the `appendChild()` method adds a node to the tree. At the top of the example, you can see both actions happening in a single step as a comment is added to the document.

Note that the system doesn't care when you append a child; you can do that as soon as the child is created, or you can wait and do it after you're completely finished processing the child.

The code generates the following XML. While it differs slightly from the sample document, functionally, it's almost identical:

```

<?xml version="1.0" ?>
<!--Sample XML Document - Chapter 8-->
<book>
    <title>
        Sample XML Thing
    </title>
    <author>
        <name>
            <first>
                Benjamin
            </first>

            <last>
                Smith
            </last>
        </name>
        <affiliation>
            Springy Widgets, Inc.
        </affiliation>
    </author>

```

```

<chapter number="1">
    <title>
        First Chapter
    </title>
    <para>
        I think widgets are great. You should buy lots of them from
        <company>
            Springy Widgets, Inc
        </company>
    .
    </para>
</chapter>
</book>

```

If you replace the call to `toprettyxml()` and the end of the document with a call to `toxml()` (with no arguments), you'll see instead an XML document generated using only a single large line for the entire document. This will provide the truest representation of the DOM tree (without extra white space), but in this case, it's almost impossible for a human to edit. It will likely only be useful to other XML parsers! That's because, while generating the DOM tree, the program didn't add explicit white space similar to what existed before.

TIP If you're using the SAX API instead of DOM, you can still generate XML. Python provides `xml.sax.saxutils.XMLGenerator`, a class to help you do just that.

Document Object Model Type Reference

This section presents some quick reference material for your use as you develop programs using DOM. Node type constants are available from `xml.dom.Node` and classes are available from your DOM implementation such as `xml.dom.minidom`. Not all types will be implemented in Python DOM classes.

Table 8-1. Node Types

Type Constant	Number	Class	Description
ELEMENT_NODE	1	Element	Used for tags in the document
ATTRIBUTE_NODE	2	Attr	Holds tag attributes
TEXT_NODE	3	Text	Used for regular text in the document
CDATA_SECTION_NODE	4	CDATASection	Holds CDATA (literal text) data
ENTITY_REFERENCE_NODE	5		A reference to an unhandled entity (one that isn't a built-in part of XML, such as &)
ENTITY_NODE	6	Entity	Definition of an XML entity
PROCESSING_INSTRUCTION_NODE	7	ProcessingInstruction	Processor-specific information
COMMENT_NODE	8	Comment	Holds text of a comment in an XML document
DOCUMENT_NODE	9	Document	The top-level node for an XML document
DOCUMENT_TYPE_NODE	10	DocumentType	Holds the type of the document
DOCUMENT_FRAGMENT_NODE	11	DocumentFragment	The top-level node of a partial document tree (used for parsing partial documents or non-well-formed documents)
NOTATION_NODE	12	Notation	Holds a DTD notation definition

Using XML-RPC

There are a number of ways to gather information from your local machine. Python comes with several modules to help with that, in fact. For instance, the `pwd` module is used on UNIX systems to gather information about the user accounts on a system. You can make simple function calls and receive information ready to use.

Traditionally, gathering information or communicating over a network is more difficult. You might have to format your data for transmission over the network, send the request, wait for and then handle the response. With a library or module on your local machine, you often just call a function.

There have been various efforts to blur the line between local and network services. One of the most popular on UNIX systems is Sun's Remote Procedure Call (RPC). However, RPC is tied very heavily to C and is generally considered outdated today. Java programmers have used Remote Method Invocation (RMI), but it, too is heavily tied to a particular language. XML-RPC is an alternative that's designed to work between different programming languages. XML-RPC derives its name from the fact that XML is used to represent the data being transmitted between machines, although most XML-RPC interfaces don't require you to work with XML directly. XML-RPC is gaining popularity and is already being used for many different services on the Internet.

Alternatives to XML-RPC

XML-RPC isn't the only RPC mechanism available to Python programmers. One "homegrown" approach can be implemented using the standard `pickle` module, which translates Python data structures into a sequence of bytes. This approach doesn't free you from the need to handle lower-level network issues, and it's tied to Python.

The main XML-RPC competitor is the Simple Object Access Protocol (SOAP). Despite its name, SOAP is generally considered to be much more complex than XML-RPC. Python users can use SOAP via the Zolera SOAP Infrastructure (ZSI), which is available from <http://pywebsvcs.sourceforge.net/zsi.html>.

The Common Object Request Broker Architecture (CORBA) is designed to be a way to manage distributed object-oriented programming. Python implementations such as Fnorb (www.fnorb.org) and omniORBpy (www.uk.research.att.com/omniORB/omniORBpy/) let you use CORBA in your Python programs.

Using XML-RPC is simple in Python thanks to the excellent `xmlrpclib` module. Let's take a quick look at an example XML-RPC client:

```
#!/usr/bin/env python
# XML-RPC Basic Client - Chapter 8 - xmlrpcbasic.py

import xmlrpclib
url = 'http://www.oreillynet.com/meerkat/xml-rpc/server.php'
s = xmlrpclib.ServerProxy(url)
catdata = s.meerkat.getCategories()
cattitles = [item['title'] for item in catdata]
cattitles.sort()
for item in cattitles:
    print item
```

This connects to O'Reilly's Meerkat service (see www.oreillynet.com/meerkat/) and retrieves a list of available categories. The XML-RPC server returns that list as a list of structures. Python's `xmlrpclib` translates that into a list of dictionaries; each hash has two keys (`id` and `title`). The program then processes it like any other Python data structure, and prints out a sorted list of titles.

XML-RPC Introspection

Many, but not all, XML-RPC servers support introspection. This provides a way for you to discover what XML-RPC methods are available on any given server. All the introspection queries also use standard XML-RPC calls. This program illustrates the XML-RPC introspection API:

```
#!/usr/bin/env python
# XML-RPC Introspection Client - Chapter 8 - xmlrpcli.py

import xmlrpclib, sys

url = 'http://www.oreillynet.com/meerkat/xml-rpc/server.php'
s = xmlrpclib.ServerProxy(url)

print "Gathering available methods..."
methods = s.system.listMethods()
```

```

while 1:
    print "\n\nAvailable Methods:"
    for i in range(len(methods)):
        print "%2d: %s" % (i + 1, methods[i])
    selection = raw_input("Select one (q to quit): ")
    if selection == 'q':

        break
    item = int(selection) - 1
    print "\n*****"
    print "Details for %s\n" % methods[item]

    for sig in s.system.methodSignature(methods[item]):
        print "Args: %s; Returns: %s" % \
              (", ".join(sig[1:]), sig[0])
    print "Help:", s.system.methodHelp(methods[item])

```

This program starts with a call to `system.listMethods()`. That procedure takes no arguments and returns a list of methods supported on the server. Armed with that information, the client asks the user to choose an item for a more detailed inquiry.

When a method is chosen, the client uses `system.methodSignature()` to obtain the XML-RPC signatures for the method. A signature indicates the number and type of arguments an XML-RPC method takes, and the type of value it returns. XML-RPC supports overloading on the server side (meaning that a method may do different things when called with different types of arguments), so more than one signature may be present. The first item in each signature indicates the method's return type, while the remaining items indicate the argument types. Although this program doesn't check for it, servers may not always support `system.methodSignature()`; if they don't, the call will either raise an exception or return something other than a list.

Finally, the program calls `system.methodHelp()` to fetch the help text for a given method. Again, this is optional, and if no help text is defined, the server returns an empty string.

Here's an example of running the program:

```
$ ./xmlrpclib.py
Gathering available methods...

Available Methods:
 1: meerkat.getChannels
 2: meerkat.getCategories
 3: meerkat.getCategoriesBySubstring
 4: meerkat.getChannelsByCategory
 5: meerkat.getChannelsBySubstring
 6: meerkat.getItems
 7: system.listMethods
 8: system.methodHelp
 9: system.methodSignature
Select one (q to quit): 9
```

Details for system.methodSignature

Args: string; Returns: array

Help: Returns an array of known signatures (an array of arrays) for the method name passed. If no signatures are known, returns a none-array (test for type != array to detect missing signature)

A Full-Featured Example

To illustrate the power of using XML-RPC, here's an actual news application built around XML-RPC and the Meerkat service:

```
#!/usr/bin/env python
# Full news reader example - Chapter 8 - xmlnewsreader.py
# This program requires Python 2.3 for the textwrap module

import xmlrpclib, sys, textwrap

class NewsCat:
    """Store categories in this class. It is easier to sort them nicely for
    the user if they can be compared with __cmp__."""
    def __init__(self, catdata):
        self.id = catdata['id']
        self.title = catdata['title']
```

```

def __cmp__(self, other):
    return cmp(self.title, other.title)

class NewsSource:
    """Primary class for a news source."""
    def __init__(self,
                 url = 'http://www.oreillynet.com/meerkat/xml-rpc/server.php'):
        self.s = xmlrpclib.ServerProxy(url)
        self.loadcats()

    def loadcats(self):
        """Load the categories from the XML-RPC server."""
        print "Loading categories..."
        catdata = self.s.meerkat.getCategories()
        self.cats = [NewsCat(item) for item in catdata]
        self.cats.sort()

    def displaycats(self):
        """Display a category menu."""
        numonline = 0
        i = 0
        for item in self.cats:
            sys.stdout.write("%2d: %20.20s " % (i + 1, item.title))
            i += 1
            numonline += 1
            if numonline % 3 == 0:
                sys.stdout.write("\n")
        if numonline != 0:
            sys.stdout.write("\n")

    def promptcat(self):
        """Ask the user for a category selection."""
        self.displaycats()
        sys.stdout.write("Select a category or q to quit: ")
        selection = sys.stdin.readline().strip()
        if selection == 'q':
            sys.exit(0)
        return int(selection) - 1

```

```

def dispcat(self, cat):
    """Displays a category (all the items in it)"""
    items = self.s.meerkat.getItems({'category': cat,
        'ids': 1,
        'descriptions': 1,
        'categories': 1,
        'channels': 1,
        'dates': 1,
        'num_items': 15})
    if not len(items):
        print "Sorry, no items in that category."
        sys.stdout.write("Press Enter to continue: ")
        sys.stdin.readline()
        return
    while 1:
        self.dispitemsummary(items)
        sys.stdout.write("Select story or q to go to main menu: ")
        selection = sys.stdin.readline().strip()
        if selection == 'q':
            return
        self.dispitem(items[int(selection) - 1])

def dispitemsummary(self, items):
    """Displays a summary of each item in the list items."""
    counter = 0
    for item in items:
        print "%2d: %s" % (counter + 1, item['title'])
        counter += 1

def dispitem(self, item):
    """Displays a single item."""
    print "--- %s ---" % item['title']
    print "Posted on", item['date']
    print "Description:"
    print textwrap.fill(item['description'])
    print "\nLink:", item['link']
    sys.stdout.write("\nPress Enter to continue: ")
    sys.stdin.readline()

```

```
n = NewsSource()
while 1:
    cat = n.promptcat()
    n.dispcat(cat)
```

This code should be largely self-explanatory. Note that it still doesn't perform any error-checking; if you supply an invalid number, the program will crash. Another thing to note is that after the `NewsSource.__init__` method, nothing in the program refers to XML-RPC. In fact, the rest of the code need not even know that calls are running through XML-RPC or using Meerkat specifically.

XML-RPC Error Handling

Things can go wrong with XML-RPC calls. Invalid arguments may be supplied, network connections may be down, or the server may be malfunctioning. Exceptions that you may see are summarized in Table 8-2.

Table 8-2. xmlrpclib Exceptions

Exception	Description
<code>xmlrpclib.Fault</code>	Indicates a server error processing your request
<code>xmlrpclib.ProtocolError</code>	HTTP transport layer had problem communicating with server
<code>xmlrpclib.ResponseError</code>	Server response couldn't be understood
<code>TypeError</code>	Arguments of an invalid type were supplied

XML-RPC Type Handling

The XML-RPC standard specifies certain types that may be transmitted. Python's `xmlrpclib` converts between those types and standard Python types. Table 8-3 describes this conversion.

Table 8-3. XML-RPC Type Conversion

Python Type	XML-RPC Type	Notes
int, long		
float	floating point	
str	string	
bool	boolean	Python programmers often use ints for this. If you're using a version of Python prior to 2.3, you can use <code>xmlrpclib.Boolean</code> objects to represent Boolean data.
list, tuple	array	
dict	structure	Keys must be strings; values must be of types shown in this table.
<code>xmlrpclib.DateTime</code>	date	
<code>xmlrpclib.Binary</code>	binary	

Passing booleans works best with Python 2.3. If you're using that version, you can use this to convert any normal Python condition to a boolean: `bool(condition)`.

Summary

XML is a standardized way to represent structured text or data. Python provides two ways to work with XML data: SAX and DOM. SAX is based on event processing, while DOM works by generating a tree structure to represent the document.

DOM trees have a Document node at the top of the structure. Some nodes in a DOM tree have children; the parent/child relationship is used to represent the hierarchical nature of the document.

Parsing an existing XML document can generate DOM trees, or they can be created from scratch by your program.

XML-RPC is a language-neutral way of passing requests and answers across the network. With a little bit of glue, calls to XML-RPC servers can look and feel just like calls to standard Python functions. XML-RPC uses XML for the underlying data representation, but XML-RPC users aren't required to know XML.



Part Three

E-mail Services

E-Mail Composition and Decoding

E-MAIL IS ONE OF THE most important means of communication available on the Internet. Millions of people exchange e-mail each day. A countless number of automated systems process e-mail as well: Everything from online order confirmations to mail readers involves computer systems that must generate, route, deliver, receive, or display messages. In this part of the book, you'll learn about e-mail messages and the protocols that handle them. This chapter focuses on the messages themselves, while the following chapters discuss delivering and downloading messages.

The format for an e-mail message is important to all the different e-mail protocols. In this chapter, you won't see any actual messages being sent; rather, you'll learn how to assemble messages (both plain and with attachments and other features) and disassemble them back into their component parts. This is often the most complex piece of the e-mail puzzle.

You'll look at e-mail messages in two ways: first, as traditional messages—simple text messages. Then, you'll look at Multipurpose Internet Mail Extensions (MIME) that form a layer above traditional messages and enable things like attachments and alternative representations of messages.

CAUTION The `email` module changed significantly in Python 2.2.2. Therefore, all examples in this chapter assume you have Python 2.2.2 or later. If you must work with versions of Python prior to 2.2.2, you may wish to investigate the `rfc822`, `mimetools`, and `multifile` modules. However, I recommend upgrading to a newer version of Python instead.

Understanding Traditional Messages

Each traditional e-mail message contains two distinct parts: *headers* and the *body*. Headers contain control data—such as the sender, destination, and subject of the message—while the body contains the message text itself. The headers

always come first, then the body. The headers are separated from the body by a blank line. Here's an example of a very simple message:

```
From: Jane Smith <jsmith@example.com>
To: Alan Jones <ajones@example.com>
Subject: Testing This E-Mail Thing
```

Hello Alan,

This is just a test message. Thanks.

Header Processing

The preceding example shows a very minimal set of headers. A message might have headers like that when a mail reader sends it out. However, as soon as it's sent, the mail server will likely add a Date header, a Received header, and possibly many more. Most mail readers don't display all the headers of a message, but if you look in your mail reader's options for an option such as "show all headers" or "view source," you should be able to see them. Here's an example from a few years ago of a message with all its headers intact:

```
Received: (at control) by bugs.debian.org; 2 Sep 1999 03:21:05 +0000
Received: (qmail 9325 invoked from network); 2 Sep 1999 03:21:05 -0000
Received: from erwin.complete.org (209.197.212.34)
    by master.debian.org with SMTP; 2 Sep 1999 03:21:04 -0000
Received: (from jgoerzen@localhost)
    by erwin.complete.org (8.9.3/8.9.3/Debian/GNU) id WAA13110;
    Wed, 1 Sep 1999 22:21:00 -0500
Sender: jgoerzen@erwin.complete.org
To: control@bugs.debian.org
Subject: foo
Mime-Version: 1.0 (generated by tm-edit 7.108)
Content-Type: text/plain; charset=US-ASCII
From: John Goerzen <jgoerzen@complete.org>
Date: 01 Sep 1999 22:21:00 -0500
Message-ID: <87u2pexalf.fsf@erwin.complete.org>
Lines: 7
X-Mailer: Gnus v5.6.45/XEmacs 20.4 - "Emerald"

severity 43733 wishlist
```

There are many more headers here than in the first example. Let's take a look at them. First, notice the Received headers. Mail servers insert them. Each mail server that the message passes through adds a new Received header above the others. You can see that this message passed through four mail servers.

Some mail server along the way—or possibly the mail reader—added the Sender line, which is similar to the From line. The Mime-Version and Content-Type headers will be discussed later on in this chapter in the “Understanding MIME” section. The Message-ID header is supposed to be a globally unique way to identify any particular message and is generated by either the mail reader or mail server when the message is first sent. The Lines header indicates the length of the message. The mail reader I used at the time, Gnus, added an X-Mailer header.

If you viewed this message in a normal mail reader, you would likely see only To, From, Subject, and Date by default. Even though this message is several years old, it would still be perfectly valid today.

Headers Don't Address Your Mail

You're probably quite accustomed to working with To, Cc, and Bcc headers in your own mail reader. These headers normally appear to control which people receive an e-mail. In fact, e-mail headers *don't* determine who gets the message, except in very special circumstances.

The recipients of a message are determined solely by the recipients specified during the Simple Mail Transport Protocol (SMTP) exchange between servers. For more details on SMTP, please refer to Chapter 10. Here are a few examples of situations in which this is apparent.

Headers with Blind Carbon Copies

If you send many e-mails, you've probably used the “Bcc” or blind carbon copy, feature of your mail reader. The Bcc feature lets you send a message to someone, but does so in a way so that “To” and “Cc” recipients don't know that it's being sent to someone else.

This is accomplished quite simply, in fact: Your mail reader never puts a Bcc header into the message. It puts the To and Cc headers there, but no Bcc header. Despite the recipients being mentioned nowhere in the message, the system still knows how to handle the situation. That's because the actual recipients are specified during the SMTP conversation with the mail server.

Headers with Mailing Lists

Mailing lists are special e-mail servers that receive a message on an address and then redistribute that message to a large list of subscribers. For example, `debian-user@lists.debian.org` is a mailing list. If you send a message to that address, it will be re-sent to tens of thousands of recipients.

Yet when you receive a message from the mailing list, the `To` header still shows `debian-user@lists.debian.org`—your address isn't there. Again, this is because the `To` header doesn't matter to the actual delivery process; it's SMTP that carries this information.

Headers with Spam

If you've been using your e-mail address for a while, chances are that you receive a great deal of unwanted commercial e-mail, or "spam." If you inspect the spam that you receive closely, you'll observe that many pieces of spam use totally fake information in the `From` and `To` headers—and some even add fake `Received` headers as well. This is to disguise the true origin of the mail, and unfortunately, is quite easy to do.

Exceptions to the Rule

Earlier I said that the headers don't determine the recipient "except in very special circumstances." These circumstances generally occur on UNIX and Linux machines. One way for mail readers (and other programs) to send mail is to pass the messages to the standard system program `/usr/sbin/sendmail`. Normally, recipients are specified on the command line to `sendmail` (again, the headers aren't consulted). However, it's possible to run `/usr/sbin/sendmail -t`. The `-t` causes the mail transport system to consult the message headers for the initial recipient list.

This is only a convenience feature. Once the headers are consulted, they're ignored for the remainder of the delivery process, which works like usual.

Additionally, some mail servers contain built-in spam or virus scanners. These scanners may consult the message headers to determine if the mail is spam, and if so, they will refuse the message on those grounds.

Display Information in Headers

Now that I've established that headers don't actually help to get your e-mail anywhere, you may be wondering what their purpose is. Headers are there to help mail readers. Here are some ways that mail readers use headers:

- The `From` header identifies the message sender to the user. It's also usually used when the user clicks the Reply button. The new message is sent to the address in the `From` header.
- The `Reply-To` header can set an alternative address for replies.
- The `Subject` header is often used for mailbox summaries.
- The `Date` header can be used to sort a mailbox by arrival date.
- The `Message-ID` and `In-Reply-To` headers help some mail readers perform threading (arranging messages hierarchically).
- The MIME headers help the mail reader display the message in the proper language, with proper formatting, and they process attachments correctly.

So, headers clearly do have an important role in the proper operation of an e-mail system. They just aren't used to actually deliver the mail.

Composing Traditional Messages

Now that you know what a traditional message looks like, let's generate one. In Python, the modules that are used to generate messages are installed under the `email` package. In this example, you'll use just one: `MIMEText`, though this program doesn't concern itself with MIME just yet. Traditional messages are limited to 7-bit data—generally that means English text only, or other languages written entirely using the standard Latin alphabet used in English.

```
#!/usr/bin/env python
# Traditional Message Generation, Simple -- Chapter 9
# trad_gen_simple.py
# This program requires Python 2.2.2 or above

from email.MIMEText import MIMEText
message = """Hello,
```

This is a test message from Chapter 9. I hope you enjoy it!

-- Anonymous""

```
msg = MIMEText(message)
msg['To'] = 'recipient@example.com'
msg['From'] = 'Test Sender <sender@example.com>'
msg['Subject'] = 'Test Message, Chapter 9'

print msg.as_string()
```

The program is simple. It creates a `MIMEText` object based on the body of the message, sets the headers, and prints the result. When you run this program, you get a nice formatted message with proper headers. The output is suitable for transmission right away. Here's what it looks like:

```
$ ./trad_gen_simple.py
Content-Type: text/plain; charset="us-ascii"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit
To: recipient@example.com
From: Test Sender <sender@example.com>
Subject: Test Message, Chapter 9
```

Hello,

This is a test message from Chapter 9. I hope you enjoy it!

-- Anonymous

Notice that three MIME headers were added; let's not worry about them for now. The other headers were added just as they were specified.

Adding the Date and Message-ID Headers

Most messages should have a Date header. The date is generated in a specific format for e-mail messages. Fortunately, there's `email.Utils.formatdate()` to generate the text for you.

Also, you should add a Message-ID header to new messages. This header is to be generated in such a way that no other message anywhere will ever have the same Message-ID. There's a function to help do that as well: `email.Utils.make_msgid()`. Here's an updated example that adds both of those headers to the message:

```

#!/usr/bin/env python
# Traditional Message Generation with Date and Message-ID -- Chapter 9
# trad_gen_newhdrs.py
# This program requires Python 2.2.2 or above

from email.MIMEText import MIMEText
from email import Utils
message = """Hello,

This is a test message from Chapter 9. I hope you enjoy it!

-- Anonymous"""

msg = MIMEText(message)
msg['To'] = 'recipient@example.com'
msg['From'] = 'Test Sender <sender@example.com>'
msg['Subject'] = 'Test Message, Chapter 9'
msg['Date'] = Utils.formatdate(localtime = 1)
msg['Message-ID'] = Utils.make_msgid()

print msg.as_string()

```

If you run the program, you'll notice two new headers in the output, as shown here:

```

$ ./trad_gen_newhdrs.py
Content-Type: text/plain; charset="us-ascii"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit
To: recipient@example.com
From: Test Sender <sender@example.com>
Subject: Test Message, Chapter 9
Date: Sat, 06 Dec 2003 11:51:08 -0500
Message-ID: <20031206175108.909.20458@grumpy.example.com>

```

Hello,

This is a test message from Chapter 9. I hope you enjoy it!

-- Anonymous

This message is now ready to send.

How Message-IDs Are Made Unique

Earlier, I stated that Message-ID headers are supposed to be generated in such a way that a given message ID will never occur for any other message anywhere in the world.

This is accomplished by adhering to a set of loose guidelines. The part of the message ID to the right of the at sign (@) is the full hostname of the machine that's generating the message ID. This helps ensure that a message ID is uniquely tied to a given computer. The part on the left is typically generated using a combination of the date and time, process ID of the program generating the ID, and some random data. The scheme isn't perfect, but is nearly so and works well in practice.

Parsing Traditional Messages

The `email` module also provides support for parsing e-mail messages. After parsing them, you can easily access individual headers and the body of the message. Here's a sample message that you'll parse. I'll call it `message.txt`.

```
Received: By test server from somewhere
Received: By another server from my machine
To: recipient@example.com
From: Test Sender <sender@example.com>
Subject: Test Message, Chapter 9
Date: Tue, 09 Dec 2003 15:29:18 -0600
Message-ID: <20031209212918.10574.60752@somewhere.example.com>
```

Hello,

This is a test message from Chapter 9. I hope you enjoy it!

-- Anonymous

Notice that it has two Received headers. This is normal for a delivered message and it's acceptable.

Basic Message Parsing

Here's a simple example of message parsing. It will first display all the headers, then pick out one in particular (the Subject line) to display. Finally, it displays the body of the message.

```
#!/usr/bin/env python
# Traditional Message Parsing -- Chapter 9
# trad_parse.py
# This program requires Python 2.2.2 or above

import sys, email

msg = email.message_from_file(sys.stdin)
print " *** Headers in message: "
for header, value in msg.items():
    print header + ":"
    print " " + value

if msg.is_multipart():
    print "This program cannot handle MIME multipart messages; exiting."
    sys.exit(1)

print "-" * 78
if 'subject' in msg:
    print "Subject: ", msg['subject']
    print "-" * 78
print "Message Body:"
print

print msg.get_payload()
```

The program starts out by loading in the message with the call to `email.message_from_file()`. That function will load a message into memory and parse it.

Next, it iterates over all the header/value pairs in the message. The code uses `msg.items()` instead of an operation such as `for key in msg.keys()` because a single key (header) may occur more than once. By accessing it as a regular dictionary, you can only retrieve one value.

Later, the program does access the message as a regular dictionary to retrieve the Subject line. Since each message has only one, this is a convenient way to load it. Finally, the message body is displayed. The program's output looks like this:

```
$ ./trad_parse.py < message.txt
*** Headers in message:
Received:
    By test server from somewhere
Received:
    By another server from my machine
To:
    recipient@example.com
From:
    Test Sender <sender@example.com>
Subject:
    Test Message, Chapter 10
Date:
    Tue, 09 Dec 2003 15:29:18 -0600
Message-ID:
    <20031209212918.10574.60752@somewhere.example.com>
```

Subject: Test Message, Chapter 10

Message Body:

Hello,

This is a test message from Chapter 10. I hope you enjoy it!

-- Anonymous

Parsing Dates

Parsing dates out of e-mail messages isn't easy. While there's a standard governing the representation of dates in message headers, it isn't trivial to implement correctly, and many mail readers generate invalid Date headers anyway. Though Python's `email.Utils` module does help out, you still have to be careful.

Standard-conforming dates typically are listed in the sender's local time zone, and also include an offset from Coordinated Universal Time (UTC). To a program parsing things, this isn't very helpful. Python internally uses seconds since midnight on January 1, 1970, UTC (the "epoch") for time and date calculations. So the time, date, and time zone must all be resolved to arrive at an accurate date.

The function `parsedate_tz()` will load a Date string, and hopefully return a ten-element tuple. It might not if the input is bad in some way. The first nine elements of the tuple can be passed to `time.mktime()`; the tenth specifies the time zone, which `mktime()` doesn't understand. Therefore, there's another function, `mktime_tz()`, which handles this special ten-element tuple and converts it into a standard seconds-since-epoch value. Here's an example program for date parsing:

```
#!/usr/bin/env python
# Date Parsing - Chapter 9 - date_parse.py
# This program requires Python 2.2.2 or above

import sys, email, time
from email import Utils

def getdate(msg):
    """Returns the date/time from msg in seconds-since-epoch, if possible.
    Otherwise, returns None."""
    if not 'date' in msg:
        # No Date header present.
        return None

    datehdr = msg['date'].strip()
    try:
        return Utils.mktime_tz(Utils.parsedate_tz(datehdr))
    except:
        # Some sort of error occurred, likely because of an invalid date.
        return None

msg = email.message_from_file(sys.stdin)

dateval = getdate(msg)
if dateval is None:
    print "No valid date was found."
else:
    print "Message was sent on", time.strftime('%A, %B %d %Y at %I:%M %p',
                                                time.localtime(dateval))
```

When you run this program, it will display the accurate time from the input, as follows:

```
$ ./date_parse.py < message.txt
Message was sent on Tuesday, December 09 2003 at 03:29 PM
```

Depending on your time zone, the output may vary slightly. The `getdate()` function is cautious with error-checking. First, it makes sure that the message actually has a `Date` header. Then, it looks for exceptions during the parsing process and returns `None` if exceptions are detected.

Dates with the `datetime` Module

Python 2.3 programmers may wish to use the new `datetime` module to manipulate dates from e-mails. The `getdate()` function returns a value suitable for passing to the `datetime` function `fromtimestamp()`.

Understanding MIME

Multipurpose Internet Mail Extensions (MIME) is a set of rules for encoding data in e-mails. They provide for such things as attachments, alternative formats of messages, communicating language encodings to the reader, and many other message structures. MIME can be complex, though fortunately most messages are fairly simple to work with. Some of the same `email` module functions already demonstrated can be used with MIME; however, there are also a number of new ones.

MIME Concepts

MIME is a suite of additions to e-mail messages. MIME is designed to be backwards-compatible whenever possible, so mail readers that aren't MIME-aware should still be able to present a reasonable version of messages. For instance, messages containing HTML should still contain a plain text part that older mail readers can display normally. There are several features of MIME to consider. You've probably used many of them regularly, but might not know exactly what's going on "under the hood."

MIME supports *multipart* messages. Normal e-mail messages contain headers and bodies. When you consider a MIME multipart message, you can have several parts in the body. These parts might be things like message text and attachments. Or, you can have *alternative multiparts*, which represent the same content different ways (for instance, plain text and HTML).

Attachments are a frequently used form of multipart messages. They're specified in such a way that the mail reader knows to give the user the chance to download or save them.

MIME supports different *transfer encodings*. Traditional e-mail messages are limited to 7-bit data, which renders them unusable for much other text using the Latin alphabet as used in English. MIME has several ways of transforming 8-bit data in ways such that it fits within the confines of e-mail systems. The "plain" encoding is the same as you would see in traditional messages, and passes text unmodified. Base-64 is a way of encoding raw binary data, and most of your attachments—such as ZIP files—would be encoded with base-64. Quoted-printable is a hybrid that tries to make the plain English text readable in old mail readers, while letting new ones also see the other characters. It's primarily used for languages such as German, which uses mostly the same Latin alphabet as English, but adds a few other characters as well.

MIME also provides *content types*, which tell the recipient what kind of content is present. For instance, `text/plain` is a plain text message, and `image/jpeg` is a JPEG image. Along with content types, for text parts, MIME can specify a *character set*. Different languages use different meanings for the same character, and with this support, a mail reader can display messages from many different languages simultaneously.

MIME content types actually are also used by other protocols. For instance, HTTP uses MIME content types to represent types of documents sent over the Web.

That's a lot to understand and support. Chances are that you won't need to support all of that in any single program.

How MIME Works

You'll recall that MIME messages must work within the limited framework of traditional messages. To do that, the MIME specification defines some headers and specially formatted body text.

For nonmultipart messages, MIME simply adds some headers to specify the type of content, its character set, and so on. For multipart messages, things get trickier. MIME places a special marker in the body that separates one part from the next. Each part can have its own (limited) set of headers, which occur at the start of the part, followed by data.

By convention, the most basic content (plain text message, if any) comes first, so that people without MIME-aware readers can read the plain text without having to see all the MIME data. Fortunately, Python can parse all of this and generate a parse tree to use. In fact, it generates a hierarchy similar to the Document Object Model in Chapter 8.

Composing MIME Attachments

To compose a message with attachments, you'll generally follow these steps:

1. Create a `MIMEMultipart()` object and set its message headers.
2. Create a `MIMEText()` object with the message body text, and attach it to the `MIMEMultipart()` object.
3. Create appropriate MIME objects for each attachment, and attach them to the `MIMEMultipart()` object.
4. Call `as_string()` on the `MIMEMultipart()` object to get the resulting message.

Here's a program that implements this algorithm:

```
#!/usr/bin/env python
# MIME attachment generation - Chapter 9 - mime_gen_basic.py
# This program requires Python 2.2.2 or above

from email.MIMEText import MIMEText
from email.MIMEMultipart import MIMEMultipart
from email.MIMEBase import MIMEBase
from email import Utils, Encoders
import mimetypes, sys

def attachment(filename):
    fd = open(filename, 'rb')
    mimetype, mimeencoding = mimetypes.guess_type(filename)
    if mimeencoding or (mimetype is None):
        mimetype = 'application/octet-stream'
    maintype, subtype = mimetype.split('/')
    if maintype == 'text':
        retval = MIMEText(fd.read(), _subtype=subtype)
    else:
        retval = MIMEBase(maintype, subtype)
        retval.set_payload(fd.read())
        Encoders.encode_base64(retval)
    retval.add_header('Content-Disposition', 'attachment',
                      filename = filename)
    fd.close()
    return retval
```

```

message = """Hello,
This is a test message from Chapter 9. I hope you enjoy it!
-- Anonymous"""

msg = MIME Multipart()
msg['To'] = 'recipient@example.com'
msg['From'] = 'Test Sender <sender@example.com>'
msg['Subject'] = 'Test Message, Chapter 9'
msg['Date'] = Utils.formatdate(localtime = 1)
msg['Message-ID'] = Utils.make_msgid()

body = MIMEText(message, _subtype='plain')
msg.attach(body)
for filename in sys.argv[1:]:
    msg.attach(attachment(filename))
print msg.as_string()

```

If you look at this code, you can see that parts of it look similar to the code that generated a traditional e-mail. It starts out by creating a message object—a `MIMEMultipart` object in this case. Then, the same headers are set as before.

Next, the body object is created—a `MIMEText` object similar to the one used earlier. This object is attached to the main message.

Then, the program loops over each file given on the command line. For each one, it creates a message object and attaches it. The `attachment()` function does the work of creating a message attachment object. First, it determines the MIME type of a file by using the `mimetypes` module. If the type can't be determined, or it has some kind of encoding, a default type of `application/octet-stream` is used.

If you're dealing with a text document, a `MIMEText` object is created to handle it. Otherwise, a `MIMEBase` generic object is created, and the contents are assumed to be binary, so they're encoded with base-64. Finally, a `Content-Disposition` header is added so that mail readers know that they're dealing with an attachment.

Take a look at the result from this program:

```

$ echo "This is a test" > test.txt
$ gzip < test.txt > test.txt.gz
$ ./mime_gen_basic.py test.txt test.txt.gz
Content-Type: multipart/mixed; boundary="=====1623374356=="
MIME-Version: 1.0
To: recipient@example.com
From: Test Sender <sender@example.com>
Subject: Test Message, Chapter 10

```

Date: Thu, 11 Dec 2003 16:00:55 -0600
Message-ID: <20031211220055.12211.26885@host.example.com>

=====1623374356==
Content-Type: text/plain; charset="us-ascii"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit

Hello,

This is a test message from Chapter 10. I hope you enjoy it!

-- Anonymous
=====1623374356==
Content-Type: text/plain; charset="us-ascii"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit
Content-Disposition: attachment; filename="test.txt"

This is a test

=====1623374356==
Content-Type: application/octet-stream
MIME-Version: 1.0
Content-Transfer-Encoding: base64
Content-Disposition: attachment; filename="test.txt.gz"

H4sIAP3o2D8AAwvJyCxWAKJEhZLU4hIuAIwtwPoPAAAA

=====1623374356====

The message starts similar to the traditional ones; you can see the To, From, Subject, Date, and Message-ID headers just like before. Note the Content-Type line, however. In this message, it indicates multipart/mixed. That tells the mail reader that the body of the message contains multiple parts, and that the string containing equals signs separates them.

Next, there's the first part. Notice that it has its own Content-Type header, giving the type of that particular part. Then, the text for the first part is given.

The second part looks similar to the first, but has the additional Content-Disposition header that the program added. Finally, there's the binary file. This one is encoded with base-64, so it's not directly readable.

Composing MIME Alternatives

MIME alternatives let you generate multiple versions of a single document. The user's mail reader will then automatically decide which one to display. The process of creating alternatives is similar to attachments, and is illustrated by the following example:

```
#!/usr/bin/env python
# MIME alternative generation - Chapter 9 - mime_gen_alt.py
# This program requires Python 2.2.2 or above

from email.MIMEText import MIMEText
from email.MIMEMultipart import MIMEMultipart
from email import Utils, Encoders
import mimetypes, sys

def alternative(data, contenttype):
    maintype, subtype = contenttype.split('/')
    if maintype == 'text':
        retval = MIMEText(data, _subtype=subtype)
    else:
        retval = MIMEBase(maintype, subtype)
        retval.set_payload(data)
        Encoders.encode_base64(retval)
    return retval

messagetext = """Hello,
```

This is a *great* test message from Chapter 9. I hope you enjoy it!

```
-- Anonymous"""
messagehtml = """Hello,<P>
This is a <B>great</B> test message from Chapter 9. I hope you enjoy
it!<P>
-- <I>Anonymous</I>"""


```

```
msg = MIMEMultipart('alternative')
msg['To'] = 'recipient@example.com'
msg['From'] = 'Test Sender <sender@example.com>'
msg['Subject'] = 'Test Message, Chapter 9'
msg['Date'] = Utils.formatdate(localtime = 1)
msg['Message-ID'] = Utils.make_msgid()
```

```
msg.attach(alternative(messagetext, 'text/plain'))
msg.attach(alternative(messagehtml, 'text/html'))
print msg.as_string()
```

Notice the differences between an alternative message and a message with attachments. With the alternative message, no Content-Disposition header is inserted. Also, the `MIMEMultipart` object is passed the alternative subtype to tell the mail reader that all objects in this multipart are alternative views of the same thing. Note again that the plain text object should come first. The result of this program is as follows:

```
$ ./mime_gen_alt.py
Content-Type: multipart/alternative; boundary="=====1543078954=="
MIME-Version: 1.0
To: recipient@example.com
From: Test Sender <sender@example.com>
Subject: Test Message, Chapter 10
Date: Thu, 11 Dec 2003 19:36:56 -0600
Message-ID: <20031212013656.21447.34593@user.example.com>

=====1543078954==
Content-Type: text/plain; charset="us-ascii"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit

Hello,<P>
This is a *great* test message from Chapter 10. I hope you enjoy it!

-- Anonymous
=====1543078954==
Content-Type: text/html; charset="us-ascii"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit

Hello,<P>
This is a <B>great</B> test message from Chapter 10. I hope you enjoy
it!<P>
-- <I>Anonymous</I>
=====1543078954====
```

An HTML-capable mail reader will choose the second view and give the user a nice HTML representation of the message. A text-only reader will choose the first view, and the user will still get a nice representation of the message.

Composing Non-English Headers

Although you've seen how MIME can encode message body parts with base-64 to allow 8-bit data to pass through, that doesn't solve the problem of headers. For instance, if your name was Michael Müller, you would have trouble representing your name in your own alphabet.

MIME provides a way to encode data in headers. To do so, you must specify a character set; in this case, ISO 8859-1 (Western Europe) is the proper choice. Here's how to do it:

```
#!/usr/bin/env python
# MIME message generation with 8-bit headers - Chapter 9
# mime_headers.py
# This program requires Python 2.2.2 or above
```

```
from email.MIMEText import MIMEText
from email.Header import Header
from email import Utils
message = """Hello,
```

This is a test message from Chapter 9. I hope you enjoy it!

```
-- Anonymous"""
```

```
msg = MIMEText(message)
msg['To'] = 'recipient@example.com'
fromhdr = Header("Michael Müller", 'iso-8859-1')
fromhdr.append('<mmueller@example.com>', 'ascii')
msg['From'] = fromhdr
msg['Subject'] = Header('Test Message, Chapter 10')
msg['Date'] = Utils.formatdate(localtime = 1)
msg['Message-ID'] = Utils.make_msgid()

print msg.as_string()
```

The code \xfc represents the character 0xFC, which is ü in ISO 8859-1. An encoded version of the name is generated. Notice that the e-mail address is then appended separately, with a different character set. If that didn't happen, the encoder would mangle the e-mail address, and MIME-unaware programs would be unable to reply to the message.

Also observe the encoding for the `Subject` line. No character set is specified, so `email.Header.Header` defaults to `ascii`—and doesn't alter it at all.

If you run this example, you'll see a result like this:

```
$ ./mime_headers.py
Content-Type: text/plain; charset="us-ascii"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit
To: recipient@example.com
From: =?iso-8859-1?q?Michael_M=FClle?= <mmueller@example.com>
Subject: Test Message, Chapter 9
Date: Thu, 11 Dec 2003 19:37:56 -0600
Message-ID: <20031212013756.21447.34593@user.example.com>
```

Hello,

This is a test message from Chapter 9. I hope you enjoy it!

-- Anonymous

Notice how the name is adjusted to be representable with a particular character set, but the e-mail address is unmodified.

Composing Nested Multiparts

Now that you know how to generate a message with alternatives and one with attachments, you may be wondering how to do both. To do that, you create a standard multipart for the main message. Then you create a `multipart/alternative` for your body text, and attach your message formats to it, then attach it to the main message. Finally, you attach the various files. Here's a program that does just that:

```

#!/usr/bin/env python
# MIME generation of embedded multipart - Chapter 9
# mime_gen_both.py
# This program requires Python 2.2.2 or above

from email.MIMEText import MIMEText
from email.MIMEMultipart import MIMEMultipart
from email.MIMEBase import MIMEBase
from email import Utils, Encoders
import mimetypes, sys

def genpart(data, contenttype):
    maintype, subtype = contenttype.split('/')
    if maintype == 'text':
        retval = MIMEText(data, _subtype=subtype)
    else:
        retval = MIMEBase(maintype, subtype)
        retval.set_payload(data)
        Encoders.encode_base64(retval)
    return retval

def attachment(filename):
    fd = open(filename, 'rb')
    mimetype, mimeencoding = mimetypes.guess_type(filename)
    if mimeencoding or (mimetype is None):
        mimetype = 'application/octet-stream'
    retval = genpart(fd.read(), mimetype)
    retval.add_header('Content-Disposition', 'attachment',
                      filename = filename)
    fd.close()
    return retval

messagetext = """Hello,

This is a *great* test message from Chapter 9. I hope you enjoy it!

-- Anonymous"""
messagehtml = """Hello,<P>
This is a <B>great</B> test message from Chapter 9. I hope you enjoy
it!<P>
-- <I>Anonymous</I>"""

```

```

msg = MIME_Multipart()
msg['To'] = 'recipient@example.com'
msg['From'] = 'Test Sender <sender@example.com>'
msg['Subject'] = 'Test Message, Chapter 9'
msg['Date'] = Utils.formatdate(localtime = 1)
msg['Message-ID'] = Utils.make_msgid()

body = MIME_Multipart('alternative')
body.attach(genpart(messagetext, 'text/plain'))
body.attach(genpart(messagehtml, 'text/html'))
msg.attach(body)

for filename in sys.argv[1:]:
    msg.attach(attachment(filename))
print msg.as_string()

```

The output from this program is large, so I won't show it here. You should also know that there's no fixed limit to how deep message components may be nested; however, it's generally best to avoid nesting any more than necessary.

Parsing MIME Messages

Python's `email` module can read a message from a file or a string and generate the same in-memory structure that you would generate yourself. So, to handle an incoming e-mail, all you have to do is navigate that structure. As a side benefit, you can make adjustments (for instance, you can remove an attachment), and then generate an e-mail message based on the new tree. Here's a program that will read in a message and display its structure by walking the tree, as shown here:

```

#!/usr/bin/env python
# MIME Message Parsing - Chapter 9 - mime_structure.py
# This program requires Python 2.2.2 or above

import sys, email

```

```

def printmsg(msg, level = 0):
    l = " " * level
    l2 = l + "|"
    print l + "+ Message Headers:"
    for header, value in msg.items():
        print l2, header + ":", value
    if msg.is_multipart():
        for item in msg.get_payload():
            printmsg(item, level + 1)

msg = email.message_from_file(sys.stdin)
printmsg(msg)

```

This program is short and simple. It simply walks the message tree. For each object, it checks to see if it's a multipart; if so, the children of that object are displayed as well. The output of this program will look somewhat like this, given the input of a message that contains a body in alternative form and a single attachment:

```

$ ./mime_gen_both.py /tmp/test.gz | ./mime_structure.py
+ Message Headers:
| Content-Type: multipart/mixed; boundary="=====1899932228=="
| MIME-Version: 1.0
| To: recipient@example.com
| From: Test Sender <sender@example.com>
| Subject: Test Message, Chapter 10
| Date: Fri, 12 Dec 2003 16:23:05 -0600
| Message-ID: <2003121222305.13361.15560@user.example.com>
| + Message Headers:
| | Content-Type: multipart/alternative; boundary="=====1287885775=="
| | MIME-Version: 1.0
| | + Message Headers:
| | | Content-Type: text/plain; charset="us-ascii"
| | | MIME-Version: 1.0
| | | Content-Transfer-Encoding: 7bit
| | + Message Headers:
| | | Content-Type: text/html; charset="us-ascii"
| | | MIME-Version: 1.0
| | | Content-Transfer-Encoding: 7bit
| + Message Headers:
| | Content-Type: application/octet-stream
| | MIME-Version: 1.0
| | Content-Transfer-Encoding: base64
| | Content-Disposition: attachment; filename="/tmp/test.gz"

```

Decoding Parts

Individual parts of a message can easily be extracted. You'll recall, though, that there are several ways that message data may be encoded. The `email` module, fortunately, can easily decode them if desired. Here's a program that will let you decode and save any component of a MIME message:

```
#!/usr/bin/env python
# MIME part decoding - Chapter 9 - mime_decode.py
# This program requires Python 2.2.2 or above

import sys, email
counter = 0
parts = []

def printmsg(msg, level = 0):
    global counter
    l = " | " * level
    ls = l + "*"
    l2 = l + "|"
    if msg.is_multipart():
        print l + "Found multipart:"
        for item in msg.get_payload():
            printmsg(item, level + 1)
    else:
        disp = ['%d. Decodable part' % (counter + 1)]
        if 'content-type' in msg:
            disp.append(msg['content-type'])
        if 'content-disposition' in msg:
            disp.append(msg['content-disposition'])
        print l + ", ".join(disp)
        counter += 1
        parts.append(msg)

inputfd = open(sys.argv[1])
msg = email.message_from_file(inputfd)
printmsg(msg)

while 1:
    print "Select part number to decode or q to quit: "
    part = sys.stdin.readline().strip()
    if part == 'q':
        sys.exit(0)
```

```

try:
    part = int(part)
    msg = parts[part - 1]
except:
    print "Invalid selection."
    continue

print "Select file to write to:"
filename = sys.stdin.readline().strip()
try:
    fd = open(filename, 'wb')
except:
    print "Invalid filename."
    continue

fd.write(msg.get_payload(decode = 1))

```

This program steps through the message like the last example. Note though that only components that aren't multipart are offered for decoding. That's because a multipart object contains only other message objects; it has no actual payload of its own. If you run the program, you'll see output like this:

```

$ ./mime_decode.py testmessage.txt
Found multipart:
| Found multipart:
| | 1. Decodable part, text/plain; charset="us-ascii"
| | 2. Decodable part, text/html; charset="us-ascii"
| 3. Decodable part, application/octet-stream, attachment; filename="/tmp/
test.gz"
Select part number to decode or q to quit:
3
Select file to write to:
/tmp/newfile.gz
Select part number to decode or q to quit:
q

```

Decoding Headers

The last trick that remains to be dealt with regarding MIME messages is decoding headers that may have been encoded with foreign languages. It's not tough with the `email.Header` module. The function `decode_header()` returns a list of header components. Each component in the list is a separately encoded part of the

header, and contains the decoded text along with its character set. For instance, consider this Python session snippet:

```
>>> x = '=?iso-8859-1?q?Michael_M=FClle?= <mmueller@example.com>'
>>> from email import Header
>>> Header.decode_header(x)
[('Michael M\xfcller', 'iso-8859-1'), ('<mmueller@example.com>', None)]
```

The e-mail address wasn't encoded, so its character set was returned as `None`. Here's a program that applies this knowledge to header parsing:

```
#!/usr/bin/env python
# MIME Header Parsing - Chapter 9
# mime_parse_headers.py
# This program requires Python 2.2.2 or above

import sys, email, codecs
from email import Header

msg = email.message_from_file(sys.stdin)
for header, value in msg.items():
    headerparts = Header.decode_header(value)
    headerval = []
    for part in headerparts:
        data, charset = part
        if charset is None:
            charset = 'ascii'
        dec = codecs.getdecoder(charset)
        enc = codecs.getencoder('iso-8859-1')
        data = enc(dec(data)[0])[0]
        headerval.append(data)
    print "%s: %s" % (header, " ".join(headerval))
```

In this example, each header is parsed with `Header.decode_header()`. Then, we assume that ISO 8859-1 is the local character set (that may not always be a safe assumption; UNIX/Linux users may be able to use `locale.getpreferredencoding(True)` to get the preferred set) and recode all data to that character set. Python's general-purpose `codecs` module is helpful for that, or you can use the `encode()` and `decode()` methods of strings. Finally, the result is printed out. Here's what the result might look like:

```
$ ./mime_headers.py | ./mime_parse_headers.py
Content-Type: text/plain; charset="us-ascii"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit
To: recipient@example.com
From: Michael Müller <mmueller@example.com>
Subject: Test Message, Chapter 10
Date: Sat, 13 Dec 2003 21:07:08 -0600
Message-ID: <20031214030708.32141.24712@christoph>
```

Summary

Traditional e-mail messages contain headers and a body. All parts of a traditional message must be represented using a 7-bit encoding, which generally prohibits the use of anything other than text using the Latin alphabet as used in English.

Headers provide useful information for mail-reader programs and for people reading mail. Contrary to what many expect, except in special circumstances, the headers don't directly dictate where messages get sent.

Python's `email` modules can both generate messages and parse messages. To generate a traditional message, an instance of `email.MIMEText` or `email.Message` can be created. The Date and Message-ID headers aren't added by default, but can be easily added using convenience functions.

To parse a traditional or MIME message, you can call `email.message_from_file(fd)` where `fd` is the file descriptor to read from. Parsing of Date headers can be tricky but is usually possible without too much difficulty.

MIME is a set of extensions to the e-mail format that permit things such as nontext data, attachments, alternative views of content, and different character sets. Multipart MIME messages can be used for attachments and alternative views, and they're constructed in a "tree" fashion.

Simple Message Transport Protocol

MAIL SERVERS USE SMTP to transmit messages across the Internet. A mail server on the Internet will accept connections using SMTP and store messages in the recipients' mailbox. Some mail servers also use SMTP to forward messages on to peers. SMTP is a protocol primarily concerned with sending, forwarding, and storing e-mail. People downloading mail from a server will generally use POP (see Chapter 11) or IMAP (see Chapter 12). For Python programmers, SMTP is usually most interesting for sending e-mail.

Many programmers have the need to generate and send e-mail in an automated fashion. Systems such as web-based shopping carts send order confirmations by e-mail. Unattended processing systems might e-mail error conditions to an operator. Feedback gathered from visitors to a website may be mailed as well. In short, sending e-mail is a common task.

There are different ways of sending messages out on different platforms. Linux and UNIX programmers will use `/usr/sbin/sendmail`, for instance. Other platforms provide mail transmission services as well.

If you have a mail server available somewhere on your network, perhaps even on the same machine running your application, then you have another choice. Instead of using a platform-specific application to send your message, you can connect to the mail server and send it directly. That's what this chapter is about.

Besides communicating with mail servers on your local machine or network, you can also use SMTP to communicate with remote servers on the Internet. In most cases, you'll have no need to do this unless you're actually writing a mail server in Python.

Introducing the SMTP Library

Python's SMTP implementation is found in `smtplib`. The `smtplib` module makes it easy to do simple tasks with SMTP. In the examples that follow, the programs will take several command-line arguments: the name of a SMTP server, a sender address, and one or more recipient addresses.

If you don't know where to find an SMTP server, you might try using localhost. Many UNIX, Linux, and Mac OS X systems have an SMTP server listening for connections from the local machine. Otherwise, consult your network administrator or Internet provider to obtain proper values. Note that you usually cannot just pick a mail server at random; many only store or forward mail from certain authorized clients.

Here's a simple SMTP program:

```
#!/usr/bin/env python
# Basic SMTP transmission - Chapter 10 - simple.py

import sys, smtplib

if len(sys.argv) < 4:
    print "Syntax: %s server fromaddr toaddr [toaddr...]" % sys.argv[0]
    sys.exit(255)

server = sys.argv[1]
fromaddr = sys.argv[2]
toaddrs = sys.argv[3:]

message = """To: %s
From: %s
Subject: Test Message from simple.py

Hello,

This is a test message sent to you from simple.py and smtplib.
""" % (''.join(toaddrs), fromaddr)

s = smtplib.SMTP(server)
s.sendmail(fromaddr, toaddrs, message)

print "Message successfully sent to %d recipient(s)" % len(toaddrs)
```

This program is quite simple. It starts by generating a simple message based on the command-line arguments (for more advanced message generation, including attachments, see Chapter 9). Then it creates an `smtplib.SMTP` object connected to the specified server. Next, all that's required is a call to `sendmail()`. If that returns successfully, you know that the message was sent.

NOTE Note that the only thing that determines who gets the message is the list of recipients passed to `sendmail()`. Even if the `To` and `Cc` headers in the message contain different values, the only people who actually receive the message are those listed in the call to `sendmail()`. That is, the message headers are totally irrelevant to the delivery process. Please see Chapter 9 for more details.

Here's an example of how to run this program:

```
$ ./simple.py localhost sender@example.com recipient@example.com
Message successfully sent to 2 recipient(s)
```

TIP Make sure that you use valid sender and receiver addresses when running this program. Some servers or junk mail filters could intercept messages with invalid addresses and silently drop them. They may not always return an error condition, so as the sending program, you may not receive notification of a problem.

Error Handling and Conversation Debugging

There are several different errors that may be raised while you're programming with `smtplib`. They are

- `socket.gaierror` for errors looking up address information
- `socket.error` for general I/O and communication problems
- `socket.herror` for other addressing errors
- `smtplib.SMTPException` or a subclass of it for SMTP conversation problems

The first three errors are covered in more detail in Chapter 2; they're passed straight through the `smtplib` module to your program. If there's a problem with your communication to the server (for instance, an e-mail address is bad), you'll get an `smtplib.SMTPException`.

The `smtplib` module also provides a way to discover what's going on under the hood. Calling `smtpobj.set_debuglevel(1)` will enable that level of detail. With

this option, you should be able to track down any problems. Here's an example program that provides basic error handling and debugging:

```
#!/usr/bin/env python
# SMTP transmission with debugging - Chapter 10 - debug.py

import sys, smtplib, socket

if len(sys.argv) < 4:
    print "Syntax: %s server fromaddr toaddr [toaddr...]" % sys.argv[0]
    sys.exit(255)

server = sys.argv[1]
fromaddr = sys.argv[2]
toaddrs = sys.argv[3:]

message = """To: %s
From: %s
Subject: Test Message from simple.py

Hello,

This is a test message sent to you from simple.py and smtplib.
""" % (''.join(toaddrs), fromaddr)

try:
    s = smtplib.SMTP(server)
    s.set_debuglevel(1)
    s.sendmail(fromaddr, toaddrs, message)
except (socket.gaierror, socket.error, socket.herror, smtplib.SMTPException), \
        e:
    print " *** Your message may not have been sent!"
    print e
    sys.exit(1)
else:
    print "Message successfully sent to %d recipient(s)" % len(toaddrs)
```

This program looks similar to the last. However, the output will be very different. Here's an example of the output:

```
$ ./debug.py localhost foo@example.com jgoerzen@complete.org
send: 'ehlo localhost\r\n'
reply: '250-localhost\r\n'
reply: '250-PIPELINING\r\n'
reply: '250-SIZE 20480000\r\n'
reply: '250-VRFY\r\n'
reply: '250-ETRN\r\n'
reply: '250-STARTTLS\r\n'
reply: '250-XVERP\r\n'
reply: '250 8BITMIME\r\n'
reply: retcode (250); Msg: localhost
PIPELINING
SIZE 20480000
VRFY
ETRN
STARTTLS
XVERP
8BITMIME
send: 'mail FROM:<foo@example.com> size=157\r\n'
reply: '250 Ok\r\n'
reply: retcode (250); Msg: Ok
send: 'rcpt TO:<jgoerzen@complete.org>\r\n'
reply: '250 Ok\r\n'
reply: retcode (250); Msg: Ok
send: 'data\r\n'
reply: '354 End data with <CR><LF>.<CR><LF>\r\n'
reply: retcode (354); Msg: End data with <CR><LF>.<CR><LF>
data: (354, 'End data with <CR><LF>.<CR><LF>')
send: 'To: jgoerzen@complete.org\r\n'
From: foo@example.com\r\n
Subject: Test Message from simple.py\r\n
\r\n
Hello,\r\n
\r\n
This is a test message sent to you from simple.py and smtplib.
\r\n.
\r\n'
reply: '250 Ok: queued as 8094C18C0\r\n'
reply: retcode (250); Msg: Ok: queued as 8094C18C0
data: (250, 'Ok: queued as 8094C18C0')
Message successfully sent to 1 recipient(s)
```

From this example, you can see the conversation that `smtplib` is having with the SMTP server over the network. As you implement code that uses more advanced SMTP features, this will be more important. Let's look at what's happening.

First, the client (`smtplib`) sends an `EHLO` command with your hostname in it. The remote server responds with its hostname, and details about optional features that it supports. Next, the client sends the `mail from` command, which indicates the e-mail address of the sender and the size of the message. The server has the opportunity to reject the message here (for instance, because it thinks you're a spammer); but in this case, it responds with `250 0k`. (Note that in this case, the code `250` is what matters; the remaining text is just an explanation and varies from server to server.)

Then the client sends a `rcpt to` command. If you were sending the message to more than one recipient, each one would be listed on the `rcpt to` line. Finally, the client sends a `data` command, transmits the actual message, and finishes the conversation.

The `smtplib` module is doing all this automatically for you in this example. In the rest of the chapter, you'll look at ways to take more control of the process so you can take advantage of some more advanced features.

CAUTION Don't get a false sense of confidence because no error was detected. In many cases, a mail server may accept a message, only to have delivery fail later. For instance, many companies have a special server that accepts mail from the outside, then forwards it to the appropriate server inside the company's firewall. That server may reject the message, even though the first accepted it. In that case, a bounce message *should* be sent to the sender of the message, even though the message was originally sent successfully.

Getting Information from EHLO

Sometimes it's nice to know information about what kind of messages a remote SMTP server will accept. For instance, most SMTP servers have a limit on the sizes of messages they permit. If you don't check what size messages are acceptable, you may transmit a very large message only to find that it's being rejected at the end of your transmission. If you're on a slow dial-up link, this could translate into over an hour of wasted time. Fortunately, it's possible to check on the maximum message size in advance.

In the original version of SMTP, a client would send a `HELO` command as the initial greeting to the server. A set of extensions to SMTP, called ESMTP, has been developed to allow more powerful conversations. ESMTP-aware clients will begin

the conversation with EHLO, which signals an ESMTP-aware server to send extended information. This extended information includes, among other things, the maximum message size.

Most modern mail servers support EHLO. In response to the initial EHLO command, the server will return information about optional SMTP features that it supports.

However, you must be careful to check the return code. Some servers don't support ESMTP. On those servers, EHLO will return an error. In this case, you must send a HELO command. Unlike the previous examples, if you attempt to use any EHLO or HELO command, `sendmail()` will no longer attempt to send these commands automatically. Here's an example that gets the maximum size from the server and returns an error before sending a message if it's too large:

```
#!/usr/bin/env python
# SMTP transmission with manual EHLO - Chapter 10 - ehlo.py

import sys, smtplib, socket

if len(sys.argv) < 4:
    print "Syntax: %s server fromaddr toaddr [toaddr...]" % sys.argv[0]
    sys.exit(255)

server = sys.argv[1]
fromaddr = sys.argv[2]
toaddrs = sys.argv[3:]

message = """To: %s
From: %s
Subject: Test Message from simple.py

Hello,

This is a test message sent to you from simple.py and smtplib.
""" % (''.join(toaddrs), fromaddr)

try:
    s = smtplib.SMTP(server)
    code = s.ehlo()[0]
    usesesmtplib = 1
    if not (200 <= code <= 299):
        usesesmtplib = 0
        code = s.helo()[0]
        if not (200 <= code <= 299):
            raise SMTPHeloError(code, resp)
```

```

if usesesmtplib and s.has_extn('size'):
    print "Maximum message size is", s.esmtp_features['size']
    if len(message) > int(s.esmtp_features['size']):
        print "Message too large; aborting."
        sys.exit(2)
    s.sendmail(fromaddr, toaddrs, message)
except (socket.gaierror, socket.error, socket.herror, smtplib.SMTPException), \
    e:
    print " *** Your message may not have been sent!"
    print e
    sys.exit(1)
else:
    print "Message successfully sent to %d recipient(s)" % len(toaddrs)

```

If you run this program, and your server provides its maximum message size, the program will display that on your screen and verify that its message doesn't exceed that size before sending. This can at times save a lot of time transmitting a large message that would just be rejected anyway. Here's what running this program might look like:

```
$ ./ehlo.py localhost foo@example.com jgoerzen@complete.org
Maximum message size is 10240000
Message successfully sent to 1 recipient(s)
```

Take a look at the portions of the code that verify the result from a call to `ehlo()` or `helo()`. Those two functions return a list—the first item in the list is a numeric result code from the SMTP server. Results between 200 and 299, inclusive, indicate success; everything else indicates a failure. Therefore, if the result is within that range, you know that the server processed the message properly.

CAUTION The same caution as before applies here: The fact that the first SMTP server accepts the message doesn't mean that it will actually be delivered; a later server may have a more restrictive maximum size.

Besides message size, other ESMTP information is available as well. For instance, some servers may accept data in raw 8-bit mode if they provide the 8BITMIME capability. Others may support encryption as described in the next section. For more on ESMTP and its capabilities, which may vary from server to server, consult RFC1869 or your server's documentation.

Using Secure Sockets Layer and Transport Layer Security

SMTP conversations can be encrypted and authenticated by SSL/TLS. In this section, you can learn about how SSL/TLS fits in with SMTP conversations. For more details on TLS, please see Chapter 15; the code presented in this chapter isn't completely secure without the certificate handling described there.

The general procedure for using TLS in SMTP is as follows:

1. Create the SMTP object like usual.
2. Send the EHLO command. If the remote server doesn't support EHLO, it will not support TLS.
3. Check `s.has_extn()` to see if `starttls` is present. If not, the remote server doesn't support TLS and the message should be sent like normal (or an error will be generated, depending on your requirements).
4. Call `starttls()` to initiate the encrypted channel.
5. Call `ehlo()` a second time; this time, it's encrypted.
6. Finally, send your message as usual.

The first question you have to ask yourself when working with TLS is what constitutes an error? Depending on your application, it could be any of the following:

- No support for TLS on the remote side
- Remote fails to establish TLS session properly
- Remote presents a certificate that cannot be validated

Let's step through each of these scenarios and see when they may be appropriate.

First, it's sometimes appropriate to treat a lack of support for TLS altogether as an error. This could be the case if you're writing an application that speaks to only a limited set of mail servers—perhaps those mail servers run by your company that you know support TLS or mail servers run by a bank that you know supports TLS. Since only a minority of mail servers on the Internet today supports TLS, a general-purpose mail program should not treat this as an error. Many TLS-aware SMTP clients will use TLS if available, but they'll fall back on standard unsecured

transmission otherwise. This is known as *opportunistic encryption* and is less secure than forcing all communications to be encrypted. However, it still presents security benefits over completely unencrypted communications.

The second example represents the case in which a remote server claims to be TLS-aware but then fails to properly establish a TLS connection. This is often due to a misconfiguration on the server's end. For maximum compatibility, you may wish to retry it with an unencrypted connection. Though this situation is rare, it does happen due to the low percentage of communications that are encrypted today.

The third example represents a case in which you cannot completely authenticate the remote server. For a complete discussion on peer validation, please see Chapter 15. If you're exchanging mail only with trusted servers, this could be a problem, but for a general-purpose client, it probably merits a warning rather than an error.

The following example will act as a general-purpose client. It will connect to a server and use TLS if possible; otherwise, it will fall back and send the message as usual.

```
#!/usr/bin/env python
# SMTP transmission with TLS - Chapter 10 - tls.py

import sys, smtplib, socket

if len(sys.argv) < 4:
    print "Syntax: %s server fromaddr toaddr [toaddr...]" % sys.argv[0]
    sys.exit(255)

server = sys.argv[1]
fromaddr = sys.argv[2]
toaddrs = sys.argv[3:]

message = """To: %s
From: %s
Subject: Test Message from simple.py

Hello,

This is a test message sent to you from simple.py and smtplib.
""" % (''.join(toaddrs), fromaddr)
```

```

try:
    s = smtplib.SMTP(server)
    code = s.ehlo()[0]
    usesesmtplib = 1
    if not (200 <= code <= 299):
        usesesmtplib = 0
        code = s.helo()[0]
        if not (200 <= code <= 299):
            raise SMTPHeloError(code, resp)

    if usesesmtplib and s.has_extn('starttls'):
        print "Negotiating TLS...."
        s.starttls()
        code = s.ehlo()[0]
        if not (200 <= code <= 299):
            print "Couldn't EHLO after STARTTLS"
            sys.exit(5)
        print "Using TLS connection."
    else:
        print "Server does not support TLS; using normal connection."
    s.sendmail(fromaddr, toaddrs, message)

except (socket.gaierror, socket.error, socket.herror, smtplib.SMTPException), \
       e:
    print " *** Your message may not have been sent!"
    print e
    sys.exit(1)
else:
    print "Message successfully sent to %d recipient(s)" % len(toaddrs)

```

If you run this program and give it a server that understands TLS, the output will look like this:

```

$ ./tls.py localhost jgoerzen@complete.org jgoerzen@complete.org
Negotiating TLS....
Using TLS connection.
Message successfully sent to 1 recipient(s)

```

The message was successfully sent using TLS. Notice that the call to `sendmail()` is the same regardless of whether TLS is used. Once TLS is started, the system hides that layer of complexity from you, so you need not worry about it.

Please note that this TLS example isn't fully secure because it doesn't perform certificate validation. Please see Chapter 15 for details on the many different ways to do that.

Authenticating

Some SMTP servers may require authentication before you're allowed to send messages. Most frequently, Internet providers that have large numbers of dial-up customers run these servers because their customers send mail through their servers. Therefore, you probably won't need to authenticate your SMTP connections frequently.

For maximum security, TLS should be used in conjunction with authentication. The proper way to do this is to establish the TLS connection first and then send your authentication information over the encrypted communications channel.

Using authentication is simple—`smtplib` provides a `login()` function that simply takes a username and a password. Here's an example:

```
#!/usr/bin/env python
# SMTP transmission with authentication - Chapter 10 - login.py

import sys, smtplib, socket
from getpass import getpass

if len(sys.argv) < 4:
    print "Syntax: %s server fromaddr toaddr [toaddr...]" % sys.argv[0]
    sys.exit(255)

server = sys.argv[1]
fromaddr = sys.argv[2]
toaddrs = sys.argv[3:]

message = """To: %s
From: %s
Subject: Test Message from simple.py

Hello,
```

This is a test message sent to you from simple.py and smtplib.
""" % (''.join(toaddrs), fromaddr)

```

sys.stdout.write("Enter username: ")
username = sys.stdin.readline().strip()
password = getpass("Enter password: ")

try:
    s = smtplib.SMTP(server)
    try:
        s.login(username, password)
    except smtplib.SMTPException, e:
        print "Authentication failed:", e
        sys.exit(1)
    s.sendmail(fromaddr, toaddrs, message)
except (socket.gaierror, socket.error, socket.herror, smtplib.SMTPException), \
    e:
    print " *** Your message may not have been sent!"
    print e
    sys.exit(2)
else:
    print "Message successfully sent to %d recipient(s)" % len(toaddrs)

```

Most servers don't support authentication; if you're using a server that doesn't support authentication, you'll receive an "Authentication failed" error message. You can prevent that by using `s.has_extn('auth')` after calling `s.ehlo()`.

You can run this program just like the previous examples. If you run it with a server that does support authentication, you'll be prompted for a username and password. If they're accepted, your message will be transmitted just like in the previous examples.

SMTP Tips

Here are some tips to help you implement SMTP clients:

- There's no way to guarantee that a message was delivered with SMTP. You can sometimes be guaranteed that it was *not* delivered, but lack of an error doesn't mean that it was delivered.
- The `sendmail()` function raises an exception if any of the recipients failed, though the message may still have been sent to other recipients. Check the exception for more details. If it's very important for you to know specifics on which addresses failed, you may need to call `sendmail()` individually for each. That isn't recommended, however, since it will cause the message body to be transmitted multiple times.

- SSL/TLS is insecure without certificate verification; the `starttls()` function takes some of the same arguments as `socket.ssl()`, which is described in Chapter 15.
- Many free e-mail providers such as Yahoo! and Hotmail limit message sizes to 500 KB or 1 MB; messages larger than that may run into size restrictions. To conserve bandwidth, use the `EHL0` method of checking the remote's maximum message size before transmitting.
- Python's `smtplib` isn't meant to be a general-purpose mail relay; rather, you should use it to send messages to an SMTP server close to you that will handle the actual delivery of mail.

Summary

SMTP is used to transmit e-mail messages to mail servers. Python provides the `smtplib` module for SMTP clients to use. By calling the `sendmail()` method of SMTP objects, you can transmit messages. The sole way of specifying the actual recipients of a message is with parameters to `sendmail()`; the message headers don't specify the actual recipients.

Several different exceptions could be raised during an SMTP conversation. Interactive programs should check for and handle them appropriately.

ESMTP is an extension to SMTP. It permits you to discover the maximum message size supported by a remote SMTP server prior to transmitting a message.

ESMTP also permits TLS, which is a way to encrypt your conversation with a remote server. Fundamentals of TLS are covered in Chapter 15.

Some SMTP servers require authentication; you can authenticate with the `login()` method.

SMTP doesn't provide functions for downloading messages from a mailbox to your own computer. To accomplish that, you'll need the protocols discussed in the next two chapters. POP, discussed in Chapter 11, is a simple way to download messages. IMAP, discussed in Chapter 12, is a more capable and powerful protocol.

CHAPTER 11

POP

POP, THE POST OFFICE PROTOCOL, is a simple protocol that's used to download e-mail from a storage area on a mail server. With POP, you can download mail from an Internet provider. The downloaded messages could then be presented to the user with a mail reader. Or, you could process these messages—perhaps implementing a mail filter.

The most common implementation of POP is known as version 3, and is commonly referred to as POP3. Because version 3 is so dominant, the terms POP and POP3 are practically interchangeable today.

POP's chief benefit—and also its largest weakness—is its simplicity. If you simply need to access a remote inbox, download the new mail from there, and maybe delete the mail after you've downloaded it, POP is perfect for you. You'll be able to accomplish this task quickly, without complex code.

However, POP does have some feature limitations. It doesn't support multiple mailboxes on the remote side, nor does it provide any reliable persistent message identification. This means that you cannot use POP as a protocol for mail synchronization (whereby a local mailbox is kept in sync with a mailbox on the server). If you need these features, you should check out the Internet Message Access Protocol (IMAP), which is covered in Chapter 12.

Python provides a module named `poplib`, which provides a convenient interface for using POP. In this chapter, you'll learn how to use `poplib` to connect to a POP server, gather summary information about a mailbox, download messages, and delete messages from the server. This covers *all* standard POP features.

Compatibility Between POP Servers

POP servers are often notoriously bad at correctly following standards. Standards also don't exist for some behaviors, so these details are left up to the server authors. Basic operations will generally be fine. But certain behaviors can vary from server to server. For instance, some servers will mark all messages as read whenever you connect to the server. Others will mark a given message as read only when it's downloaded. Some will never mark it as read at all. Keep this in mind as you read this chapter.

Connecting and Authenticating

POP supports multiple authentication methods. The two most common are basic username and password authentication and APOP, which is an optional extension to POP that helps prevent attackers from stealing passwords transmitted in cleartext. The process of connecting and authenticating to a remote server looks like this:

1. Create a `POP3` object, and pass the remote hostname and port to it.
2. Call `user()` and `pass_()` to send the username and password. Note the underscore in `pass_()`. It's present because `pass` is a keyword in Python and can't be used for a method name.
3. If `poplib.error_proto` is raised, the login has failed and the string associated with the exception contains the explanatory text sent by the server.

Here's an example that uses these steps to log in to a remote POP server. Once connected, it calls `stat()`, which returns a simple tuple of the number of messages in the mailbox and the messages' total size. Finally, it calls `quit()`, which closes the POP connection, as shown here:

```
#!/usr/bin/env python
# POP connection and authentication - Chapter 11 - popconn.py

import getpass, poplib, sys
(host, user) = sys.argv[1:]
passwd = getpass.getpass()

p = poplib.POP3(host)
try:
    p.user(user)
    p.pass_(passwd)
except poplib.error_proto, e:
    print "Login failed:", e
    sys.exit(1)
status = p.stat()
print "Mailbox has %d messages for a total of %d bytes" % (status[0],
    status[1])
p.quit()
```

This program attempts standard authentication and displays a brief message about the mailbox if authentication succeeds. If the authentication fails, the program displays an error message and terminates.

You can test this program if you have a POP account somewhere (most people do). Give it the name of your POP server and user login name on the command line (if you don't know this information, contact your Internet provider or network administrator). You'll be prompted for your password. The program will not download or alter any mail.

CAUTION While this program doesn't alter any messages, some POP servers will nonetheless alter mailbox flags. Running the examples in this chapter against a live mailbox could cause you to lose information about which messages are read, unread, new, or old. Unfortunately, that behavior is server-dependent and beyond the control of POP clients. I strongly recommend running these examples against a test mailbox rather than your live mailbox.

Here's an example invocation:

```
$ ./popconn.py pop.example.com jgoerzen
Password:
Mailbox has 6 messages for a total of 75202 bytes
```

Many servers support or require APOP authentication. APOP uses a simple bit of cryptography to make it difficult for network sniffers to grab your password. While this does help security somewhat, it isn't a complete solution. Many people use IMAP instead of POP. Unlike POP, IMAP is widely deployed with SSL support.

While many POP servers support or require APOP, many other servers don't understand APOP at all. Therefore, most POP clients will attempt APOP authentication first, and if it fails, fall back to standard authentication. Here's a version of the previous example, modified to attempt APOP and to fall back to standard authentication if it fails.

```
#!/usr/bin/env python
# POP connection and authentication with APOP - Chapter 11 - apop.py

import getpass, poplib, sys
(host, user) = sys.argv[1:]
passwd = getpass.getpass()
```

```

p = poplib.POP3(host)
try:
    print "Attempting APOP authentication..."
    p.apop(user, passwd)
except poplib.error_proto:
    print "Attempting standard authentication..."
    try:
        p.user(user)
        p.pass_(passwd)
    except poplib.error_proto, e:
        print "Login failed:", e
        sys.exit(1)

status = p.stat()
print "Mailbox has %d messages for a total of %d bytes" % (status[0],
    status[1])
p.quit()

```

In this case, if the `apop()` call raises an error (either because the server doesn't support APOP or the login failed), the standard authentication is attempted. If that also fails, then the login error is displayed.

As soon as a login succeeds by whatever method, most POP servers will lock the mailbox. That means that no mail may be delivered, and no alterations may be made, as long as your POP connection persists or until `quit()` is called.

You can run this example in the same manner as the first one. If your POP server supports APOP, you'll log in with APOP; otherwise, you'll use standard authentication.

Semantics of Locking and `quit()`

Different POP servers interpret locking in different ways. Some POP servers will not perform any locking at all. Others may block only POP readers, but still may permit delivery. Still others block everything while a POP session is in progress.

Some POP servers don't properly detect errors and will keep a box locked indefinitely if you don't call `quit()`. At one time, the world's most popular POP server fell into this category. To prevent hassles for your users and administrators, it's vital to always call `quit()` when finishing up a POP session. A `try...finally` block or `atexit` handler could be useful for you.

Obtaining Mailbox Information

The preceding example shows you `stat()`, which returns the number of messages in the mailbox and the total size of them. Another useful command is `list()`, which returns more detailed information about each message. The most interesting part is the message number, which is required to retrieve messages later. Note that there may be gaps in message numbers: For instance, a mailbox may contain message numbers 1, 2, 5, 6, and 9. Also, the number assigned to a particular message may be different for each different connection to the POP server. Here's a program that uses `list()` to display information about each message:

```
#!/usr/bin/env python
# POP mailbox scanning - Chapter 11 - mailbox.py

import getpass, poplib, sys
(host, user) = sys.argv[1:]
passwd = getpass.getpass()

p = poplib.POP3(host)
try:
    p.user(user)
    p.pass_(passwd)
except poplib.error_proto, e:
    print "Login failed:", e
    sys.exit(1)
status = p.stat()
print "Mailbox has %d messages for a total of %d bytes" % (status[0],
    status[1])
for item in p.list()[1]:
    number, octets = item.split(' ')
    print "Message %s: %s bytes" % (number, octets)
p.quit()
```

The `list()` function returns a tuple containing two items. The first item is a response code, which you can usually ignore (`list()` raises exceptions on errors). The second item is a list of strings. That's why this program says `p.list()[1]`—this is grabbing the second item.

Each string in the list contains two items, separated by a space: the message number and the message size in bytes. The previous example uses `split()` to retrieve the individual components. Here's what the output might look like:

```
$ ./mailbox.py popserver.example.com testuser
Password:
Mailbox has 2 messages for a total of 4703 bytes
Message 1: 3339 bytes
Message 2: 1364 bytes
```

Downloading Messages

The `poplib` function `retr()` is used to download a message. It downloads exactly one message at a time. You must pass it the message number you're looking for. After transmitting the message, most—but not all—POP servers will set the "seen" flag for the message. Unfortunately, the POP standard doesn't provide a way to use that flag, so this only matters for non-POP mail readers' access to the message store.

Here's an example program that will download all the messages from the server and leave them there. It takes a filename, which it will create, and it will write the messages to the file in standard UNIX mbox format. Note, though, that to work with mboxes on live systems, you'll need additional file locking and safeguard code that isn't included here.

The mbox Format and Locking

The mbox format stores many e-mail messages in a single file. This makes it easy to write to and simple to manage. However, dealing with concurrent accesses is tricky. For instance, if a POP reader is deleting a message while two new messages are arriving, the three programs must synchronize access among themselves to prevent data corruption.

Traditionally, on UNIX and Linux platforms, the `flock()` or `fcntl()` calls would be used to perform locking. However, on many implementations of the Network File System (NFS), such locking doesn't propagate across system boundaries. Many large UNIX shops use NFS for their mail spool.

Various other standards exist for locking mboxes, and they differ from program to program. To address all these concerns, other mailbox formats such as Maildir exist. Maildir stores each message in a file in a specially managed directory. This directory is designed so that no locking is necessary. However, the Maildir specification makes use of colons in filenames. Microsoft operating systems prohibit colons in filenames, so Maildirs are unavailable on Windows. To provide the most portable example code, these examples use mbox because they run on Windows as well.

```

#!/usr/bin/env python
# POP mailbox downloader - Chapter 11 - download.py

import getpass, poplib, sys, email
(host, user, dest) = sys.argv[1:]
passwd = getpass.getpass()

# Open a mailbox for appending.
destfd = open(dest, "at")

# Log in like usual.
p = poplib.POP3(host)
try:
    p.user(user)
    p.pass_(passwd)
except poplib.error_proto, e:
    print "Login failed:", e
    sys.exit(1)

# Iterate over the list of messages in the mailbox
for item in p.list()[1]:
    number, octets = item.split(' ')
    print "Downloading message %s (%s bytes)" % (number, octets)

    # Retrieve the message (storing it in a list of lines)
    lines = p.retr(number)[1]

    # Create an e-mail object representing the message
    msg = email.message_from_string("\n".join(lines))

    # Write it out to the mailbox
    destfd.write(msg.as_string(unixfrom = 1))

    # Make sure there's an extra newline separating messages
    destfd.write("\n")

# Log out and close file descriptors
p.quit()
destfd.close()

```

The return value from `retr()` is a little strange. It returns a tuple consisting of the result code and the message. But the message isn't formatted as a string; it's given to you as a list of strings where each element in the list represents one line

of the message. Fortunately, you can easily convert this to a standard string with `"\n".join(lines)`.

This program takes three command-line parameters. The first two are the same as you saw in previous examples: the name of your POP server and a username. The third argument gives the name of a file to be used as a mailbox and will be created if it doesn't already exist. Here's an example:

```
$ ./download.py pop.example.com jgoerzen testmbox
Password:
.
    Downloading message 1 (17488 bytes)
    Downloading message 2 (44402 bytes)
    Downloading message 3 (2605 bytes)
    Downloading message 4 (2611 bytes)
```

Deleting Messages

The preceding example never deletes any messages from the server. They will continue to pile up on the server, being re-downloaded each time. This corresponds to the “leave mail on server” behavior in many POP clients.

In many cases, you'll want to delete the messages after downloading them. There is a `dele()` function to do just that. This sends the POP `DELE` command, which, in most cases, flags the messages for deletion. Most POP servers don't actually delete them until you call `quit()`. However, certain servers delete the messages immediately, so this behavior cannot be relied upon.

The `dele()` command takes only one message number, so it must be called once for each message to be deleted. I recommend that you only call `dele()` after you're sure the message has been successfully processed or written to disk. In Python, when writing to a regular file, you know this is the case if no call to `write()` or `close()` raises an exception. Therefore, in the following example, the system remembers which messages were downloaded, closes the mailbox, and then issues the `dele()` commands.

CAUTION Do not run this example against your main mailbox. It will delete messages! You should only run this example against a mailbox you do not care about.

```

#!/usr/bin/env python
# POP mailbox downloader with deletion - Chapter 11
# download-and-delete.py
#####
# WARNING: This program deletes mail from the specified mailbox.
#           DO NOT point it to any mailbox you care about!
#####

import getpass, poplib, sys, email

def log(text):
    """Simple function to write status information"""
    sys.stdout.write(text)
    sys.stdout.flush()

(host, user, dest) = sys.argv[1:]
passwd = getpass.getpass()

# Open a mailbox for appending
destfd = open(dest, "at")

log("Connecting to %s...\n" % host)
p = poplib.POP3(host)
try:
    log("Logging on...")
    p.user(user)
    p.pass_(passwd)
    log(" success.\n")
except poplib.error_proto, e:
    print "Login failed:", e
    sys.exit(1)

# Load list of messages present in mailbox
log("Scanning INBOX...")
mblist = p.list()[1]
log("%d messages.\n" % len(mblist))

dellist = []

# Iterate over the list of messages in the mailbox
for item in mblist:
    number, octets = item.split(' ')
    log("Downloading message %s (%s bytes)... " % (number, octets))
    # Download message
    # Delete message
    dellist.append(item)

```

```
# Retrieve the message (storing it in a list of lines)
lines = p.retr(number)[1]

# Create an e-mail object representing the message
msg = email.message_from_string("\n".join(lines))

# Write it out to the mailbox
destfd.write(msg.as_string(unixfrom = 1))

# Make sure there's an extra newline separating messages
destfd.write("\n")

# Add it to the list of messages to delete later
dellist.append(number)

log(" done.\n")

# Close the mailbox
destfd.close()

counter = 0      # Just a convenience for the status messages

# Iterate over the list of messages to delete
for number in dellist:
    counter += 1
    log("Deleting message %d of %d\r" % (counter, len(dellist)))

    # Delete the message.
    p.dele(number)

# Display summary information
if counter > 0:
    log("Successfully deleted %d messages from server.\n" % counter)
else:
    log("No messages present to download.\n")

log("Closing connection... ")

# Log out
p.quit()
log(" done.\n")
```

If you run this program, you'll see output similar to this:

```
$ ./download-and-delete.py popserver testuser /tmp/mailbox
Password:
Connecting to popserver...
Logging on... success.
Scanning INBOX... 5 messages.
Downloading message 1... done.
Downloading message 2... done.
Downloading message 3... done.
Downloading message 4... done.
Downloading message 5... done.
Successfully deleted 5 messages from server.
Closing connection... done.
```

You can then use a program that understands mbox files to examine the /tmp/mailbox file.

Summary

POP, the Post Office Protocol, provides a simple way to download e-mail messages stored on a remote server. With Python's `poplib` interface, you can obtain information about the number of messages in a user's inbox and the size of each message. You can also retrieve or delete individual messages by number.

Connecting to a POP server may lock a mailbox. Therefore, it's important to try to keep POP sessions as brief as possible and always call `quit()` when done.

POP normally transmits authentication information in the clear (without using encryption). The APOP feature allows for the encryption of passwords. Some servers mandate APOP usage, while others don't support it. The usual method of authenticating is to try APOP and fall back on standard authentication if it fails.

Although POP is a simple and widely deployed protocol, it has a number of drawbacks that make it unsuitable for some applications. For instance, it can access only one folder and doesn't provide persistent tracking of individual messages. The next chapter discusses IMAP, a protocol that provides the features of POP with a number of new features as well.

CHAPTER 12

IMAP

IMAP, the Internet Message Access Protocol, is similar in concept to the POP Protocol described in Chapter 11. IMAP, however, is a much more robust and powerful protocol. In addition to supporting all the features of POP, IMAP also supports the following:

- Multiple mail folders, not just the user's inbox.
- Storage of flags (read, replied, seen, deleted) on the IMAP server, reading of stored flags, and sharing of these flags with mail readers.
- Server-side searching for messages. With IMAP, you don't need to download messages in order to search them.
- Server-side copying and moving of messages between folders.
- Ability to add new messages to a remote folder.
- Persistent unique message numbering, making possible synchronization with a server, client-side message filtering (letting you delete the appropriate message from the server later), and multithreaded clients.
- Shared and read-only folders.
- Some IMAP servers can present nonmail sources (such as Usenet news) as mail, which clients can specifically request.
- Some IMAP servers support storage of mail in nonstandard locations, which clients can specifically request.
- IMAP clients can selectively download certain parts of a message—for instance, only a particular attachment or only the message headers.

These features, taken together, mean that IMAP can be used as much more than the simple download-and-delete protocol that POP is. For instance, many mail readers (such as Mozilla or Outlook) that support IMAP present an IMAP folder exactly like a local folder. When a user clicks on a message, the mail reader

downloads it from the IMAP server right then, rather than downloading all messages in advance, and the reader also sets the server's read flag immediately.

IMAP clients can also synchronize themselves with an IMAP server. For instance, someone about to leave on a business trip might download an IMAP folder to a laptop. Then, on the road, mail might be read, deleted, or replied to. The user's mail program would record these actions and store them up to execute later. When the laptop is connected back up to the network, the messages that were deleted on the laptop can be deleted on the server, saving the effort of having to delete them twice. The mail reader could also mark messages read on the laptop as read on the server as well. That way, when the person uses a normal desktop computer to check mail on the same IMAP server, the exact same messages with the same flags will be visible.

The net result is one of IMAP's biggest advantages over POP: Users can see the same mail in the same state from multiple machines at any time. POP users must face either seeing the same mail multiple times (if their clients are set to leave mail on a server), or they'll see only certain messages on certain machines (if the clients delete the mail). IMAP can solve both of these problems.

IMAP can also be used in exactly the same manner as POP for those who don't want or need its advanced features.

There are several versions of the IMAP Protocol available. The most recent, and by far the most popular, is known as IMAP4rev1. In fact, it's so much more popular than the older versions that "IMAP" is generally synonymous with IMAP4rev1. This chapter assumes that IMAP servers are IMAP4rev1 servers. Very old IMAP servers, which are quite uncommon, may not support all features discussed in this chapter.

In this chapter you'll learn how to use IMAP from Python. You'll learn about error handling, and how to work with folders by obtaining messages and information about them, searching through them, and moving messages between them.

Understanding IMAP in Python

Python's standard library includes an `imaplib` module. This module is very similar to the `poplib` module covered in Chapter 11. That's good if you're writing a client that uses IMAP in a way similar to POP.

However, if you're writing a more complex client to take advantage of some of IMAP's more powerful features, `imaplib` isn't so great. This is primarily because it leaves a good deal of parsing to you, the client programmer, so your programs can get large and complex.

There's another IMAP library for Python that's provided as part of the Twisted project (www.twistedmatrix.com). Twisted is a framework for writing network applications in Python. It's designed to be multitasking and uses asynchronous I/O throughout the entire library. You'll learn what that means later, but for now, suffice it to say that Twisted's IMAP library is very good, but it's also most likely very different from any other networking library you've ever worked with before. Since you're interested in IMAP, I'll assume that you want a library that handles its features well. Therefore, this chapter will cover the Twisted IMAP library.

Like HTTP, IMAP is a large and complex protocol, and this chapter will not cover every possible command and feature. The 108-page document describing IMAP is known as RFC3501 and is available from <ftp://ftp.rfc-editor.org/in-notes/rfc3501.txt>. I recommend that you start with this chapter, and consult the RFC if you need additional information. Also, the Twisted API documentation at www.twistedmatrix.com provides reference (but little how-to) information about IMAP. IMAP specifically is discussed at www.twistedmatrix.com/documents/TwistedDocs/current/api/twisted.protocols.imap4.html.

Introducing IMAP in Twisted

Most client networking libraries—`poplib` and `imaplib` included—work in pretty much the same way. You write code that calls library routines. The library makes calls to the server, and your code *blocks* (that is, the function doesn't return) until a result is retrieved from the network.

Twisted turns that approach on its head. When you access the network, you tell Twisted what function to pass the result to. The actual call you make returns immediately, and when a result is obtained from the network, your function is called with that result as a parameter. The Twisted developers call this paradigm "don't call us, we'll call you." This model is also known as *event-based* programming. The function that gets called with a result is known as a *callback function*.

Twisted does this because it allows you to effectively multitask without using multiple processes or threads. For client programming, this isn't as often an issue, but it can provide an easy and powerful way to improve your performance. By using multiple connections to a remote server, in some cases, you can achieve significant performance gains. Twisted is also frequently used for servers. Chapter 22 discusses the use of Twisted for server programming.

Twisted isn't a standard part of Python, but is available with a liberal no-royalty license. Your operating system may ship with it already; otherwise, it can be downloaded from the Twisted website. The examples in this chapter assume you have Twisted 1.1.0 or later installed.

Understanding Twisted Basics

Here's a basic Twisted program that will connect and retrieve the server's capability string. That string simply states what IMAP features the server supports.

```
#!/usr/bin/env python
# Basic connection with Twisted - Chapter 12 - tconn.py
# Note: This example assumes you have Twisted 1.1.0 or above installed.

from twisted.internet import defer, reactor, protocol
from twisted.protocols.imap4 import IMAP4Client
import sys

class IMAPClient(IMAP4Client):
    def connectionMade(self):
        print "I have successfully connected to the server!"
        d = self.getCapabilities()
        d.addCallback(self.gotcapabilities)

    def gotcapabilities(self, caps):
        if caps == None:
            print "Server did not return a capability list."
        else:
            for key, value in caps.items():
                print "%s: %s" % (key, str(value))

        # This is the last thing, so stop the reactor.
        self.logout()
        reactor.stop()

class IMAPFactory(protocol.ClientFactory):
    protocol = IMAPClient

    def clientConnectionFailed(self, connector, reason):
        print "Client connection failed:", reason
        reactor.stop()

reactor.connectTCP(sys.argv[1], 143, IMAPFactory())
reactor.run()
```

Most of the Twisted client programs you'll see are centered around two classes: a protocol class (`IMAPClient` in this example) and a factory class (`IMAPFactory`). The factory class manages the connection to the server. The protocol class implements the conversation with the server. In this case, you extend the built-in `IMAP4Client` class, and start your own logic with the `connectionMade()` method.

When the program starts, you first establish the network connection. The hostname is pulled from the command line, and the default IMAP port 143 is used. An `IMAPFactory` object is passed along as well.

The last line in the program is `reactor.run()`. In Twisted, the reactor is the component that handles network events. The call to `reactor.run()` doesn't return until something calls `reactor.stop()`.

The reactor handles the details of establishing the connection. When the connection is set up, it calls `connectionMade()`.

The program's logic begins in `connectionMade()`. It calls `getCapabilities()`, which is a simple command that returns a list of optional features supported by the IMAP server. Like virtually every call in Twisted, `getCapabilities()` returns an object known as a Deferred, which is the object at the center of the event-based model in Twisted. It's used to tell Twisted what to do when a reply is received.

Once the Deferred object is received, the program calls `addCallback()` on that object. This tells the reactor what function to call when the capability response is received from the server. In this case, it should call `gotcapabilities()`. That's the only method in this program that doesn't override another method in a base class. Note that the call to `addCallback()` doesn't use the customary parenthesis for `gotcapabilities()`. That's because you don't want to call `gotcapabilities()` at the time `addCallback()` is called; instead, you pass the function itself to `addCallback()` and let Twisted call it later.

The `gotcapabilities()` function receives an argument from Twisted—the result of the request you submitted with `getCapabilities()`. It prints out some data on the screen, then logs out and stops the reactor. This causes `reactor.run()` to return, and since it's the last line in the program, the program exits.

You might be thinking that this sounds like a very lengthy explanation for a very simple operation. It's a long explanation, but that's because Twisted works differently from libraries you're probably accustomed to. Don't worry; it's easy to follow Twisted code, and learning the patterns of developing with Twisted isn't hard.

To run the program, give it the name of your IMAP server on the command line. If you don't know the name of your server, contact your Internet provider or network support desk. Here's a sample output from this example program:

```
$ ./tconn.py imap.example.com
I have successfully connected to the server!
SORT: None
THREAD: ['ORDEREDSUBJECT', 'REFERENCES']
NAMESPACE: None
QUOTA: None
IMAP4rev1: None
UIDPLUS: None
IDLE: None
CHILDREN: None
```

You'll likely see a different set of capabilities than this one. RFC3501 defines the standard set of capability responses and should be consulted for the meaning of a particular response.

Logging In

Basic IMAP authentication means simply supplying a username and a password to the `login()` function in the protocol. Here's a program that does that, and takes the opportunity to illustrate multiple callbacks with a single `Deferred` object:

```
#!/usr/bin/env python
# Basic connection and authentication with Twisted - Chapter 12
# tlogin.py
# Note: This example assumes you have Twisted 1.1.0 or above installed.
#
# This program expects a hostname and a username as command-line
# arguments.
#
# Note: this program will hang if given a bad password.

from twisted.internet import defer, reactor, protocol
from twisted.protocols.imap4 import IMAP4Client
import sys, getpass

class IMAPClient(IMAP4Client):
    """Our IMAP protocol class. This class simply starts our IMAP logic when
    a connection is established."""
    def connectionMade(self):
        print "I have successfully connected to the server!"
```

```

# Create the IMAPLogic object.  It will add some of its own methods
# as callbacks.
IMAPLogic(self)
print "connectionMade returning."


class IMAPFactory(protocol.ClientFactory):
    """A Twisted factory class.  This class will take a username and password
when an instance is created, saving them for later use."""
protocol = IMAPClient

    def __init__(self, username, password):
        self.username = username
        self.password = password

    def clientConnectionFailed(self, connector, reason):
        print "Client connection failed:", reason
        reactor.stop()

class IMAPLogic:
    """This class implements the main logic for the program."""
    def __init__(self, proto):
        # Save off passed-in data for later use.
        self.proto = proto
        self.factory = proto.factory

        # Attempt to log in, giving the stored username and password.
        d = self.proto.login(self.factory.username, self.factory.password)

        # When logged in, call self.loggedin() and pass its result to
        # self.stopreactor().
        d.addCallback(self.loggedin)
        d.addCallback(self.stopreactor)

        # Tell the user what happened.
        print "IMAPLogic.__init__ returning."

    def loggedin(self, data):
        """Called after we are logged in.  The IMAP4Protocol login() function
will pass an argument containing None, so the data parameter is there
to receive it."""
        print "I'm logged in!"
        return self.logout()

```

```

def logout(self):
    """Call this to log out. Any arguments specified are ignored."""
    print "Logging out."
    d = self.proto.logout()
    return d

def stopreactor(self, data = None):
    """Call this to stop the reactor."""
    print "Stopping reactor."
    reactor.stop()

# Read the password from the terminal
password = getpass.getpass("Enter password for %s on %s: " % \
    (sys.argv[2], sys.argv[1]))

# Connect to the remote system
reactor.connectTCP(sys.argv[1], 143, IMAPFactory(sys.argv[2], password))

# And run the program.
reactor.run()

```

This example separates your own program's logic from the protocol class. This is a matter of taste; I prefer this approach, but you could just as easily continue extending the `IMAP4Client` class. Doing so would simply mean putting methods such as `loggedin()` in that class.

The `__init__()` method for `IMAPLogic` calls `addCallback()` twice for a `Deferred` object. This will normally cause the first callback to be called, then its result will be passed to the second callback. There's an added complication here: The first of those added callbacks, to `loggedin()`, returns a `Deferred` object on its own. When a callback function returns a `Deferred` object itself, it's treated specially. Twisted will not call the next callback on the "parent" `Deferred` until all callbacks on this one have been called—and then it will send the result of the last one to the next callback on the parent. Here `login()` winds up returning a `Deferred` with no callbacks. So this is what happens:

1. After receiving a positive acknowledgment from the server to a login request, `loggedin()` is called.
2. The `loggedin()` function eventually returns a `Deferred` object corresponding to a logout request. This object has no callbacks on it. It will effectively cause execution of the callbacks to be paused until the logout command is processed.
3. Finally, `stopreactor()` is called.

Let's trace the overall execution of this program. First, it prompts the user for a password. Then, it establishes the connection as before. Note, though, that the username and password get passed to the `IMAPFactory` object. In fact, the `__init__()` method simply saves those for later use.

When the connection is established, `connectionMade()` is called as before. However, this time it simply instantiates an `IMAPLogic` object.

The `__init__()` method of the `IMAPLogic` object gets called next. It saves the protocol object (`proto`) and the factory object (`proto.factory`), then calls the protocol object's `login()` method. Like most other Twisted methods, that method returns a `Deferred` object. A callback is added, so that once the user has been logged in, the `loggedin()` method is called. But then another callback is added: `self.stopreactor`. This means that the result of `loggedin()`—*or the last Deferred that it returns*—is passed to `stopreactor()`.

Note the code for `loggedin()`. It returns the value from `logout()`, which is, in turn, another `Deferred`.

In this case, there's no callback added to `logout()`. That means that when the `logout` operation is complete, its result is passed directly to `stopreactor()`, since that was the next callback on the “parent.”

Notice that the `stopreactor()` method is defined to take an optional argument `data`. You'll recall that callbacks are always passed the result of the previous call on the deferred “chain.” The `stopreactor()` function doesn't really care about this result, but it has to accept it; otherwise, Python will generate an error.

Here's what it looks like when you successfully run this program:

```
$ ./tlogin.py imap.example.com jgoerzen
Enter password for jgoerzen on christoph:
I have successfully connected to the server!
IMAPLogic.__init__ returning.
connectionMade returning.
I'm logged in!
Logging out.
Stopping reactor.
```

Notice how the `IMAPLogic.__init__()` method returns *before* the login is complete. This is how Twisted works; the `I'm logged in!` message isn't displayed until the response is received from the network and the reactor invokes the callback.

Error Handling

The previous example doesn't do any error handling at all. In fact, if you supply a bad username or password, you'll get an error message but the program will hang. The reason is that the callbacks are never called, and thus `reactor.stop()` is never called, so there's never any handling of errors going on.

Twisted doesn't raise exceptions like normal Python code would. That's because it can't; by the time an error occurs, control is already outside your code.

Instead, Twisted has a notion called *errback*. An errback is similar to a callback, but is invoked if an error occurs. Here's a program that illustrates the use of an errback. It adds a special handler for login failures. This handler will prompt the user to reenter a password when the login attempt fails. It also adds a generic handler for any other errors. Together, these will prevent the program from hanging if an error occurs, as shown here:

```
#!/usr/bin/env python
# Error handling Twisted - Chapter 12 - t-error.py
# Note: This example assumes you have Twisted 1.1.0 or above installed.
#
# This program expects a hostname and a username as command-line
# arguments.

from twisted.internet import defer, reactor, protocol
from twisted.protocols.imap4 import IMAP4Client
import sys, getpass

class IMAPClient(IMAP4Client):
    """Our IMAP protocol class. This class simply starts our IMAP logic when
    a connection is established."""
    def connectionMade(self):
        print "I have successfully connected to the server!"
        IMAPLogic(self)
        print "connectionMade returning."

class IMAPFactory(protocol.ClientFactory):
    """A Twisted factory class. This class will take a username and password
    when an instance is created, saving them for later use."""
    protocol = IMAPClient

    def __init__(self, username, password):
        self.username = username
        self.password = password

    def clientConnectionFailed(self, connector, reason):
        print "Client connection failed:", reason
        reactor.stop()
```

```

class IMAPLogic:
    """This class implements the main logic for the program."""
    def __init__(self, proto):
        self.proto = proto
        self.factory = proto.factory
        self.logentries = 1

        d = self.login()
        d.addCallback(self.loggedin)
        d.addErrback(self.loginerror)
        d.addCallback(self.logout)
        d.addCallback(self.stopreactor)

    # This is a generic catch-all handler—if any unhandled error
    # occurs, shut down the connection and terminate the reactor.
    d.addErrback(self.errorhappened)

    print "IMAPLogic.__init__ returning."

    def login(self):
        print "Logging in..."
        return self.proto.login(self.factory.username, self.factory.password)

    def loggedin(self, data):
        """Called after we are logged in. The IMAP4Protocol.login() function
        will pass an argument containing None, so the data parameter is there
        to receive it."""
        print "I'm logged in!"

    def logout(self, data = None):
        """Call this to log out. Any arguments specified are ignored."""
        print "Logging out."
        d = self.proto.logout()
        return d

    def stopreactor(self, data = None):
        """Call this to stop the reactor."""
        print "Stopping reactor."
        reactor.stop()

```

```

def errorhappened(self, failure):
    print "An error occurred:", failure.getErrorMessage()
    print "Because of the error, I am logging out and stopping reactor..."
    d = self.logout()
    d.addBoth(self.stopreactor)
    return failure

def loginerror(self, failure):
    print "Your login failed (attempt %d)." % self.logintries
    if self.logintries >= 3:
        print "You have tried to log in three times; I'm giving up."
        return failure
    self.logintries += 1

    # Prompt for new login info.
    sys.stdout.write("New username: ")
    self.factory.username = sys.stdin.readline().strip()
    self.factory.password = getpass.getpass("New password: ")

    # And try it again. Send errors back here.
    d = self.login()
    d.addErrback(self.loginerror)
    return d

password = getpass.getpass("Enter password for %s on %s: " % \
    (sys.argv[2], sys.argv[1]))
reactor.connectTCP(sys.argv[1], 143, IMAPFactory(sys.argv[2], password))
reactor.run()

```

Take a look at `IMAPLogic.__init__()`. Notice the two new calls to `addErrback()`. The first call occurs right after the `login` process. It means that any errors that occur up to that point will be sent to `loginerror()`. The second call occurs after everything else. Since it is last, it will catch *any* error that occurs in that whole chain of callbacks. It passes the error to the generic handler `errorhappened()`.

Now look at `loginerror()`. It's passed an object—a `Failure` object. If the program decides that it can't fix the problem (in this case, because there were too many tries), it can just return the error again. Twisted will pass it on to the next error handler; in this case, that means that it gets sent to `errorhappened()`.

However, usually `loginerror()` can handle the error. It asks the user for a new username and password and will retry the login. Notice that it adds itself as the errback from this new login—otherwise, if the new login attempt failed, it would go directly to `errorhappened()`. Finally, it returns the `Deferred`. Recall that this causes Twisted to wait for it before proceeding to the next callback in the parent. The

final result is that if the user is able to log in successfully with the new username and password, the rest of the code will have no idea that a failure ever occurred.

The `errorhappened()` function prints out the error message and then logs out and stops the reactor. Notice that it uses `addBoth()` after the call to `self.logout()`. This means that `self.stopreactor` will get called for both a success and a failure. You want the reactor to always be stopped, even if the error is so bad that even a call to `logout()` fails. This ensures that that will always happen.

Here's a sample session for this program. The first time I was asked for a password, I supplied an incorrect one. The second time, a correct password was supplied.

```
$ ./t-error.py imap.example.com jgoerzen
Enter password for jgoerzen on christoph:
I have successfully connected to the server!
Logging in...
IMAPLogic.__init__ returning.
connectionMade returning.
Your login failed (attempt 1).
New username: jgoerzen
New password:
Logging in...
Logging out.
Stopping reactor.
```

Note that `errorhappened()` was never called here. That's because the `loginerror()` function never returned the `Failure` object again—it was able to correct the problem, and processing continued.

Here's an example in which a bad password was supplied repeatedly, until the program gave up:

```
$ ./terror.py imap.example.com jgoerzen
Enter password for jgoerzen on christoph:
I have successfully connected to the server!
Logging in...
IMAPLogic.__init__ returning.
connectionMade returning.
Your login failed (attempt 1).
New username: jgoerzen
New password:
Logging in...
Your login failed (attempt 2).
New username: jgoerzen
New password:
```

```

Logging in...
Your login failed (attempt 3).
You have tried to log in three times; I'm giving up.
An error occurred: Login failed.
Because of the error, I am logging out and stopping reactor...
Logging out.
Failure: twisted.protocols.imap4.IMAP4Exception: Login failed.
Stopping reactor.

```

Notice the `An error occurred` message from `errorhappened()` appeared this time. That's because `loginerror()` returned the `Failure` object after the third attempt, so it "fell through" to `errorhappened()`, which logged out and stopped the reactor.

Scanning the Folder List

Since IMAP supports multiple folders (also known as mailboxes), you'll sometimes need to get a list of folders that the server supports. For instance, a mail reader may wish to present a clickable list of folders to the user. With IMAP, you can get a list of folders in the user's account. Here's a program that does just that. Note that this example, and those that follow, use much of the same basic code that was used in the examples before.

```

#!/usr/bin/env python
# IMAP folder listing - Chapter 12 - tlist.py
# Note: This example assumes you have Twisted 1.1.0 or above installed.
# Command-line args: hostname, username

from twisted.internet import defer, reactor, protocol
from twisted.protocols.imap4 import IMAP4Client
import sys, getpass

class IMAPClient(IMAP4Client):
    def connectionMade(self):
        IMAPLogic(self)

class IMAPFactory(protocol.ClientFactory):
    protocol = IMAPClient
    def __init__(self, username, password):
        self.username = username
        self.password = password

```

```

def clientConnectionFailed(self, connector, reason):
    print "Client connection failed:", reason
    reactor.stop()

class IMAPLogic:
    """This class implements the main logic for the program."""
    def __init__(self, proto):
        self.proto = proto
        self.factory = proto.factory
        d = self.proto.login(self.factory.username, self.factory.password)
        d.addCallback(lambda x: self.proto.list('', '*'))
        d.addCallback(self.listresult)
        d.addCallback(self.logout)
        d.addCallback(self.stopreactor)

        d.addErrback(self.errorhappened)

    def listresult(self, data):
        print "%-35s %-5s %-37s" % ('Mailbox Name', 'Delim', 'Mailbox Flags')
        print '-' * 35, '-' * 5, '-' * 37
        for box in data:
            flags, delim, name = box
            print "%-35s %-5s %-37s" % (name, delim, '.join(flags))

    def logout(self, data = None):
        return self.proto.logout()

    def stopreactor(self, data = None):
        reactor.stop()

    def errorhappened(self, failure):
        print "An error occurred:", failure.getErrorMessage()
        print "Because of the error, I am logging out and stopping reactor..."
        d = self.logout()
        d.addBoth(self.stopreactor)
        return failure

password = getpass.getpass("Enter password for %s on %s: " %
    (sys.argv[2], sys.argv[1]))
reactor.connectTCP(sys.argv[1], 143, IMAPFactory(sys.argv[2], password))
reactor.run()

```

Listing folders is fairly simple. The call to `list()` takes two parameters: a reference and folder or folder pattern. The reference is usually left blank, and the folder pattern `*` is used to match all folders. The reference has a server-specific meaning, and could be used to refer to alternate mail storage locations or nonmail data such as Usenet news. The wildcard character `*` works similarly as with a UNIX shell—it matches zero or more characters. Thus, the pattern `Python*` would match any folder whose name begins with `Python`. One other wildcard character is available. The percent sign matches any character except the hierarchy delimiter.

The Deferred from `list()` yields a list of tuples. Each tuple has three items: a tuple of flags for the folder, the hierarchy delimiter, and the folder name. The hierarchy delimiter may be useful if you wish to create new folders; it commonly is either `"/"` or `". "`, but may be other values as well. It may also be useful if you wish to represent folders as a tree.

The standard flags listed may be zero or more of the following:

- `\NoInferiors` means that the folder doesn't contain any subfolders and that it's not possible for it to contain subfolders in the future. Your IMAP client will receive an error if it tries to create a folder under this folder.
- `\NoSelect` means that it's not possible to `SELECT` or `EXAMINE` this folder. That is, this folder doesn't and cannot contain any messages.
- `\Marked` means that the server considers this box to be interesting in some way; generally, this would be that new messages have been delivered since the last time the folder was examined. However, the absence of `\Marked` doesn't guarantee that the folder doesn't contain new messages; some servers don't implement `\Marked` at all.
- `\Unmarked` guarantees that the folder doesn't contain new messages.

Some servers return additional flags not covered in the standard. Your code must be able to accept and ignore those additional flags. Here's a sample output from this program:

```
$ ./tlist.py imap.example.com jgoerzen
Enter password for jgoerzen on christoph:
Mailbox Name          Delim Mailbox Flags
-----
INBOX.Drafts          .
INBOX.sent-mail        .
INBOX.Trash            .
INBOX                  . \Unmarked, \HasChildren
```

In this case, the mail server returned the nonstandard flags \HasNoChildren and \HasChildren as well as the standard flag \Unmarked. Thus, you know that the INBOX folder doesn't contain new messages, but you don't know whether or not the other folders do.

The folder INBOX is always guaranteed to be available and is defined as the primary preferred place where new mail is delivered. Accessing INBOX should give you the same messages as you would get with a POP client.

Examining Folders

Before you can actually download, search, or modify any messages, you must select or examine a particular folder. When you *select* a folder, you tell the IMAP server that all the following commands—until you change folders or exit the current one—will apply to the selected folder. When you *examine* a folder, the same thing occurs, but you open the folder in read-only mode. You should always use `examine()` instead of `select()` if you know you won't be modifying anything in the folder. In practical terms, whenever someone refers to selecting a folder with IMAP, an `examine()` command could be substituted unless data is going to be modified.

Message Numbers vs. UIDs

IMAP provides two different ways to refer to a specific message within a folder: a *message number* and a *UID* (unique identification). The difference between the two lies with persistence; message numbers are assigned when you select the folder. If you revisit the same folder later, a given message may have a different number. For programs such as mail readers or simple downloaders, this behavior (which is the same as POP) is fine; you don't need the numbers to stay the same.

A UID is designed to remain the same even if you close your connection to the server and don't reconnect again for another week. If a message had UID 1053 today, the same message will have UID 1053 tomorrow, and no other message in that folder will ever have UID 1053. If you're writing a synchronization tool, this behavior is quite useful; it will allow you to verify with 100 percent certainty that actions are being taken against the correct message.

There are certain conditions in which the server's promise to never change UID numbering may be broken. For instance, if a user deleted a folder on the server and then created a new one with the same name, numbering will likely be reset. IMAP provides UID validity checking, which will let you detect that this change has occurred, and let you know that any UIDs you have stored cannot be trusted.

Most IMAP commands that work with specific messages can take either message numbers or UIDs. The Twisted IMAP library provides a `uid` parameter to such commands. It defaults to false, but if true, the numbers passed to that command are taken to be UIDs instead of message numbers.

Message Ranges

Most IMAP commands that work with messages can work with one or more messages. This can make processing far faster. Instead of issuing separate commands and receiving separate responses for each individual message, you can operate on a group of messages as a whole. The operation works faster since you no longer have to deal with network latency for so many commands.

Twisted provides an object called `MessageSet`, which helps you compose lists of messages for IMAP. Alternatively, you can supply a string that represents ranges in standard IMAP form. For instance, the string `2,4:6,20:*` might include messages 2, 4, 5, 6, 20, 21, 22 in a mailbox with 22 messages.

Summary Information

When you select (or examine) a folder, the IMAP server provides some summary information about the folder. This information includes information about the folder itself and its messages. With Twisted, the information is given to the callback of a call to `examine()` or `select()`. The standard summary items are

- `EXISTS`. Specifies the number of messages in the folder.
- `FLAGS`. Specifies a list of the flags that can be set on messages in this folder.
- `RECENT`. Specifies the server's approximation of the number of messages that have appeared in the folder since the last time an IMAP client has selected it.
- `PERMANENTFLAGS`. Specifies the list of custom flags that can be set on messages and is usually empty.
- `UIDNEXT`. Is the server's *estimation* of the next UID to be assigned to a new message.

- **UIDVALIDITY**. Specifies a string that can be used by clients to verify that the UID numbering hasn't changed.
- **UNSEEN**. Specifies the message number of the first unseen message in the folder.

Of these, servers are only required to return **FLAGS**, **EXISTS**, and **RECENT**, though most will include at least **UIDVALIDITY** as well.

Here's an example of reading and displaying the summary information. You can see that Twisted provides these pieces of information in a dictionary. The keys of the dictionary correspond to the names of information, and the values correspond to the data.

```
#!/usr/bin/env python
# IMAP folder examine information - Chapter 12
# texamine.py
# Note: This example assumes you have Twisted 1.1.0 or above installed.
# Command-line args: hostname, username

from twisted.internet import defer, reactor, protocol
from twisted.protocols.imap4 import IMAP4Client
import sys, getpass

class IMAPClient(IMAP4Client):
    def connectionMade(self):
        IMAPLogic(self)

class IMAPFactory(protocol.ClientFactory):
    protocol = IMAPClient
    def __init__(self, username, password):
        self.username = username
        self.password = password

    def clientConnectionFailed(self, connector, reason):
        print "Client connection failed:", reason
        reactor.stop()

class IMAPLogic:
    """This class implements the main logic for the program."""
    def __init__(self, proto):
        self.proto = proto
        self.factory = proto.factory
        d = self.proto.login(self.factory.username, self.factory.password)
```

```

# This lambda creates a function that takes one argument,
# ignores it, and returns the value of self.proto.examine().
# It's useful because we don't want to pass the result of
# self.proto.login() to examine().
d.addCallback(lambda x: self.proto.examine('INBOX'))
d.addCallback(self.examineresult)
d.addCallback(self.logout)
d.addCallback(self.stopreactor)

d.addErrback(self.errorhappened)

def examineresult(self, data):
    for key, value in data.items():
        if isinstance(value, tuple):
            print "%s: %s" % (key, ", ".join(value))
        else:
            print "%s: %s" % (key, value)

def logout(self, data = None):
    return self.proto.logout()

def stopreactor(self, data = None):
    reactor.stop()

def errorhappened(self, failure):
    print "An error occurred:", failure.getErrorMessage()
    d = self.logout()
    d.addBoth(self.stopreactor)
    return failure

password = getpass.getpass("Enter password for %s on %s: " % \
    (sys.argv[2], sys.argv[1]))
reactor.connectTCP(sys.argv[1], 143, IMAPFactory(sys.argv[2], password))
reactor.run()

```

When run, this program displays results such as this:

```

$ ./texamine.py imap.example.com jgoerzen
Enter password for jgoerzen on imap.example.com:
EXISTS: 114
PERMANENTFLAGS:
READ-WRITE: 0
FLAGS: \Draft, \Answered, \Flagged, \Deleted, \Seen, \Recent
UIDVALIDITY: 1071066868
RECENT: 0

```

That shows that my INBOX contains 114 messages, none of which are recent. If your program is interested in processing messages across multiple executions, you can compare the UIDVALIDITY to a stored value from a previous session. If it's the same, you know that any UIDs that you received then will apply to the same messages now.

Basic Downloading

In Chapter 11, you can find a program that connects to a POP server and downloads every message in the user's mailbox, writing the result out to a file. The same program is possible with IMAP.

With IMAP, the `FETCH` command is used to download mail. The Twisted library provides many different fetch commands that are used to implement the different ways that messages can be selected and downloaded.

Downloading an Entire Mailbox with One Command

The simplest way to do this involves downloading all messages at once, in a single big gulp. While this is the simplest and requires the least network traffic, it requires Twisted to cache the result in memory before passing it to you. Therefore, for very large mailboxes, it's not practical. Here's an example:

```
#!/usr/bin/env python
# IMAP downloader with single fetch command - Chapter 12
# tdlbig.py
# Note: This example assumes you have Twisted 1.1.0 or above installed.
# Command-line args: hostname, username, destination file

from twisted.internet import defer, reactor, protocol
from twisted.protocols.imap4 import IMAP4Client
import sys, getpass, email

class IMAPClient(IMAP4Client):
    def connectionMade(self):
        IMAPLogic(self)

class IMAPFactory(protocol.ClientFactory):
    protocol = IMAPClient
    def __init__(self, username, password):
        self.username = username
        self.password = password
```

```
def clientConnectionFailed(self, connector, reason):
    print "Client connection failed:", reason
    reactor.stop()

class IMAPLogic:
    """This class implements the main logic for the program."""
    def __init__(self, proto):
        self.proto = proto
        self.factory = proto.factory
        d = self.proto.login(self.factory.username, self.factory.password)

        # These lambdas create functions that take one argument,
        # ignore it, and return the value of self.proto.examine().
        # It's useful because we don't want to pass the result of
        # self.proto.login() to examine().
        d.addCallback(lambda x: self.proto.examine('INBOX'))
        d.addCallback(lambda x: self.proto.fetchSpecific('1:*', peek = 1))
        d.addCallback(self.gotmessages)
        d.addCallback(self.logout)
        d.addCallback(self.stopreactor)

        d.addErrback(self.errorhappened)

    def gotmessages(self, data):
        destfd = open(sys.argv[3], "at")
        for key, value in data.items():
            print "Writing message", key
            msg = email.message_from_string(value[0][2])
            destfd.write(msg.as_string(unixfrom = 1))
            destfd.write("\n")
        destfd.close()

    def logout(self, data = None):
        return self.proto.logout()

    def stopreactor(self, data = None):
        reactor.stop()

    def errorhappened(self, failure):
        print "An error occurred:", failure.getErrorMessage()
        d = self.logout()
        d.addBoth(self.stopreactor)
        return failure
```

```

password = getpass.getpass("Enter password for %s on %s: " % \
    (sys.argv[2], sys.argv[1]))
reactor.connectTCP(sys.argv[1], 143, IMAPFactory(sys.argv[2], password))
reactor.run()

```

Like previous examples, this program connects and authenticates in a typical way. Then it calls `examine()` to select the folder `INBOX`. Next, it calls `fetchSpecific()`. This function downloads one or more messages from a mail folder. The first argument, '`1:*`', is a message range that corresponds to all messages in the folder. By setting `peek` to a true value, messages aren't modified as you read them. If `peek` were not specified, the `\Seen` flag would automatically be added for any messages downloaded.

The `gotmessages()` function is invoked with the downloaded messages. It's passed a dictionary whose keys are the message number, and its values are a list of components of the message. In this case, I requested only one component (the entire body), which is assumed to reside at `value[0][2]`. (Sometimes that assumption is wrong if extra data is present; in those cases, this example will break, but the next example will work.) The message then is written out in the same fashion as with POP.

Here's an example invocation:

```

$ ./tdlbig.py imap.example.com jgoerzen mailbox
Enter password for jgoerzen on imap.example.com:
Writing message 1
Writing message 2

```

Downloading Messages Individually

Messages can be quite large—many mail systems permit users to have hundreds or thousands of messages that can be 10 MB or more. That kind of mailbox can easily exceed the RAM on the client machine. The previous example loads all messages into RAM before writing, but that solution simply doesn't scale.

To combat that problem, you can issue a request to find out the message numbers present in the mailbox, then download those messages individually. IMAP has no command that returns message numbers solely, but it does have many commands that return small bits of information alongside message numbers.

In this example, the UID of a message is returned. You'll recall that the UID is a unique ID that's supposed to remain constant even across connections. While that capability isn't needed for this program, it does show how to use the UID in a basic sense, as shown here:

```
#!/usr/bin/env python
# IMAP downloader - Chapter 12 - tdownload.py
# Note: This example assumes you have Twisted 1.1.0 or above installed.
# Command-line args: hostname, username, destination file

from twisted.internet import defer, reactor, protocol
from twisted.protocols.imap4 import IMAP4Client
import sys, getpass, email

class IMAPClient(IMAP4Client):
    def connectionMade(self):
        IMAPLogic(self)

class IMAPFactory(protocol.ClientFactory):
    protocol = IMAPClient
    def __init__(self, username, password):
        self.username = username
        self.password = password

    def clientConnectionFailed(self, connector, reason):
        print "Client connection failed:", reason
        reactor.stop()

class IMAPLogic:
    """This class implements the main logic for the program."""
    def __init__(self, proto):
        self.proto = proto
        self.factory = proto.factory
        d = self.proto.login(self.factory.username, self.factory.password)
        d.addCallback(lambda x: self.proto.examine('INBOX'))
        d.addCallback(lambda x: self.proto.fetchUID('1:*'))
        d.addCallback(self.handleuids)
        d.addCallback(self.logout)
        d.addCallback(self.stopreactor)

        d.addErrback(self.errorhappened)
```

```

def handleuids(self, uids):
    dlist = []
    destfd = open(sys.argv[3], "at")
    for data in uids.values():
        uid = data['UID']
        d = self.proto.fetchSpecific(uid, uid = 1, peek = 1)
        d.addCallback(self.gotmessage, destfd, uid)
        dlist.append(d)
    dl = defer.DeferredList(dlist)
    dl.addCallback(lambda x, fd: fd.close(), destfd)
    return dl

def gotmessage(self, data, destfd, uid):
    print "Received message UID", uid
    for key, value in data.items():
        print "Writing message", key
        i = value[0].index('BODY') + 2
        msg = email.message_from_string(value[0][i])
        destfd.write(msg.as_string(unixfrom = 1))
        destfd.write("\n")

def logout(self, data = None):
    return self.proto.logout()

def stopreactor(self, data = None):
    reactor.stop()

def errorhappened(self, failure):
    print "An error occurred:", failure.getErrorMessage()
    d = self.logout()
    d.addBoth(self.stopreactor)
    return failure

password = getpass.getpass("Enter password for %s on %s: " % \
    (sys.argv[2], sys.argv[1]))
reactor.connectTCP(sys.argv[1], 143, IMAPFactory(sys.argv[2], password))
reactor.run()

```

Here's a sample invocation of this example. Like the POP downloading program, its output is written to the file specified by the third argument:

```
$ ./tdownload.py imap.example.com jgoerzen mbox
Enter password for jgoerzen on imap.example.com:
Received message UID 37025
Writing message 1
Received message UID 37026
Writing message 2
Received message UID 37027
Writing message 3
Received message UID 37028
Writing message 4
Received message UID 37029
```

Writing message 5 calls `fetchUID('1:*)` to obtain the UIDs of all messages in the folder. The result is passed to `handleuids()`. The value passed in consists of a dictionary mapping from the message number to another dictionary containing information about this message—in this case, only the key `UID`. This example could have used message sequence numbers as the previous example did; it simply uses UIDs instead to illustrate their use.

Notice the loop in `handleuids()` that processes the message. For each message in the folder, it starts off a call to `fetchSpecific()`. Just to show how it's done, the program uses the `uid = 1` parameter tells `fetchSpecific()` to treat the first argument as a UID rather than a message number.

Note, though, that the program doesn't bother to wait for `fetchSpecific()` to return before calling it again. In fact, it's entirely possible that `fetchSpecific()` could be called several hundred times before even the first message has been downloaded. That's OK; Twisted will queue up the calls and run them one at a time over the network connection.

As each call to `fetchSpecific()` is made, the `Deferred` object it returns is added to the list `dlist`. When the list is complete, a new `DeferredList` object is created. A `DeferredList` will only call its callbacks when all the `Deferred` objects passed in have finished processing. The `DeferredList` will pass a list of all return values from its component objects to the first callback. In this case, you aren't interested in that value and simply schedule it to close the file once all messages are downloaded. The `DeferredList` gets returned, and thus after it's finished, `logout()` will be called.

Notice the call to `index('BODY')` in `gotmessage()`. In the previous example, the message body was at `value[0][2]`. In fact, previously, `value[0]` looked something like `['BODY', [], 'All lines of body text are here']`.

Now, `value[0]` will look more like `['UID', '760', 'BODY', [], 'All lines of body text are here']`. The `fetchSpecific()` command has caused the IMAP server to return more information. The part you're interested in is the one that follows '`BODY`' and `[]`; hence, you find where '`BODY`' is and count two steps past it.

Flagging and Deleting Messages

All messages on an IMAP server can have one or more flags. The set of standard flags specified in RFC3501 are summarized in Table 12-1.

Table 12-1. Standard IMAP Flags

Flag	Description
<code>\Answered</code>	The user has replied to the message.
<code>\Deleted</code>	The message will be permanently removed the next time <code>EXPUNGE</code> is called.
<code>\Draft</code>	The user hasn't finished composing the message.
<code>\Flagged</code>	The message has been singled out specially. The purpose and meaning of this flag varies between mail readers.
<code>\Recent</code>	No other IMAP client has seen this message before. <code>\Recent</code> is unique in that the flag cannot be added or removed by normal commands; it's automatically removed after the mailbox is selected.
<code>\Seen</code>	The message has been read.

As you can see, these flags correspond roughly to the information that many mail readers visually present about each message. While the terminology may differ ("new" instead of "not seen"), the meaning is broadly understood.

Particular servers may also support other flags, and those flags don't necessarily begin with the backslash. Also, the `\Recent` flag isn't reliably supported, and an IMAP client can treat it as, at best, a hint.

Reading Flags

Twisted provides a convenient function called `fetchFlags()`, which is used to obtain the flags on a message or a set of messages. Here's an example of fetching and displaying flags:

```
#!/usr/bin/env python
# IMAP flag display -- Chapter 12 - tflags.py
# Note: This example assumes you have Twisted 1.1.0 or above installed.
# Command-line args: hostname, username

from twisted.internet import defer, reactor, protocol
from twisted.protocols.imap4 import IMAP4Client
import sys, getpass, email

class IMAPClient(IMAP4Client):
    def connectionMade(self):
        IMAPLogic(self)

class IMAPFactory(protocol.ClientFactory):
    protocol = IMAPClient
    def __init__(self, username, password):
        self.username = username
        self.password = password

    def clientConnectionFailed(self, connector, reason):
        print "Client connection failed:", reason
        reactor.stop()

class IMAPLogic:
    """This class implements the main logic for the program."""
    def __init__(self, proto):
        self.proto = proto
        self.factory = proto.factory
        d = self.proto.login(self.factory.username, self.factory.password)
        d.addCallback(lambda x: self.proto.examine('INBOX'))
        d.addCallback(lambda x: self.proto.fetchFlags('1:*'))
        d.addCallback(self.handleflags)
        d.addCallback(self.logout)
        d.addCallback(self.stopreactor)
        d.addErrback(self.errorhappened)
```

```

def handleflags(self, flags):
    for num, flaglist in flags.items():
        print "Message %s has flags %s" % \
            (num, ", ".join(flaglist['FLAGS']))

def logout(self, data = None):
    return self.proto.logout()

def stopreactor(self, data = None):
    reactor.stop()

def errorhappened(self, failure):
    print "An error occurred:", failure.getErrorMessage()
    d = self.logout()
    d.addBoth(self.stopreactor)
    return failure

password = getpass.getpass("Enter password for %s on %s: " % \
    (sys.argv[2], sys.argv[1]))
reactor.connectTCP(sys.argv[1], 143, IMAPFactory(sys.argv[2], password))
reactor.run()

```

When you run the program (which shouldn't modify your mailbox at all), you get output similar to this:

```

$ ./tflags.py imap.example.com jgoerzen
Enter password for jgoerzen on imap.example.com:
Message 1 has flags \Seen
Message 2 has flags \Seen
Message 3 has flags \Seen
Message 4 has flags \Seen
...
Message 31 has flags \Seen
Message 32 has flags \Answered, \Seen
Message 33 has flags \Seen
...
Message 220 has flags \Seen
Message 221 has flags \Seen, \Recent
Message 222 has flags \Seen, \Recent

```

Notice that this IMAP server is putting the \Seen and \Recent flags on the same message—a combination that doesn't make logical sense. Many servers do strange things like this with the \Recent flag.

Setting Flags

Flags can be set on messages. Most IMAP servers will make them persist across connections, though this isn't guaranteed. Twisted provides three methods for altering flags: `setFlags()`, `addFlags()`, and `removeFlags()`.

The `setFlags()` function takes a specific set of flags to apply to the message. Flags are added or removed from the message as necessary so that the message winds up having the requested flags. The `addFlags()` and `removeFlags()` add or remove flags, respectively. All of these functions simply take a list of strings, whereby each string specifies a flag to work with. It's not an error to add a flag to a message that already has it, or to delete a flag from a message that doesn't have that flag. The `tdownload-and-delete.py` example in the next section illustrates the usage of `addFlags()`.

Deleting Messages

Deleting messages from an IMAP mailbox is mostly a special case of setting flags. For each message that is to be deleted, you need to set the flag `\Deleted`. Finally, when you're ready to actually delete the message, you call `expunge()`. This function simply causes the IMAP server to delete every message in the mailbox that has the `\Deleted` flag. Since IMAP servers don't guarantee persistent storage of flags, you should make sure to run `expunge()` before disconnecting if you really want the message to go away.

Here's an updated example of the downloader program. This message will download the mail and then delete it, mimicking a POP client without a "leave mail on server" option. If you're interested in how POP and IMAP compare, you could compare this program to the `download-and-delete.py` example in Chapter 11.

```
#!/usr/bin/env python
# IMAP downloader with message deletion—Chapter 12
# tdownload-and-delete.py
# Note: This example assumes you have Twisted 1.1.0 or above installed.
# Command-line args: hostname, username, destination file
#
# WARNING: THIS PROGRAM WILL DELETE ALL MAIL FROM THE INBOX!

from twisted.internet import defer, reactor, protocol
from twisted.protocols.imap4 import IMAP4Client, MessageSet
import sys, getpass, email
```

```

class IMAPClient(IMAP4Client):
    def connectionMade(self):
        IMAPLogic(self)

class IMAPFactory(protocol.ClientFactory):
    protocol = IMAPClient
    def __init__(self, username, password):
        self.username = username
        self.password = password

    def clientConnectionFailed(self, connector, reason):
        print "Client connection failed:", reason
        reactor.stop()

class IMAPLogic:
    """This class implements the main logic for the program."""
    def __init__(self, proto):
        self.proto = proto
        self.factory = proto.factory
        d = self.proto.login(self.factory.username, self.factory.password)
        d.addCallback(lambda x: self.proto.select('INBOX'))
        d.addCallback(lambda x: self.proto.fetchUID('1:*'))
        d.addCallback(self.handleuids)
        d.addCallback(self.deletemessages)
        d.addCallback(self.logout)
        d.addCallback(self.stopreactor)

        d.addErrback(self.errorhappened)

    def handleuids(self, uids):
        self.uidlist = MessageSet()
        dlist = []
        destfd = open(sys.argv[3], "at")
        for num, data in uids.items():
            uid = data['UID']
            d = self.proto.fetchSpecific(uid, uid = 1, peek = 1)
            d.addCallback(self.gotmessage, destfd, uid)
            dlist.append(d)
        dl = defer.DeferredList(dlist)
        dl.addCallback(lambda x, fd: fd.close(), destfd)
        return dl

```

```

def gotmessage(self, data, destfd, uid):
    print "Received message UID", uid
    for key, value in data.items():
        print "Writing message", key
        i = value[0].index('BODY') + 2
        msg = email.message_from_string(value[0][i])
        destfd.write(msg.as_string(unixfrom = 1))
        destfd.write("\n")
    self.uidlist.add(int(uid))

def deletemessages(self, data = None):
    print "Deleting messages", str(self.uidlist)
    d = self.proto.addFlags(str(self.uidlist), ["\\Deleted"], uid = 1)
    d.addCallback(lambda x: self.proto.expunge())
    return d

def logout(self, data = None):
    return self.proto.logout()

def stopreactor(self, data = None):
    reactor.stop()

def errorhappened(self, failure):
    print "An error occurred:", failure.getErrorMessage()
    d = self.logout()
    d.addBoth(self.stopreactor)
    return failure

print "WARNING: this program will delete all mail from the INBOX!"
password = getpass.getpass("Enter password for %s on %s: " % \
    (sys.argv[2], sys.argv[1]))
reactor.connectTCP(sys.argv[1], 143, IMAPFactory(sys.argv[2], password))
reactor.run()

```

This program is similar to the previous download example, but there are some changes. In handleuids(), self.uidlist is created and holds a MessageSet object. Twisted provides MessageSet to conveniently summarize message ranges for IMAP. For instance, if you added message numbers 3, 4, 5, 6, 8, 10, 11, and 12, the MessageSet could generate the IMAP range string 3:6,8,10:12 for you, which includes the same messages.

Then, in `gotmessage()`, after each message is successfully written to disk, its UID is added to the message list. Finally, after all messages are downloaded, `deletemessages()` gets called. It sets the `\Deleted` flag on each message that was successfully processed, then calls `expunge()` to physically remove the messages. Note that the backslash is a special character in Python strings, so it's escaped as "`\\\Deleted`".

CAUTION Don't run this program with any important mailbox, such as your primary mailbox. All messages in it will be deleted!

You can run the program like this:

```
$ ./tdownload-and-delete.py imap.example.com jgoerzen testmbox
WARNING: this program will delete all mail from the INBOX!
Enter password for jgoerzen on imap.example.com:
Received message UID 37030
Writing message 1
Received message UID 37036
Writing message 2
Deleting messages 37030,37036
```

Retrieving Message Parts

One nice feature of IMAP is that it lets you retrieve individual parts of a message. You can use this capability to retrieve the message headers, the body, or individual MIME parts of the message. Not all IMAP servers support individual MIME part retrieval, but most will.

Components of messages can be specified in one of two ways: by using pre-defined component names, or by using component numbers. In some cases, the component names may be more useful; for instance, if you always know you need just the headers, you can ask for them. However, if you require a specific MIME part, you'll have to use numbers.

Twisted provides several functions that fetch named parts of the message, or information about the message. Table 12-2 summarizes these functions.

Table 12-2. Functions for Reading Message Components and Metadata

Function Name	Data Retrieved
fetchAll()	Date, envelope, flags, and size
fetchBody()	Message body text (no headers)
fetchBodyStructure()	Structure of the message body
fetchEnvelope()	Message envelope (certain headers parsed out)
fetchHeaders()	Message headers as one string
fetchMessage()	Message headers plus body
fetchSimplifiedBody()	Same as fetchBodyStructure(), but omits some infrequently used information
fetchSpecific()	Part or all of a message; selectable headers and/or body for that part

Finding Message Structures

Before you can use `fetchSpecific()` to find a particular part of a message, you need to know the number for that part. To do that, you can use `fetchBodyStructure()` or `fetchSimplifiedBody()`. Here's an example program that connects to an INBOX and displays the structure of each message contained in that mailbox (it should not modify your mailbox).

```
#!/usr/bin/env python
# IMAP structure display -- Chapter 12 - tstructure.py
# Note: This example assumes you have Twisted 1.1.0 or above installed.
# Command-line args: hostname, username

from twisted.internet import defer, reactor, protocol
from twisted.protocols.imap4 import IMAP4Client
import sys, getpass, email

class IMAPClient(IMAP4Client):
    def connectionMade(self):
        IMAPLogic(self)
```

```

class IMAPFactory(protocol.ClientFactory):
    protocol = IMAPClient
    def __init__(self, username, password):
        self.username = username
        self.password = password

    def clientConnectionFailed(self, connector, reason):
        print "Client connection failed:", reason
        reactor.stop()

class IMAPLogic:
    """This class implements the main logic for the program."""
    def __init__(self, proto):
        self.proto = proto
        self.factory = proto.factory
        d = self.proto.login(self.factory.username, self.factory.password)
        d.addCallback(lambda x: self.proto.examine('INBOX'))
        d.addCallback(lambda x: self.downloadinfo())
        d.addCallback(self.displayinfo)
        d.addCallback(self.logout)
        d.addCallback(self.stopreactor)

        d.addErrback(self.errorhappened)

    def downloadinfo(self):
        dstructure = self.proto.fetchSimplifiedBody('1:*', uid = 1)
        envelope = self.proto.fetchEnvelope('1:*', uid = 1)
        return defer.DeferredList([dstructure, envelope])

    def displayinfo(self, data):
        structure, envelope = data
        for msgnum, structdata in structure[1].items():
            envelopedata = envelope[1][msgnum]['ENVELOPE']
            print "Message %s (%s): %s" % (msgnum, structdata['UID'],
                envelopedata[1])
            parts = structdata['BODY']
            self.printpart(parts)

```

```

def printpart(self, part, itemnum = '1', istoplevel = 1):
    #print "part:", part
    #print "part[0]:", part[0]
    if not istoplevel:
        nextitem = itemnum + '.1'
    else:
        nextitem = itemnum
    if not isinstance(part[0], str):
        for item in part[:-1]:
            self.printpart(item, nextitem, 0)
            # Increment the counter.
            cparts = nextitem.split('.')
            cparts[-1] = str(int(cparts[-1]) + 1)
            nextitem = '.'.join(cparts)
        else:
            print "  %s: %s/%s" % (itemnum, part[0], part[1])
            if part[0].lower() == 'message' and part[1].lower() == 'rfc822':
                self.printpart(part[8], nextitem, 0)

def logout(self, data = None):
    return self.proto.logout()

def stopreactor(self, data = None):
    reactor.stop()

def errorhappened(self, failure):
    print "An error occurred:", failure.getErrorMessage()
    d = self.logout()
    d.addBoth(self.stopreactor)
    return failure

password = getpass.getpass("Enter password for %s on %s: " % \
    (sys.argv[2], sys.argv[1]))
reactor.connectTCP(sys.argv[1], 143, IMAPFactory(sys.argv[2], password))
reactor.run()

```

Here are some snippets of the output from running this program on my own INBOX:

```
$ ./tstructure.py imap.example.com jgoerzen
Enter password for jgoerzen on imap.example.com:
Message 1 (11): patch
 1: text/plain
 2: text/x-diff
Message 2 (214): Minor correction
 1: text/plain
Message 57 (808): Any changes to your contact info?
 1.1: text/plain
 1.2: text/html
 2: text/x-vcard
Message 78 (829): Gopher
 1: text/plain
 2: text/html
Message 142 (893): Fwd: Undelivered Mail Returned to Sender
 1: text/plain
 2: message/delivery-status
 3: message/rfc822
 3.1: text/plain
```

The first message, number 1, is a message containing two MIME parts: a plain text part and a text/x-diff part. These are numbered 1 and 2 for IMAP. Message 2 is a simple message that may not even use MIME; it has only a plain text component.

Message 57 is an interesting case. Part 1 isn't displayed, because it's solely a multipart/alternative. You could retrieve part 1 and get both the text and HTML versions. Part 1.1 is text; 1.2 is HTML. Finally, part 2 contains contact information.

This message is different from the more standard alternative (message 78) because it contains the v-card, which isn't part of the alternative. Normally, if a message consists solely of alternatives, all the alternatives are top-level parts.

Finally, message 142 contains an embedded message. Observe that part 3 is a message/rfc822 type; that is the MIME type of a message itself. IMAP servers will then open up those messages, and their components are listed as subparts. In this case, it has only one component, and that component is a plain text one—part 3.1.

Let's look at the source code for this program. The `downloadinfo()` function first calls `fetchSimplifiedBody()` for all messages. It passes along `uid = 1` so that UID information is present in the output (you'll need that for the next example). Next, it retrieves the envelope; that would be nice to display to the user. Since these both return callbacks, they are each added to a `DeferredList`, which is returned.

Because of the callback sequence in `__init__()`, `displayinfo()` is passed the results of `downloadinfo()`. The `DeferredList` object passes along a list of return values, which are split out into structure and envelope. The program iterates over the list, printing message numbers, UIDs, and headers. Finally, it passes the structure data to `printpart()`.

NOTE In this case, the structure data was placed under 'BODY' in the data returned from `fetchSimplifiedBody()`. If you instead use `fetchBodyStructure()`, the equivalent data will be in 'BODYSTRUCTURE'.

IMAP returns the body structure in a rather confusing format, and, unfortunately, Twisted doesn't help parse it much. I put some commented-out lines at the top of `printpart()`; if you want to see what Twisted is giving it, uncomment them.

The `printpart()` function always receives a list. This list may contain either a bunch of strings describing a particular part, or another list that contains parts or subparts itself. The code checks to see what it got; if it was a list, it generates appropriate item numbers and passes the components to itself recursively. Otherwise, it displays the part. If the part was an embedded message, again it calls itself recursively.

Retrieving Numbered Parts

Now that you have a part number (such as 1 or 3.2), you can use that to retrieve that particular part from the server. You may wish to do this if, for instance, you downloaded only the text of a message on the first pass, and now want to download an attachment. Downloading a specific part requires `fetchSpecific()`. Here's an example:

```
#!/usr/bin/env python
# IMAP part downloader - Chapter 12 - tdlpart.py
# Note: This example assumes you have Twisted 1.1.0 or above installed.
# Command-line args: hostname, username

from twisted.internet import defer, reactor, protocol
from twisted.protocols.imap4 import IMAP4Client
import sys, getpass, email
```

```

class IMAPClient(IMAP4Client):
    def connectionMade(self):
        IMAPLogic(self)

class IMAPFactory(protocol.ClientFactory):
    protocol = IMAPClient
    def __init__(self, username, password, uid, part):
        self.username = username
        self.password = password
        self.uid = uid
        self.part = part

    def clientConnectionFailed(self, connector, reason):
        print "Client connection failed:", reason
        reactor.stop()

class IMAPLogic:
    """This class implements the main logic for the program."""
    def __init__(self, proto):
        self.proto = proto
        self.factory = proto.factory
        d = self.proto.login(self.factory.username, self.factory.password)
        d.addCallback(lambda x: self.proto.examine('INBOX'))
        d.addCallback(lambda x: self.proto.fetchSpecific(self.factory.uid,
            uid = 1, headerNumber = self.factory.part.split('.'), peek = 1))
        d.addCallback(self.displaypart)
        d.addCallback(self.logout)
        d.addCallback(self.stopreactor)

        d.addErrback(self.errorhappened)

    def displaypart(self, data):
        for key, value in data.items():
            i = value[0].index('BODY') + 2
            print value[0][i]

    def logout(self, data = None):
        return self.proto.logout()

    def stopreactor(self, data = None):
        reactor.stop()

```

```

def errorhappened(self, failure):
    print "An error occurred:", failure.getErrorMessage()
    d = self.logout()
    d.addBoth(self.stopreactor)
    return failure

password = getpass.getpass("Enter password for %s on %s: " % \
    (sys.argv[2], sys.argv[1]))
sys.stdout.write("Enter message UID: ")
uid = int(sys.stdin.readline().strip())
sys.stdout.write("Enter message part: ")
part = sys.stdin.readline().strip()
reactor.connectTCP(sys.argv[1], 143, IMAPFactory(sys.argv[2], password, uid,
    part))
reactor.run()

```

You can use the UID number (in parenthesis in the output from the previous example) and part number with this program to download the content of a specific message part. Here's an example session:

```

$ ./tdlpart.py imap.example.com jgoerzen
Enter password for jgoerzen on imap.example.com:
Enter message UID: 829
Enter message part: 1
Hi,

```

Thank you for the work on PyGopherd. We are using it here

...

Note that there are no headers in this text. That's because the `text/plain` component is in the body and doesn't include the message headers.

Finding Messages

IMAP also provides features for searching mailboxes. In addition to saving you time from having to code search algorithms, it's also a convenience to the user because the search executes quicker. With a system such as POP, the full text of all messages would have to be downloaded in order to perform the search. With IMAP, you can send search criteria to the server, and let the server perform the search for you. You'll use the function `search()` to do this, but first you need to know how to generate queries.

Composing Queries

The `search()` function takes a list of queries, which will be combined using “and” logic; that is, for a message to match, it must pass all supplied queries.

The `twisted.protocols.imap4` module contains three functions that are used to compose queries: `Query()`, `Not()`, and `Or()`. The `Not()` function takes a query and negates it so that a message that doesn’t meet the criteria will be considered a match. The `Or()` function takes two or more queries; a message that meets the criteria of one or more will be considered a match. All three functions return a value that can be passed to `Or()` or `Not()` or on to `search()`.

`Query()` itself takes one or more keyword arguments; they are “anded” together (a message matches only if all arguments hold). Tables 12-3, 12-4, and 12-5 describe the keywords that are available. For keywords that take a `bool`, you should set the value to `True`. Setting them to `False` will have no effect.

Table 12-3. Flag-Related Search Keywords

Keyword	Parameter	Messages Matched
<code>answered</code>	<code>bool</code>	Messages with \Answered flag
<code>deleted</code>	<code>bool</code>	Messages with \Deleted flag
<code>draft</code>	<code>bool</code>	Messages with \Draft flag
<code>flagged</code>	<code>bool</code>	Messages with \Flagged flag
<code>keyword</code>	Keyword string	Messages with the server-defined keyword flag set
<code>new</code>	<code>bool</code>	Messages with \Recent flag but without \Seen flag
<code>old</code>	<code>bool</code>	Messages without \Recent flag
<code>unanswered</code>	<code>bool</code>	Messages without \Answered flag
<code>undeleted</code>	<code>bool</code>	Messages without \Deleted flag
<code>undraft</code>	<code>bool</code>	Messages without \Draft flag
<code>unflagged</code>	<code>bool</code>	Messages without \Flagged flag
<code>unkeyword</code>	Keyword string	Messages without the server-defined keyword flag set
<code>unseen</code>	<code>bool</code>	Messages without \Seen flag

Table 12-4. Header-Related Search Keywords

Keyword	Parameter	Messages Matched
bcc	Search substring	All messages containing the parameter in the BCC header.
cc	Search substring	All messages containing the parameter anywhere in the CC header.
from	Search substring	All messages containing the parameter anywhere in the From header. Note that MIME-encoded headers aren't decoded prior to searching, so this is best used for e-mail addresses instead of names.
header	Tuple of (header name, search substring)	All messages in which the specified header contains the specified substring.
sentbefore	Date, format: 25-Dec-2004	Messages with the Date header before the specified date.
senton	Date, format: 25-Dec-2004	Messages with the Date header falling on the specified date.
sentsince	Date, format: 25-Dec-2004	Messages with the Date header falling after the specified date.
subject	Search substring	All messages containing the parameter anywhere in the Subject header. MIME-encoded Subject headers may not be decoded prior to searching.
to	Search substring	All messages containing the parameter in the To header. MIME-encoded headers aren't decoded prior to searching, so this is best used for e-mail addresses instead of names.

Table 12-5. Other Search Keywords

Keyword	Parameter	Messages Matched
before	Date, format: 25-Dec-2004	Messages with IMAP date before the parameter.
body	Search substring	All messages containing the parameter anywhere in the message body. The header isn't searched.
larger	Integer size	Messages strictly larger than the specified number of bytes.
messages	Message set string	Messages with numbers the fall in the specified set.
on	Date, format: 25-Dec-2004	Messages with an IMAP date that falls on the specified date.
since	Date, format: 25-Dec-2004	Messages with an IMAP date that falls after the specified date.
smaller	Integer size	Messages strictly smaller than the specified number of bytes.
text	Search substring	All messages containing the parameter anywhere in the headers or the body.
uid	Message set string	Messages with UIDs that fall in the specified set.

Running Queries

Now that you know what options are available with the query composition, you can go ahead and run your queries. Here's an example that queries a mailbox without modifying it:

```

#!/usr/bin/env python
# IMAP searching - Chapter 12 - tsearch.py
# Note: This example assumes you have Twisted 1.1.0 or above installed.
# Command-line args: hostname, username

from twisted.internet import defer, reactor, protocol
from twisted.protocols.imap4 import IMAP4Client, Query, Not, Or, MessageSet
import sys, getpass, email

class IMAPClient(IMAP4Client):
    def connectionMade(self):
        IMAPLogic(self)

class IMAPFactory(protocol.ClientFactory):
    protocol = IMAPClient
    def __init__(self, username, password):
        self.username = username
        self.password = password

    def clientConnectionFailed(self, connector, reason):
        print "Client connection failed:", reason
        reactor.stop()

class IMAPLogic:
    """This class implements the main logic for the program."""
    def __init__(self, proto):
        self.proto = proto
        self.factory = proto.factory
        d = self.proto.login(self.factory.username, self.factory.password)
        d.addCallback(lambda x: self.proto.examine('INBOX'))
        d.addCallback(self.runquery)
        d.addCallback(self.printqueryresult)
        d.addCallback(self.logout)
        d.addCallback(self.stopreactor)

        d.addErrback(self.errorhappened)

    def runquery(self, data = None):
        # Find messages without "test" in the subject, and that have either
        # \Seen or \Answered flags.
        subjq = Not(Query(subject = "test"))
        flagq = Or(Query(seen = 1), Query(answered = 1))
        return self.proto.search(subjq, flagq)

```

```

def printqueryresult(self, result):
    print "The following %d messages matched:" % len(result)
    m = MessageSet()
    for item in result:
        m.add(item)
    print str(m)

def logout(self, data = None):
    return self.proto.logout()

def stopreactor(self, data = None):
    reactor.stop()

def errorhappened(self, failure):
    print "An error occurred:", failure.getErrorMessage()
    d = self.logout()
    d.addBoth(self.stopreactor)
    return failure

password = getpass.getpass("Enter password for %s on %s: " % \
    (sys.argv[2], sys.argv[1]))
reactor.connectTCP(sys.argv[1], 143, IMAPFactory(sys.argv[2], password))
reactor.run()

```

Take a look at the `runquery()` function. It generates two queries. Those two queries are supposed to be combined with “and” logic, but there’s no `And()` function. That’s OK, because all the queries passed to `search()` are automatically combined with “and” logic. You can, in fact, pass as many queries as you like to `search()`. Since this effectively performs a boolean “and” operation, you can rewrite any boolean search function using this mechanism.

The `printqueryresult()` function displays a set of messages that’s matched. Here’s an example of the output:

```

$ ./tsearch.py imap.example.com jgoerzen
Enter password for jgoerzen on imap.example.com:
The following 224 messages matched:
1:48,50:114,116:184,186:227

```

Adding Messages

It's possible to add a message to a mailbox with IMAP. You don't need to send the message first with SMTP; IMAP is all that's needed. Adding a message is a simple process, though there are a couple things to be aware of.

The primary concern is line endings. Many UNIX machines use the ASCII linefeed character (0x0A, or "\n" in Python) to designate the end of a line of text. Windows machines use two characters: the carriage return (0x0D, or "\r" in Python) and linefeed. Older Macs use just the carriage return.

IMAP internally uses CR-LF ("\r\n" in Python) to designate the end of a line. Some IMAP servers will cause trouble if you upload a message that uses any other character for the end of line. Therefore, you must always use caution to translate uploaded messages to have the correct line endings. This problem is more common than you might expect, since most local mailbox formats use only "\n" for the end of a line. However, you must also be cautious in what you do; some messages may surprisingly use "\r\n", and IMAP clients have been known to fail in some situations in which a message uses *both* different line endings!

Here's an example of uploading a message to a folder. It contains logic to ensure that line endings are always correct from any "\n" or "\r\n" file.

```
#!/usr/bin/env python
# IMAP message upload - Chapter 12 - tappend.py
# Note: This example assumes you have Twisted 1.1.0 or above installed.
# Command-line args: hostname, username, sourcefile

from twisted.internet import defer, reactor, protocol
from twisted.protocols.imap4 import IMAP4Client
import sys, getpass, email
from StringIO import StringIO

class IMAPClient(IMAP4Client):
    def connectionMade(self):
        IMAPLogic(self)

class IMAPFactory(protocol.ClientFactory):
    protocol = IMAPClient
    def __init__(self, username, password):
        self.username = username
        self.password = password
```

```

def clientConnectionFailed(self, connector, reason):
    print "Client connection failed:", reason
    reactor.stop()

class IMAPLogic:
    """This class implements the main logic for the program."""
    def __init__(self, proto):
        self.proto = proto
        self.factory = proto.factory
        d = self.proto.login(self.factory.username, self.factory.password)
        d.addCallback(self.upload)
        d.addCallback(self.logout)
        d.addCallback(self.stopreactor)

        d.addErrback(self.errorhappened)

    def upload(self, data = None):
        fd = open(sys.argv[3])
        content = fd.read()
        fd.close()

        # Make sure it's all \r\n
        content = "\r\n".join(content.splitlines()) + "\r\n"

        fakefd = StringIO(content)
        return self.proto.append('INBOX', fakefd)

    def logout(self, data = None):
        return self.proto.logout()

    def stopreactor(self, data = None):
        reactor.stop()

    def errorhappened(self, failure):
        print "An error occurred:", failure.getErrorMessage()
        d = self.logout()
        d.addBoth(self.stopreactor)
        return failure

password = getpass.getpass("Enter password for %s on %s: " % \
    (sys.argv[2], sys.argv[1]))
reactor.connectTCP(sys.argv[1], 143, IMAPFactory(sys.argv[2], password))
reactor.run()

```

Run that program on a message file (Chapter 9 has several examples that generate suitable files), and you'll see a new message appear in your INBOX. Here's an example:

```
$ ./tappend.py imap.example.com jgoerzen message.txt
Enter password for jgoerzen on imap.example.com:
```

Notice that there's no call to `select()` or `examine()` in this program. The `append()` call is one of a small handful that don't require it; in this case, `append()` discovers which mailbox you want by taking a mailbox name as a parameter.

Also notice that Twisted's `append()` call expects a file descriptor as an argument. If you know that your input file is already in "\r\n" format, you could pass its file descriptor directly. In this case, Python's `StringIO` module is used to present an in-memory string as a substitute.

Creating and Deleting Folders

It's possible to create or delete folders on the IMAP server. There are two functions Twisted provides for this purpose: `create()` and `delete()`. They each take only a single argument—a string with the name of the mailbox to create or delete. Some IMAP servers or configurations may not permit these operations, or may have restrictions on naming; be sure to have error checking in place when calling them. Here's an example that creates and then deletes a folder:

```
d = obj.create('testfolder')
d.addCallback(lambda ignore: obj.delete('testfolder'))
```

Moving Messages Between Folders

The final topic is moving messages between folders. IMAP can do a server-side copy, which means that if you want to copy or move a message from one folder to the next, you don't have to download and re-append it. There's a `copy()` function to do this. It takes three arguments: a message set, the destination mailbox, and a boolean indicating whether or not you're supplying a UID. The source mailbox is taken to be the one that you've most recently selected. The copy operation should preserve all flags and IMAP dates. Here's an example of copying message number 5 from the user's inbox to a folder called Notes:

```
d = obj.examine('INBOX')
d.addCallback(lambda ignore: objcopy('5', 'Notes', 0))
```

Summary

IMAP is a robust protocol for accessing messages stored on remote servers. Two different IMAP libraries exist for Python: `imaplib`, which is built in to Python, and the IMAP library in Twisted. The library in Twisted is more convenient for many purposes and is described in this chapter.

Twisted uses event-based programming, which is called the “don’t call us, we’ll call you” method. Instead of calling a function and waiting until a reply is received, a program using Twisted hands that function another function to call when a reply is ready. In Twisted, the object used to manage this is the `Deferred`. Twisted programs normally run until `reactor.stop()` is called.

Error handling is accomplished by adding `errbacks` to a `Deferred` object. The given function gets called when an error occurs.

To obtain a list of available folders, you can call `list()`. The folder `INBOX` is always available and represents the user’s primary mailbox.

Most message operations require that you select or examine a folder before performing message operations. If you select a folder, you tell the IMAP server that you’ll be working with that folder. If you examine a folder, you do the same thing, but open the folder in read-only mode. Both `select()` and `examine()` provide folder summary information when called.

One of Twisted’s `fetch...()` commands can be used to download messages. These commands can download one or more messages at a time, and they can also download only certain parts of the messages.

Each IMAP message can have various flags. You can read the flags with `fetchFlags()` and set them with `setFlags()`, `addFlags()`, or `removeFlags()`. To delete a message, you would add the `\Deleted` flag and then call `expunge()`.

The `search()` function performs a server-side search. It takes one or more `Query` objects that specify the search criteria.

The `copy()` function copies a message from one folder to another on the server.

You can use `append()` to add a message to a folder. The `create()` and `delete()` functions create or remove whole folders. Unlike most commands, you don’t need to select or examine the folder before using these commands.



Part Four

General-Purpose Client Protocols

CHAPTER 13

FTP

FTP, THE FILE TRANSFER PROTOCOL, is one of the oldest and most widely used protocols on the Internet. As its name suggests, FTP is designed to transfer files from one location to another. It's bidirectional, meaning that it can be used to both upload and download files. Commands are provided to delete and rename files as well as to create and delete directories. FTP also provides features, benefiting UNIX/Linux servers most, that allow clients to alter permissions of files and directories on the server side. Some FTP servers support other features, such as automatic compression and archiving of directories.

In this chapter, you'll learn how to use Python's FTP interface, `ftplib`. Armed with that knowledge, you'll be able to automatically transfer files to and from remote machines, obtain lists of directories on FTP servers, and obtain information about those servers. You might use this to automatically upload the results of a data-processing job each night or to download a list of products for a user.

Understanding FTP

FTP has a long history, dating back to the early 1970s and predating the arrival of TCP. There are a number of features of FTP that are virtually never encountered today because they're of no use on modern computing platforms. Most of these features relate to systems that cannot support the modern notion of a file as being simply a stream of bytes whose interpretation is up to the application. Many FTP clients and servers don't support these features since they're virtually obsolete all but on paper.

In its present form, FTP is used to exchange files. It can send files in both directions, and with an appropriately intelligent client, two FTP servers can be made to exchange data with each other directly, without requiring the data to be downloaded to a client first.

NOTE The FTP standard is RFC959, available at www.faqs.org/rfcs/rfc959.html.



Communication Channels

FTP is unusual because it actually uses two TCP connections to get the job done. One connection is the control or command channel. Commands and brief responses, such as acknowledgments or error codes, are transmitted over that connection. The second connection is the data channel. This connection is used solely for transmitting file data or other blocks of information, such as directory listings. Technically, the data channel is full duplex, meaning that it allows files to be transmitted in both directions over it simultaneously. However, in actual practice, this capability is rarely used.

In the traditional sense, the process of downloading a file from an FTP server ran mostly like this:

1. First, the FTP client establishes a command connection by connecting to the FTP port on the server.
2. The client authenticates as appropriate.
3. The client changes to the appropriate directory on the server.
4. The client begins listening on a new port for the data connection, and then informs the server about that port.
5. The server connects to the port the client requested.
6. The file is then transmitted, and the data connection is closed.

This worked well in the early days of the Internet, for which every machine worth running FTP on had a public IP address and firewalls were relatively rare. Today, however, the picture is more complicated. Firewalls are the norm, and many people don't have real public IP addresses.

To accommodate this situation, FTP also supports what's known as passive mode. In this scenario, the server opens a port and tells the client where to connect. Other than that, everything behaves the same way.

Passive mode is the default with most FTP clients as well as Python's `ftplib` module.

Authentication and Anonymous FTP

Normally, when you connect to a remote FTP server, you'll provide a username and password to access the system. The FTP Protocol also provides for a third authentication token, called an *account*, which is rarely used. Once logged in,

you would typically have the same permissions on the remote system as if you were sitting at a terminal on the remote machine.

Many have decided that it would be useful to permit people without accounts to access certain areas of FTP servers. To permit this, visitors log in with the username “anonymous” and traditionally send their own e-mail address as a password. Servers configured this way are called *anonymous FTP servers*. Many large FTP sites use anonymous FTP so that anyone who wants files can get them. This is, in concept, similar to web servers for which no authentication is required before viewing documents. To the FTP client, the connection looks exactly the same as any other.

Using FTP in Python

The Python module `ftplib` is the primary interface to FTP for Python programmers. It handles the details of establishing the various connections for you, and provides convenient ways to automate some commands.

TIP If you’re only interested in downloading files, the `urllib2` module discussed in Chapter 6 also supports FTP and may be easier to use for simple downloading tasks. In this chapter, I describe `ftplib` because it provides FTP-specific features that aren’t available with `urllib2`.

Here’s a basic `ftplib` example. It connects to a remote server, displays the welcome message, and prints the current working directory.

```
=!/usr/bin/env python
= Basic connection - Chapter 13 - connect.py

from ftplib import FTP

f = FTP('ftp.ibiblio.org')
print "Welcome:", f.getwelcome()
f.login()

print "CWD:", f.getcwd()
f.quit()
```

The welcome message generally isn't of interest to a computer program, but may be interesting to a user running the program interactively. The `login()` function can take several parameters: a username, password, and account. If called without parameters, it will log in anonymously, sending the user anonymous and a generic anonymous password to the FTP server. That's fine in this case.

The `pwd()` function simply returns the current working directory on the remote site. The `quit()` function logs out and closes the connection. Here's an example session:

```
$ ./connect.py
Welcome: 220 ProFTPD Server (Bring it on...)
CWD: /
```

Downloading ASCII Files

FTP transfers normally use one of two modes: ASCII and binary (or image) mode. In ASCII mode, files are transferred line by line, which allows the client machine to store files with line endings appropriate on its platform. Here's a simple program that downloads a file in ASCII mode:

```
#!/usr/bin/env python
# ASCII download - Chapter 13 - asciidl.py
# Downloads README from remote and writes it to disk.

from ftplib import FTP

def writeline(data):
    fd.write(data + "\n")

f = FTP('ftp.kernel.org')
f.login()

f.cwd('/pub/linux/kernel')
fd = open('README', 'wt')
f.retrlines('RETR README', writeline)
fd.close()

f.quit()
```

The `cwd()` function changes the directory on the remote system. Then the `retrlines()` function begins the transfer. Its first parameter specifies a command to run on the remote system and usually is RETR, followed by a filename. Its second

parameter is a function that's called with each line retrieved; if omitted, the data is simply printed to standard output. The lines are passed with the end-of-line character stripped; so the `writeline()` function simply adds it on and writes the data out. To run this program, just run `./asciidl.py`. You'll see a file named `README` show up after it exits.

Downloading Binary Files

Basic binary file transfers work in very much the same way as text file transfers. Here's an example:

```
= /usr/bin/env python
= Binary download - Chapter 13 - binarydl.py

from ftplib import FTP

= = FTP('ftp.kernel.org')
.= login()

.= cwd('/pub/linux/kernel/v1.0')
= s = open('patch8.gz', 'wb')
.= retrbinary('RETR patch8.gz', fd.write)
= s.close()

.= quit()
```

In this example, a binary file is downloaded—note that after running it, the file `patch8.gz` will appear in your current working directory. The `retrbinary()` function simply passes blocks of data to the specified function. This is convenient, since a file object's `write()` function expects just such data. In this case, no custom function is necessary.

Advanced Binary Downloading

The `ftplib` module provides a second function that can be used for binary downloading: `transfercmd()`. This command provides a more low-level interface, but can be useful if you wish to know a little bit more about what's going on. This lets you keep track of the number of bytes transferred, and you can use that information to display status updates for the user. Here's a sample program that uses `transfercmd()`:

```

#!/usr/bin/env python
# Advanced binary download - Chapter 13 - advbinarydl.py

from ftplib import FTP
import sys

f = FTP('ftp.kernel.org')
f.login()

f.cwd('/pub/linux/kernel/v1.0')
f.voidcmd("TYPE I")

datasock, estsize = f.ntransfercmd("RETR linux-1.0.tar.gz")
transbytes = 0
fd = open('linux-1.0.tar.gz', 'wb')
while 1:
    buf = datasock.recv(2048)
    if not len(buf):
        break
    fd.write(buf)
    transbytes += len(buf)
    sys.stdout.write("Received %d " % transbytes)

# This "if" only passes if estsize is nonzero and is not None.
# That's exactly what we want, since if it's zero, we'd get a
# divide-by-zero error.
if estsize:
    sys.stdout.write("of %d bytes (%.1f%%)\r" % \
                    (estsize, 100.0 * float(transbytes) / float(estsize)))
else:
    sys.stdout.write("bytes\r")
    sys.stdout.flush()
sys.stdout.write("\n")
fd.close()
datasock.close()
f.voidresp()

f.quit()

```

There are a few new things to note here. First, there's a call to `voidcmd()`. This passes an FTP command directly to the server, checks for an error, but returns nothing. In this case, you run `TYPE I`. That sets the transfer mode to image, or binary. In the previous example, `retrbinary()` automatically ran that command behind the scenes, but `ntransfercmd()` doesn't.

Next, note that the `transfercmd()` returns a tuple consisting of a data socket and an estimated size. Always bear in mind that the size is merely an estimate and shouldn't be considered authoritative. Also, if no size estimate is available from the FTP server, the estimated size will be `None`.

The data socket is, in fact, a plain socket, so all the socket functions described in chapters 1 through 5 will work on it. In this example, you'll use a simple loop to `recv()` the data from the socket, write it out to disk, and print status updates.

After receiving the data, it's important to close the data socket and call `voidresp()`. The `voidresp()` function reads the response from the server and raises an exception if there was any error. Even if you don't care about detecting errors, if you fail to call `voidresp()` future commands will likely fail in strange ways because their results will be out of sync with the server. Here's an example of running this program:

```
$ ./advbinarydl.py
Received 1259161 of 1259161 bytes (100.0%)
```

Uploading Data

Data can, of course, be uploaded as well. Like downloading, there are two basic functions for uploading: `storbinary()` and `storlines()`. Both take a command to run and a file-like object. The `storbinary()` function will call `read()` on that object, while `storlines()` calls `readline()`. Note that this differs from the download functions for which you supply the function itself. Here's an example that uploads a file in binary mode:

```
#!/usr/bin/env python
# Binary upload - Chapter 13 - binaryul.py
# Arguments: host, username, localfile, remotepath

from ftplib import FTP
import sys, getpass, os.path

host, username, localfile, remotepath = sys.argv[1:]
password = getpass.getpass("Enter password for %s on %s: " % \
    (username, host))
f = FTP(host)
f.login(username, password)
```

```

f.cwd(remotePath)
fd = open(localfile, 'rb')
f.storbinary('STOR %s' % os.path.basename(localfile), fd)
fd.close()

f.quit()

```

This program indeed looks quite similar to earlier versions. Since most anonymous FTP sites don't permit file uploading, this example requires a server and logs in to it with your own account. It then changes to the specified directory, uploads the file, and exits. You can modify this program to upload a file in ASCII mode by simply changing `storbinary()` to `storlines()`.

Here's an example session:

```
$ ./binaryul.py ftp.example.com jgoerzen /bin/sh /tmp
Enter password for jgoerzen on ftp.example.com:
```

This would log in to `ftp.example.com` as `jgoerzen`, then upload the `/bin/sh` from my system to the `/tmp` directory on the remote.

Advanced Binary Uploading

Like the download process, it's also possible to upload files using `transfercmd()`, as follows:

```

#!/usr/bin/env python
# Advanced binary upload - Chapter 13 - advbinaryul.py
# Arguments: host, username, localfile, remotepath

from ftplib import FTP
import sys, getpass, os.path

host, username, localfile, remotepath = sys.argv[1:]
password = getpass.getpass("Enter password for %s on %s: " % \
    (username, host))
f = FTP(host)
f.login(username, password)

f.cwd(remotepath)
f voidcmd("TYPE I")

```

```

fd = open(localfile, 'rb')
datasock, esize = f.ntransfercmd('STOR %s' % os.path.basename(localfile))
esize = os.stat(localfile)[6]
transbytes = 0

while 1:
    buf = fd.read(2048)
    if not len(buf):
        break
    datasock.sendall(buf)
    transbytes += len(buf)
    sys.stdout.write("Sent %d of %d bytes (%.1f%%)\r" % (transbytes, esize,
        100.0 * float(transbytes) / float(esize)))
    sys.stdout.flush()
datasock.close()
sys.stdout.write("\n")
fd.close()
f voidresp()

f.quit()

```

Compared to the advanced binary download example, this one is very similar. Note the call to `datasock.close()` though. When uploading data, closing the socket is the signal to the server that the upload is complete. If you fail to close the data socket after uploading all your data, the server will be forever stuck waiting for the rest of the data to arrive. Here's an example session:

```

$ ./advbinaryul.py ftp.example.com jgoerzen /bin/sh /tmp
Enter password for jgoerzen on ftp.example.com:
Sent 628684 of 628684 bytes (100.0%)

```

Handling Errors

Like most Python modules, `ftplib` raises exceptions when errors occur. The `ftplib` module defines several exceptions, and it can also raise `socket.error` and `IOError`. As a convenience there's a tuple, called `ftplib.all_errors`, that includes all errors that could be raised by `ftplib`. This is often a useful shortcut. You can enclose your code in a `try: block`, and use `except ftplib.all_errors` to catch any error that may occur.

One of the problems with the basic `retrbinary()` function is that in order to use it easily, you'll usually wind up opening the file on the local end before beginning the transfer from the remote. If the file doesn't exist or the RETR command otherwise

fails, you'll have to close and delete the local file, or else wind up with a 0-length file. With the `ntransfercmd()` method, you can check for a problem prior to opening a local file. The previous example already follows these guidelines; if `ntransfercmd()` fails, the exception will cause the program to terminate before the local file is opened.

Scanning Directories

The FTP Protocol provides two ways to discover information about server files and directories. These are implemented in `ftplib` in the `nlst()` and `dir()` functions.

The `nlst()` function returns a list of entries in a given directory. You'll thus receive the names of all files and directories present. However, that is all that's given. There's no information on whether a particular entry is a file or a directory, on its size, or anything else.

The `dir()` function returns a directory listing from the remote. This listing is in a system-defined format, but typically contains a filename, size, modification date, and type. On UNIX servers, it's typically the output of `ls -l` or `ls -la`. Windows servers may use the output of `dir`. Other systems may have their own specific formats. While the output may often be useful to an end user, it's difficult for a program to use it due to the varying output formats. Some clients that need this data implement parsers for the many different types of output formats, or the one type in use in a particular situation.

Here's an example of using `nlst()` to get directory information:

```
#!/usr/bin/env python
# NLST example - Chapter 13 - nlst.py

from ftplib import FTP

f = FTP('ftp.kernel.org')
f.login()

f.cwd('/pub/linux/kernel')
entries = f.nlst()
entries.sort()

print "%d entries:" % len(entries)
for entry in entries:
    print entry
f.quit()
```

When you run this program, you'll see output like this:

```
$ ./nlst.py
22 entries:
COPYING
CREDITS
Historic
README
SillySounds
crypto
people
ports
projects
testing
uemacs
v1.0
v1.1
v1.2
...
```

If you were to use an FTP client to manually log on to the server, you'd see the same files listed. Notice that the filenames are in a convenient format for automated processing—a list of filenames—but there's no extra information. Now contrast that with the output from this example, which uses `dir()`:

```
#!/usr/bin/env python
# dir() example - Chapter 13 - dir.py

from ftplib import FTP

f = FTP('ftp.kernel.org')
f.login()

f.cwd('/pub/linux/kernel')
entries = []
f.dir(entries.append)

print "%d entries:" % len(entries)
for entry in entries:
    print entry
f.quit()
```

Notice the call to `f.dir()`. It can take a function that's called for each line, just like `retrlines()`. This capability is handy for adding things to a list; `entries.append()` is called for each line and adds it to the list. The output of this program looks like this:

```
$ ./dir.py
22 entries:
-r--r--r--  1 korg      korg          18458 Mar 13 1994 COPYING
-r--r--r--  1 korg      korg          36981 Sep 16 1996 CREDITS
drwxrwsr-x  4 korg      korg          4096 Mar 20 2003 Historic
-r--r--r--  1 korg      korg          12056 Sep 16 1996 README
drwxrwsr-x  2 korg      korg          4096 Apr 14 2000 SillySounds
drwxrwsr-x  5 korg      korg          4096 Nov 24 2001 crypto
drwxrwsr-x  54 korg     korg          4096 Sep 16 19:52 people
drwxrwsr-x  6 korg      korg          4096 Mar 13 2003 ports
drwxrwsr-x  3 korg      korg          4096 Sep 16 2000 projects
drwxrwsr-x  3 korg      korg          4096 Feb 14 2002 testing
drwxrwsr-x  2 korg      korg          4096 Mar 20 2003 uemacs
drwxrwsr-x  2 korg      korg          4096 Mar 20 2003 v1.0
drwxrwsr-x  2 korg      korg          20480 Mar 20 2003 v1.1
drwxrwsr-x  2 korg      korg          8192 Mar 20 2003 v1.2
...

```

This time the list contains lines of output from `ls -l` on the server. While there's a lot more information for the user, this is a lot more difficult to process for the programmer.

Parsing UNIX Directory Listings

If you have a directory listing like the previous one, you'll notice that most UNIX servers provide a directory listing in pretty much the same format. If you have listings like the previous one, you process them with code like this:

```
#!/usr/bin/env python
# dir() parsing example - Chapter 13 - dirparse.py

from ftplib import FTP

class DirEntry:
    def __init__(self, line):
        self.parts = line.split(None, 8)
```

```

def isvalid(self):
    return len(self.parts) >= 6

def gettype(self):
    """Returns - for regular file; d for directory; l for symlink."""
    return self.parts[0][0]

def getfilename(self):
    if self.gettype() != 'l':
        return self.parts[-1]
    else:
        return self.parts[-1].split(' -> ', 1)[0]

def getlinkdest(self):
    if self.gettype() == 'l':
        return self.parts[-1].split(' -> ', 1)[1]
    else:
        raise RuntimeError, "getlinkdest() called on non-link item"

class DirScanner(dict):
    """A dictionary object that can load itself up with keys being filenames
    and values being DirEntry objects. Call addline() from your FTP dir()
    call."""

    def addline(self, line):
        obj = DirEntry(line)
        if obj.isvalid():
            self[obj.getfilename()] = obj

f = FTP('ftp.kernel.org')
f.login()

f.cwd('/pub/linux/kernel')
d = DirScanner()
f.dir(d.addline)

print "%d entries: % len(d.keys())"
for key, value in d.items():
    print "%s: type %s" % (key, value.gettype())

f.quit()

```

The `DirEntry` class represents a single entry in a directory. It defines an `isValid()` function, which attempts to guess whether the line in question is valid. The check is simplistic, but will catch extra nonentry lines that sometimes occur in directory listings. The remaining functions, such as `gettype()` and `getfilename()`, simply return information by parsing the listing line.

`DirScanner` subclasses the Python built-in dictionary object, adding one additional method: `addline()`, which takes a line of data from `dir()` and, if it's valid, adds it to the dictionary. The key will be the filename listed on that line.

In the program, it's a simple matter to create the `DirScanner` object and have `f.dir()` call `addline()` for each line of information. Finally, the results are printed. They'll look somewhat like this:

```
$ ./dirparse.py
22 entries:
people: type d
testing: type d
COPYING: type -
Historic: type d
v2.6: type d
v2.4: type d
v2.5: type d
v2.2: type d
v2.3: type d
v2.0: type d
v2.1: type d
...
...
```

Discovering Information Without Parsing Listings

There are caveats to parsing directory listings, especially if the remote server doesn't return UNIX-format listings. By combining the output of `nlst()` with some other commands and error checking, it's possible to at least determine whether a given file is a regular file or a directory. This can all be done without having to use `dir()` results in any way, and it's done in a completely platform-independent manner. So it will work not just with Unix servers, but also with Windows, Mac OS X, and VMS servers. Here's an example:

```
#!/usr/bin/env python
# nlst() with file/directory detection example - Chapter 13
# nlstscan.py

import ftplib

class DirEntry:
    def __init__(self, filename, ftpobj, startingdir = None):
        self.filename = filename
        if startingdir == None:
            startingdir = ftpobj.pwd()
        try:
            ftpobj.cwd(filename)
            self.filetype = 'd'
            ftpobj.cwd(startingdir)
        except ftplib.error_perm:
            self.filetype = '-'

    def gettype(self):
        """Returns - for regular file; d for directory."""
        return self.filetype

    def getfilename(self):
        return self.filename

f = ftplib.FTP('ftp.kernel.org')
f.login()

f.cwd('/pub/linux/kernel')
nitems = f.nlst()
items = [DirEntry(item, f, f.pwd()) for item in nitems]

print "%d entries:" % len(items)
for item in items:
    print "%s: type %s" % (item.getfilename(), item.gettype())
f.quit()
```

The key to this program is in `DirEntry`'s `__init__()` method. For each result from `nlst()`, the program attempts to change to a directory with that name. If this succeeds, the program knows that the result is the name of an actual directory, and it changes back to the starting directory. If the `cwd()` attempt fails, it will raise `ftplib.error_perm`, which is a good indication that the item in question isn't a directory. In some cases, this logic may be fooled; for instance, if the user doesn't have permission to use `cwd()` to change to the directory, this code may assume it has a file to deal with; but a subsequent attempt to download the file will fail. If you're concerned about this rare case, you could also attempt to start downloading a file. If that fails, you know you have something to which you don't have permission, but still don't know what.

This program will usually be slower than parsing the output from `dir()` since it has to issue so many different `cwd()` commands. However, the output from this program looks just like the output from the program that parsed the `dir()` result.

Downloading Recursively

Now that you're able to obtain directory listings and differentiate files from directories, it's possible to download entire directory trees from an FTP server. The following example downloads an entire directory tree from the Linux kernel repository.

```
#!/usr/bin/env python
# Recursive downloader - Chapter 13 - recursedl.py

from ftplib import FTP
import os, sys

class DirEntry:
    def __init__(self, line):
        self.parts = line.split(None, 8)

    def isvalid(self):
        return len(self.parts) >= 6

    def gettype(self):
        """Returns - for regular file; d for directory; l for symlink."""
        return self.parts[0][0]
```

```

def getfilename(self):
    if self.gettype() != 'l':
        return self.parts[-1]
    else:
        return self.parts[-1].split(' -> ', 1)[0]

def getlinkdest(self):
    if self.gettype() == 'l':
        return self.parts[-1].split(' -> ', 1)[1]
    else:
        raise RuntimeError, "getlinkdest() called on non-link item"

class DirScanner(dict):
    def addline(self, line):
        obj = DirEntry(line)
        if obj.isinvalid():
            self[obj.getfilename()] = obj

def downloadfile(ftplibobj, filename):
    ftpobj.voidcmd("TYPE I")
    datasock, estsize = ftpobj.ntransfercmd("RETR %s" % filename)
    transbytes = 0
    fd = open(filename, 'wb')
    while 1:
        buf = datasock.recv(2048)
        if not len(buf):
            break
        fd.write(buf)
        transbytes += len(buf)
        sys.stdout.write("%s: Received %d " % (filename, transbytes))
        if estsize:
            sys.stdout.write("of %d bytes (%.1f%%)\r" % (estsize,
                100.0 * float(transbytes) / float(estsize)))
        else:
            sys.stdout.write("bytes\r")
    fd.close()
    datasock.close()
    ftpobj.voidresp()
    sys.stdout.write("\n")

```

```

def downloaddir(ftpobj, localpath, remotepath):
    print "*** Processing directory", remotepath
    localpath = os.path.abspath(localpath)
    oldlocaldir = os.getcwd()
    if not os.path.isdir(localpath):
        os.mkdir(localpath)
    olldir = ftpobj.pwd()
    try:
        os.chdir(localpath)
        ftpobj.cwd(remotepath)
        d = DirScanner()
        f.dir(d.addline)

        for filename, entryobj in d.items():
            if entryobj.gettype() == '-':
                downloadfile(ftpobj, filename)
            elif entryobj.gettype() == 'd':
                downloaddir(ftpobj, localpath + '/' + filename,
                           remotepath + '/' + filename)
            # Re-display directory info
            print "*** Processing directory", remotepath
    finally:
        os.chdir(oldlocaldir)
        ftpobj.cwd(olldir)

f = FTP('ftp.kernel.org')
f.login()

downloaddir(f, 'old-versions', '/pub/linux/kernel/Historic/old-versions')

f.quit()

```

The `DirEntry` and `DirScanner` classes are the same as in the earlier `dir()`-based scanner. The `downloadfile()` function works like the advanced binary download example. The `downloaddir()` function is the heart of the program. It takes a local directory and a remote directory. It then scans the remote directory and looks at each entry.

If a given entry is a file, `downloadfile()` is called to download it. If the entry is a directory, then `downloaddir()` calls itself to process the new directory. At the end of processing, the `finally` clause ensures that working directories are back where they were when the function was first invoked. When you run this program, you'll see output somewhat like this:

```
$ ./recessedl.py
*** Processing directory /pub/linux/kernel/Historic/old-versions
linux-0.96a.tar.bz2: Received 174003 of 174003 bytes (100.0%)
linux-0.96b.patch2.bz2.sign: Received 248 of 248 bytes (100.0%)
linux-0.96b.patch2.gz: Received 5825 of 5825 bytes (100.0%)
RELNOTES-0.95a: Received 6257 of 6257 bytes (100.0%)
*** Processing directory /pub/linux/kernel/Historic/old-versions/tytso
linux-0.10.tar.sign: Received 248 of 248 bytes (100.0%)
linux-0.10.tar.bz2.sign: Received 248 of 248 bytes (100.0%)
linux-0.10.tar.bz2: Received 90032 of 90032 bytes (100.0%)
linux-0.10.tar.gz.sign: Received 248 of 248 bytes (100.0%)
linux-0.10.tar.gz: Received 123051 of 123051 bytes (100.0%)
*** Processing directory /pub/linux/kernel/Historic/old-versions
linux-0.96a.patch4.bz2.sign: Received 248 of 248 bytes (100.0%)
linux-0.96a.patch3.gz.sign: Received 248 of 248 bytes (100.0%)
linux-0.95.tar.bz2.sign: Received 248 of 248 bytes (100.0%)
...
...
```

Here, you can see the program began processing with the `old-versions` directory. It downloaded four files from that directory, then it encountered the `tytso` subdirectory. The program downloaded all files from `tytso`, then resumed processing other files in `old-versions`.

Manipulating Server Files and Directories

The `ftplib` module provides several simple methods for manipulating files and directories on the server. With these methods, you can do things like delete files and directories and do renames.

Deleting Files and Directories

You can call `delete()` to delete a file. You can also use `rmd()` to delete a directory, though most systems require the directory to be empty before it can be deleted. Both functions take a single parameter—a file or directory name—and will raise an exception if an error occurs.

Creating Directories

The `mkd()` function takes a single parameter and creates a new directory that has that name. There must not be any existing file or directory with the name. If there is, or any other error occurs, an exception will be raised.

Moving and Renaming Files

The `rename()` function takes two parameters: the name of an existing file and a new name for the file. It works essentially the same as the UNIX command `mv`. If both names are in the same directory, the file is essentially renamed. If the destination specifies a name in a different directory, the file is moved.

Summary

FTP lets you transfer files between your machine and a remote FTP server. In Python, the `ftplib` library is used to talk to FTP servers.

FTP supports binary and ASCII transfers. ASCII transfers are usually used for text files, and permit line endings to be adjusted as the file is transferred. Binary transfers are used for everything else. The `retrlines()` function is used to download a file in ASCII mode, while `retrbinary()` downloads a file in binary mode.

You can also upload files to a remote server. The `storlines()` function uploads a file in ASCII mode, and `storbinary()` uploads a file in binary mode.

The `ntransfercmd()` function can be used for binary uploads and downloads. It gives you more control over the transfer process and is often used to present status updates for the user.

The `ftplib` module raises exceptions on errors. The special tuple `ftplib.all_errors` can be used to catch any error that it might raise.

You can use `cwd()` to change to a particular directory on the remote end. The `nlist()` command returns a simple list of all entries (files or directories) in a given directory. The `dir()` command returns a list in server-specific format, which often includes more details. Even with `nlist()`, you can usually detect whether an entry is a file or directory by attempting to use `cwd()` to change to it and noting whether you get an error.

Database Clients

DATABASES IN DIFFERENT FORMS are becoming quite commonplace. They're used to store everything from customer names and addresses to support ticket incidents and bugs.

There are many different databases available. Python programmers typically use one of two different types: SQL relational databases and local file (dbm) databases. This chapter focuses on SQL databases; dbm databases are typically used for small projects and don't generally involve networking. This chapter also assumes that you're already familiar with SQL and operating your chosen SQL server.

In this chapter, you'll learn how to integrate your Python programs with SQL databases. By doing that, you can accomplish everything from storing login information for a CGI script to accounting information for a bookkeeping program.

SQL and Networking

All of the most popular SQL database servers support networking. This capability allows the programs that use the database to run on a different machine from the database server itself. The database server then can receive queries over the network, gather the data, and return the results over the network.

This provides an excellent solution for many people. A database server may be specifically optimized with large and fast disks. Applications may be installed on workstations all over a company, or may even query the database over the Internet. However, there are some additional issues to consider when accessing a database server over a network.

One of those issues is reliability. Modern SQL database servers provide a way to ensure that a given modification to the database is either carried out in its entirety or not at all. However, there are circumstances in which an application may not know whether the transaction was carried out. For instance, if no acknowledgment is received from the database, the client has no way of knowing if the server crashed and didn't commit the transaction, or if the network just went down and couldn't return a result.

Another issue is security. Most database connections are unencrypted for performance reasons. This may make them unsuitable for use over the public Internet, or even in certain LAN situations. Some database servers offer optional encryption using SSL. This should be used if the client must transmit data over an insecure network to the server.

SQL in Python

Python, naturally, supports many different databases. However, the network protocol for communicating with databases differs from one vendor's server to the next. In the early days of Python, each database had its own Python module, and all of those modules worked differently and provided different functions.

Many Python developers disliked that situation, since it made it impossible to write code that would work across different database server types. A specification called DB-API was created. All modules for connecting to a database would provide a common interface, even if the underlying network protocol were quite different. This is a similar concept to Java Database Connectivity (JDBC) or Open Database Connectivity (ODBC).

The current version of the DB-API specification is 2.0. You may find both the specification itself and links to various modules at the Python Database Topic Guide at www.python.org/topics/database/. As of the time of this writing, the following databases are known to support DB-API version 2.0:

- DB/2
- Gadfly (DB-API compliance is uncertain)
- Ingres and OpenIngres
- JDBC (driver conversion layer; when run under Java, can access any database with JDBC drivers)
- MySQL 3.22.19 and above
- ODBC (driver conversion layer; when run on a system with ODBC installed, can access any database configured with ODBC)
- Oracle
- PostgreSQL
- SAP DB
- Sybase (mostly DB-API 2.0 compliant)
- ThinkSQL

If your database isn't on this list, you may still be able to work with it. The list only mentions those modules that are DB-API 2.0 compliant; some modules aren't actively maintained and may only support version 1 of the specification. Older modules may exist for Python that predate DB-API itself.

If no Python module is available, you can also try the JDBC or ODBC driver conversion layer. For instance, if your database provides JDBC drivers, you can still write your program in Python. You'll then run it under the Jython interpreter (a Python interpreter written in Java instead of C), which has access to the Java JDBC system. The zxJDBC DB-API module makes calls into JDBC instead of directly to a database server. Between all these mechanisms, it's quite rare indeed to find a database server that you cannot communicate with from Python. zxJDBC will be covered in detail later in this chapter. Two ODBC interfaces are available at the Python Database Topic Guide site.

There's also one item in the preceding list to highlight specifically: Gadfly. Gadfly is a small SQL database written in pure Python. If you don't require extreme performance or scalability, you may be able to simply embed Gadfly into your programs. Gadfly's home page is <http://gadfly.sourceforge.net/>.

For simplicity and clarity, examples in this chapter (except where noted) are written for PostgreSQL. If you're using a different database server, small modifications may be necessary.

You'll probably need to obtain database drivers, since Python doesn't ship any by default. The Python Database Topic Guide site has links to the sites for many Python drivers. You can also check with your operating system or distribution provider for prebuilt packages.

Connecting

The process of connecting is the one thing that varies most from one database server to the next. In this section, you'll learn how to connect using three different database modules. Please consult the documentation for your particular database server to obtain greater detail.

In every case, once you've connected to the database, you'll have a database handle (commonly referred to as `dbh`). This is a class that represents the connection, and is your primary interface into the database system. Even if you're not running Postgres, you should be able to simply replace the connection code in the Postgres examples with the appropriate code for your database, and achieve a working result.

PostgreSQL

Several different modules exist for connecting to PostgreSQL from Python. In this chapter, I use `psycopg`, which you can download from <http://initd.org/software/initd psycopg>. The `psycopg` module has been tested and found to conform to DB-API 2.0 on multiple platforms.

If you don't already have a PostgreSQL server, you may obtain the PostgreSQL system at no charge from www.postgresql.org. PostgreSQL should run on almost any platform that Python supports and provides a robust implementation of ANSI SQL.

The `psycopg connect()` call takes a string that contains all the information required to establish a connection to the remote server. In the following example, there's a `getdsn()` function that generates this string:

```
#!/usr/bin/env python
# Basic connection to PostgreSQL with psycopg - Chapter 14
# connect_psycopg.py

import psycopg

def getdsn(db = None, user = None, passwd = None, host = None):
    if user == None:
        # Default user to the one they're logged in as
        import os, pwd
        user = pwd.getpwuid(os.getuid())[0]
    if db == None:
        # Default to the username.
        db = user
    dsn = 'dbname=%s user=%s' % (db, user)
    if passwd != None:
        dsn += ' password=' + passwd
    if host != None:
        dsn += ' host=' + host
    return dsn

dsn = getdsn()
print "Connecting to %s" % dsn
dbh = psycopg.connect(dsn)
print "Connection successful."
dbh.close()
```

MySQL

The Python interface to MySQL is known variously as MySQLdb or MySQL-Python. It can be downloaded from <http://sourceforge.net/projects/mysql-python> and is DB-API 2.0 compliant.

If you don't already have a MySQL server, you can obtain one at no charge from www.mysql.org. MySQL runs on multiple platforms and puts its greatest emphasis on performance.

Unlike PostgreSQL, the MySQLdb connect() function takes various parameters rather than a single string. The most important parameters are

- user. The username to use when connecting. Defaults to the current logged-in username as the preceding PostgreSQL example did.
- passwd. The password to use. No default.
- db. The database to connect to. No default.
- host. The hostname of the database. If not specified, connects to the database server running on the local machine.
- port. The TCP port number. Defaults to 3306, the MySQL default.

Here's an example of connecting to a database named `foo`, using all the other defaults:

```
#!/usr/bin/env python
# Basic connection to MySQL with mysqladb - Chapter 14
# connect_mysqladb.py

import MySQLdb

print "Connecting..."
dbh = MySQLdb.connect(db = "foo")
print "Connection successful."
dbh.close()
```

Jython zxJDBC

This method is somewhat different from all the others in several ways. First, and most obviously, it's different because the standard Python interpreter isn't used. Instead, the Jython interpreter is required. If you don't have Jython already, you can download it from www.jython.org.

The standard Python interpreter is written in C. Jython, on the other hand, is written in Java. This means it can run on any platform that supports Java. It also means that Python programs that run under Jython have access to Java objects, APIs, and methods.

Java provides a standard for database connectivity called Java Database Connectivity (JDBC). In concept, JDBC is similar to Python's DB-API, but the actual interfaces are significantly different. The zxJDBC module is a bridge between DB-API and JDBC. It translates DB-API calls into their equivalent JDBC ones, and translates results back to DB-API formats.

Many commercial databases that don't support Python do support JDBC. Therefore, you can still write portable Python code that can run on normal Python or with zxJDBC. Databases with normal DB-API modules can be supported directly; the rest can use zxJDBC.

Before you can use zxJDBC, you must have JDBC working. First, of course, you must have a working Java Runtime Environment. These can be obtained from various vendors, which may vary by platform.

Next, you'll need to make sure that your JDBC driver is available to Java. In many cases, this is as simple as adding a directory or JAR file to your CLASSPATH. The method of setting this as a default varies between implementations; however, you can often just set an environment variable named CLASSPATH.

Finally, you need to know the Java name of your JDBC driver module and what parameters to give to it upon connection. zxJDBC will pass these values through to Java for the initial connection.

Here's an example of establishing a zxJDBC connection with PostgreSQL:

```
#!/usr/bin/env /jython
# Basic connection through zxJDBC to PostgreSQL - Chapter 14
# connect_zxjdbc.py

from com.ziclix.python.sql import zxJDBC
import os

dbh = zxJDBC.connect('jdbc:postgresql://localhost/foo',
                     'jgoerzen', None, 'org.postgresql.Driver')
print "Connection successful."
dbh.close()
```

This example will connect to the database named `foo` on the local machine, supplying the username `jgoerzen` with no password.

If this example fails for you, with a message about failing to find `com.ziclix`, or driver handler errors, it's possible that your copy of Jython doesn't include the full zxJDBC implementation. You can download the full Jython from www.jython.org to fix the problem.

As a final note, be aware that the current Jython release as of the time of this writing corresponds to Python 2.1. Therefore, if you want your code to be compatible with Jython and zxJDBC, you must not use any Python features more recent than 2.1. That said, the Jython team is at work updating its Python compliance, so a newer version may be available by the time you read this.

Executing Commands

Now that you know how to connect to your chosen database server, it's time to actually send commands to it. These examples will use PostgreSQL, but should work with most other database servers as well. If you're using a different server, you can simply replace the connection code with the appropriate code for your server and usually arrive at a working example. (This connection code is the same in every example.) You'll also need a working database server, a database present, and an account on that server. These examples are designed to be executed in order.

To run any command, you must first obtain a *cursor*. The cursor concept is designed to help make it easy to handle results from queries, but for now, we'll focus on commands that return no results. Here's a basic program that creates a table and loads some data into it:

```
#!/usr/bin/env python
# Execute - Chapter 14 - execute.py

import psycopg

def getdsn(db = None, user = None, passwd = None, host = None):
    if user == None:
        # Default user to the one they're logged in as
        import os, pwd
        user = pwd.getpwuid(os.getuid())[0]
    if db == None:
        # Default to the username.
        db = user
    dsn = 'dbname=%s user=%s' % (db, user)
    if passwd != None:
        dsn += ' password=' + passwd
    if host != None:
        dsn += ' host=' + host
    return dsn
```

```

dsn = getdsn()
print "Connecting to %s" % dsn
dbh = psycopg.connect(dsn)
print "Connection successful.

cur = dbh.cursor()
cur.execute("""CREATE TABLE ch14 (
    mynum      integer UNIQUE,
    mystring   varchar(30)""")

cur.execute("INSERT INTO ch14 VALUES (5, 'Five')")
cur.execute("INSERT INTO ch14 VALUES (0)")
dbh.commit()
dbh.close()

```

Let's look at how this code works. First, the program connects in the same manner as before. Next, it creates a cursor object. After that, it calls `execute()` three times. Notice that it never actually checks the return value of this call. If there was a problem with it on the server, an exception would be raised.

The last line in the program calls `commit()`. Most modern databases will not actually make any changes to the database itself unless you specifically call `commit()` to commit them to disk.

Transactions

Most modern database servers support transactions. Transactions group together modifications to the database into a single command.

Transactions are an important tool for database reliability. With complex databases, a single logical change may well affect many different tables and require several queries to carry out. For instance, adding a new customer may require an addition to a customer table as well as one to an address table.

If one of those additions is carried out, but the other one wasn't for some reason—perhaps due to an error or a server crash—the database will be in an inconsistent state. Software that expects both tables to have linked entries will be confused.

A transaction guarantees that either all changes will be applied or none of them will. In addition, it guarantees that no changes are applied until `commit()` is called.

In the previous example, consider what would happen if an error occurred before the second `INSERT` statement. An exception would be raised, and the program would terminate. Since `commit()` was never called, no changes would be

made to the database. The table would not have been created, and the first row would not have been inserted.

There's a companion function called `rollback()`. This function effectively discards any changes since the last call to `commit()` or `rollback()`. This function is useful if you discover an error and wish to cancel the transaction being sent.

You should also use `rollback()` in exception handlers. Some database servers may not behave as expected if you attempt to `commit()` changes after a server error occurred.

Performance Implications of Transactions

The performance of transactions is highly dependent on the specific server implementation. However, some general rules do apply:

- Committing after each individual command is often the slowest method of updating a database.
- Committing extremely large amounts of data at once may overflow the server's transaction buffer and cause errors or crashes.

Let's say you have a large list of data to insert into a database—perhaps you're importing 50,000 rows. For each row, you'll run an `INSERT INTO` command. You might be tempted to call `commit()` after each row, since that's all there is to the transaction. That will work, but it will be slow, and will require you to be able to handle partial loads in your code.

You might also consider just doing one `commit()` after all 50,000 `INSERT` commands have been sent. That might not work, depending on your database. You might overflow your server's RAM by causing it to try to store those 50,000 commands in its `commit()` buffer. Some servers will be fine with this, but if you want your code to be portable to multiple databases, you shouldn't do it.

A good compromise would be to call `commit()` after every 100 or so rows. That way, you avoid the performance penalty of committing all the time while avoiding filling up your server's buffers. However, you'll need to make sure you can deal with a problem if the server crashes. Your program will have to either empty out the table, or be sure not to reinsert the same row twice.

Hiding Changes Until You're Finished

One interesting side effect of the transaction system is that other users and programs won't see the changes you're making until you commit them. Here's an example that takes advantage of that concept:

```

#!/usr/bin/env python
# Commit example - Chapter 14 - commit.py

import psycopg

def getdsn(db = None, user = None, passwd = None, host = None):
    if user == None:
        # Default user to the one they're logged in as
        import os, pwd
        user = pwd.getpwuid(os.getuid())[0]
    if db == None:
        # Default to the username.
        db = user
    dsn = 'dbname=%s user=%s' % (db, user)
    if passwd != None:
        dsn += ' password=' + passwd
    if host != None:
        dsn += ' host=' + host
    return dsn

dsn = getdsn()
print "Connecting to %s" % dsn
dbh = psycopg.connect(dsn)
print "Connection successful."

cur = dbh.cursor()
cur.execute("DELETE FROM ch14")
cur.execute("INSERT INTO ch14 VALUES (0)")

dbh.commit()
dbh.close()

```

This program replaces the contents of the table with a single row. Some automated import programs may run in a periodic basis, replacing the contents of a table with new data.

The nice part about using transactions in this scenario is that other users are still able to read from the table, even after you've sent your `DELETE FROM` command. That's possible since they don't actually see the changes until `commit()` is called. So even if it takes you half an hour to load all the new data into a table, the users will still see the full old version until you call `commit()`.

CAUTION DB-API doesn't define whether changes not yet committed are visible to the *same* program in which they're made. You may or may not see your uncommitted changes if you run some queries over the data. In most cases, establishing a second connection to the database will guarantee that you don't see the pending changes.

Also, remember that some databases don't support transactions. On those systems, all changes are made immediately, `commit()` does nothing, and `rollback()` will cause an error. All the well-known modern free and commercial databases do support transactions, however.

Repeating Commands

A common problem database programmers face is issuing many similar commands. For instance, you might need to load thousands of new records into a table. With SQL that means issuing thousands of `INSERT INTO` commands.

This can be inefficient. SQL is an actual language so the database server must interpret your commands. Also, network latency can be a problem; after each command, the client must generally await a response to check for errors.

Many SQL servers support optimizations that can reduce the performance problems associated with repeating commands many times. With Python's DB-API, the solution can take one of two forms: sending a command along with a list of data, or sending the same command several times with different data.

Parameter Styles

Normally, a command has the data embedded. For instance, `INSERT INTO ch14 VALUES (12, 'Twelve')` contains the actual data (12 and Twelve) that will be added to the table. In order to be able to avoid reinterpreting the command for each record, there must be a way to separate the data from the command.

Consider, for example, that you wish to add the following rows to the database:

```
12    Twelve
13    Thirteen
14    Fourteen
15    Fifteen
```

You might translate that into Python code like this:

```
cur.execute("INSERT INTO ch14 VALUES (12, 'Twelve')")
cur.execute("INSERT INTO ch14 VALUES (13, 'Thirteen')")
cur.execute("INSERT INTO ch14 VALUES (14, 'Fourteen')")
cur.execute("INSERT INTO ch14 VALUES (15, 'Fifteen')")
```

But that's inefficient since all these calls must be made separately and interpreted separately. What we need is a way to combine similar calls like this into a single command set. DB-API provides such a thing.

Unfortunately, for DB-API programmers there are *five* ways to do this. Each database module can choose which to support. Your program must use the method supported by your particular database module. If you use a different method, your program will not work. This program will show you which method your database uses:

```
#!/usr/bin/env python
# Parameter style - Chapter 14 - paramstyle.py

import psycopg
print psycopg.paramstyle
```

Each database module is required to declare a `paramstyle` variable that you can check. The parameter style defines the format of placeholders added to the code. Here are the possible styles, in order from least to most preferred according to the DB-API specification:

- `qmark`. Denotes the question-mark style. Each bit of data in the command string will be replaced with a question mark, and the arguments will be given as a list or tuple. For instance, `INSERT INTO ch14 VALUES (?, ?)`.
- `format`. Uses `printf()`-style format codes, without supporting the Python extensions for named parameters. It takes a list or a tuple to convert. For instance, `INSERT INTO ch14 VALUES(%d, %s)`.
- `numeric`. Denotes the numeric style. Each bit of data in the command string will be replaced with a colon followed by a number (starting from 1), and the arguments will be given as a list or tuple. For instance, `INSERT INTO ch14 VALUES(:1, :2)`.

- `named`. Denotes the named style. It's similar to `numeric`, but uses names instead of numbers after the colon. It takes a dictionary to convert. For instance, `INSERT INTO ch14 VALUES(:number, :text)`.
- `pyformat`. Supports Python-style named parameters and takes a dictionary to convert. For instance, `INSERT INTO ch14 VALUES(%(number)d, %(text)s)`.

Of the three database modules discussed in this chapter, PostgreSQL uses `pyformat`, MySQL uses `format`, and zxJDBC uses `qmark`. In this chapter, `pyformat` will be demonstrated.

Using `executemany()`

The `executemany()` function takes a command and a list of records that it runs the command against. Each record in that list is either a list itself or a dictionary, depending on the parameter style used by the database module. Here's an example:

```
#!/usr/bin/env python
# executemany() example - Chapter 14 - executemany.py

import psycopg2

rows = ({'num': 0, 'text': 'Zero'},
        {'num': 1, 'text': 'Item One'},
        {'num': 2, 'text': 'Item Two'},
        {'num': 3, 'text': 'Three'})

def getdsn(db = None, user = None, passwd = None, host = None):
    if user == None:
        # Default user to the one they're logged in as
        import os, pwd
        user = pwd.getpwuid(os.getuid())[0]
    if db == None:
        # Default to the username.
        db = user
    dsn = 'dbname=%s user=%s' % (db, user)
    if passwd != None:
        dsn += ' password=' + passwd
    if host != None:
        dsn += ' host=' + host
    return dsn
```

```

dsn = getdsn()
print "Connecting to %s" % dsn
dbh = psycopg.connect(dsn)
print "Connection successful."

cur = dbh.cursor()
cur.execute("DELETE FROM ch14")
cur.executemany("INSERT INTO ch14 VALUES (%(num)d, %(text)s)", rows)
dbh.commit()
dbh.close()

```

This program will insert the four rows defined in `rows`. This will happen in whatever way is most efficient for the database. With most database back-ends, optimizations can occur here. However, some old or small databases may not support any optimizations. In those cases, `executemany()` will still work but will not provide performance benefits.

NOTE No quote marks were necessary around the strings, even though this is required for SQL. Database modules that implement `pyformat` will automatically insert those for a string format.

Handling Situations That Are Inappropriate for `executemany()`

While `executemany()` works well in a great many situations, there are still situations in which it's inappropriate. One of the main drawbacks of `executemany()` is that it requires the data for all rows to be in memory prior to running any commands. This is a problem if your data set is huge; it may exceed the memory resources of the system.

It's still possible to obtain some performance optimization out of `execute()` when `executemany()` is inappropriate for your needs. According to the DB-API specification, when `execute()` is called periodically, database back-ends can perform optimizations. But its first argument must be pointing to the same object and not just a string containing the same value; rather, the exact same string object in memory. Like `executemany()`, it's not guaranteed that they can do any optimization, and it's also expected that `execute()` will still not be as fast as `executemany()`. But if you're unable to use `executemany()`, this is the next best option.

```

#!/usr/bin/env python
# optimized execute() for multiple rows example - Chapter 14
# execute-multiple.py

import psycopg

rows = ( {'num': 0, 'text': 'Zero'},
         {'num': 1, 'text': 'Item One'},
         {'num': 2, 'text': 'Item Two'},
         {'num': 3, 'text': 'Three'})

def getdsn(db = None, user = None, passwd = None, host = None):
    if user == None:
        # Default user to the one they're logged in as
        import os, pwd
        user = pwd.getpwuid(os.getuid())[0]
    if db == None:
        # Default to the username.
        db = user
    dsn = 'dbname=%s user=%s' % (db, user)
    if passwd != None:
        dsn += ' password=' + passwd
    if host != None:
        dsn += ' host=' + host
    return dsn

dsn = getdsn()
print "Connecting to %s" % dsn
dbh = psycopg.connect(dsn)
print "Connection successful."

cur = dbh.cursor()
cur.execute("DELETE FROM ch14")

# It is best to set this query before the loop!
query = "INSERT INTO ch14 VALUES (%(num)d, %(text)s)"

for row in rows:
    cur.execute(query, row)
dbh.commit()
dbh.close()

```

Although in this case, the `rows` list is still held in memory, you could just as easily create the individual `row` dictionary on the fly based on data read in from a

file or from some other data source. Since you can now execute the command for one row at a time, there's no longer a problem with consuming all your memory.

However, bear in mind the warning from the transactions section earlier; if you're working with extremely large data sets, it's best to call `commit()` periodically.

Retrieving Data

A large part of the DB-API specification is devoted to retrieving data from a database. Queries are sent via the cursor's `execute()` method and results are obtained via one of cursor's many `fetch...()` methods. This section details the different `fetch...()` methods and provides examples and instructions for each.

Using `fetchall()`

The `fetchall()` function obtains all rows in your result set (or all remaining results if you've already retrieved some with another `fetch...()` function). It returns a list of sequences (such as tuples or lists). Each sequence in the list corresponds to one record, with the columns being represented as items in the sequence.

It's easy to use `fetchall()` in many situations. However, you should take care to make sure that your result set is a manageable size. When you use `fetchall()`, the entire contents of your result are loaded into memory, so if you're retrieving a huge set of data, you shouldn't use `fetchall()`. Here's an example of `fetchall()`:

```
#!/usr/bin/env python
# fetchall() - Chapter 14
# fetchall.py
# Adjust the connect() call below for your database.

import psycopg
dbh = psycopg.connect('dbname=jgoerzen user=jgoerzen')
print "Connection successful."

cur = dbh.cursor()
cur.execute("SELECT * FROM ch14")

rows = cur.fetchall()
for row in rows:
    print row

dbh.close()
```

In this example, you simply request all the data from the `ch14` table, and print out each row. The results look like this:

```
$ ./fetchall.py
Connecting to dbname=jgoerzen user=jgoerzen
Connection successful.
(0, 'Zero')
(1, 'Item One')
(2, 'Item Two')
(3, 'Three')
```

Using fetchmany()

The `fetchall()` command may not be appropriate sometimes, such as in situations dealing with very large result sets. But if you still want to retrieve data in chunks, the `fetchmany()` function may be perfect. Its return value is the same as `fetchall()`, but it limits itself to only returning a certain amount of data each time it's called. Therefore, you'll need to continue calling it until the result set is exhausted. When there's no more data to return, `fetchmany()` returns an empty sequence.

You can configure how many results are returned at a time by setting the `arraysize` attribute of the cursor beforehand, or you can override that by passing a specific size to `fetchmany()`. Here's an example of using `fetchmany()`:

```
=!/usr/bin/env python
# fetchmany() - Chapter 14 - fetchmany.py
# Adjust the connect() call below for your database.

import psycopg
dbh = psycopg.connect('dbname=jgoerzen user=jgoerzen')
print "Connection successful.

cur = dbh.cursor()
cur.execute("SELECT * FROM ch14")
cur.arraysize = 2

while 1:
    rows = cur.fetchmany()
    print "Obtained %d results from fetchmany()." % len(rows)
    if not len(rows):
        break

    for row in rows:
        print row

dbh.close()
```

This code is similar to the `fetchall()` example, except that there's simply another loop involved. I set the `arraysize` far lower than you're likely to see in real systems in order to illustrate its effect on `fetchmany()`. Here's the output from this example:

```
$ ./fetchmany.py
Connection successful.
Obtained 2 results from fetchmany().
(0, 'Zero')
(1, 'Item One')
Obtained 2 results from fetchmany().
(2, 'Item Two')
(3, 'Three')
Obtained 0 results from fetchmany().
```

Using `fetchone()`

The final data retrieval function is `fetchone()`. This function will return a single row. If there's no more data available, it returns `None`. Since the `fetchone()` function returns only a single row at a time, there's no memory problem to be worried about.

On the other hand, with some databases, this function is slower than using `fetchmany()` or `fetchall()` on large data sets. In many cases, however, that performance difference is negligible. Here's an example of `fetchone()`:

```
#!/usr/bin/env python
# fetchone() - Chapter 14 - fetchone.py
# Adjust the connect() call below for your database.

import psycopg
dbh = psycopg.connect('dbname=jgoerzen user=jgoerzen')
print "Connection successful."

cur = dbh.cursor()
cur.execute("SELECT * FROM ch14")
cur.arraysize = 2

while 1:
    row = cur.fetchone()
    if row is None:
        break
    print row

dbh.close()
```

When you run this program, you get this result:

```
$ ./fetchone.py
Connection successful.
(0, 'Zero')
(1, 'Item One')
(2, 'Item Two')
(3, 'Three')
```

Reading Metadata

In addition to returning the data requested by a query, database servers also return metadata. This metadata contains information such as the name and type of the data in each result column.

With DB-API, most of the metadata is accessible by using the `description` variable of your cursor object after executing a query. Here's an example that shows that metadata is available:

```
=!/usr/bin/env python
# metadata example - Chapter 14 - description.py
# Adjust the connect() call below for your database.

import psycopg
dbh = psycopg.connect('dbname=jgoerzen user=jgoerzen')
print "Connection successful.

cur = dbh.cursor()
cur.execute("SELECT * FROM ch14")

for column in cur.description:
    name, type_code, display_size, internal_size, precision, scale, null_ok =
        column

    print "Column name:", name
    print "Type code:", type_code
    print "Display size:", display_size
    print "Internal size:", internal_size
    print "Precision:", precision
    print "Scale:", scale
    print "Null OK:", null_ok
    print

dbh.close()
```

When run, the program produces the following output:

```
$ ./description.py
Connection successful.
Column name: mynum
Type code: 23
Display size: 1
Internal size: 4
Precision: None
Scale: None
Null OK: None

Column name: mystring
Type code: 1043
Display size: 8
Internal size: 30
Precision: None
Scale: None
Null OK: None
```

The piece of data that most people will be interested in is the column name. This can be useful when you don't know the names of columns that are being returned; for instance, because you're running queries that are supplied by the user, you might not know what columns are coming back. While column names can be useful, you should be careful. Consider, for instance, the query `SELECT SUM(x), SUM(y) FROM foo`. Some database servers would return two columns named `sum`; others might return `SUM(X)` and `SUM(Y)` columns. Despite this problem, many people find the column name useful.

According to DB-API, databases are required to provide at least the column name and type code. However, some don't provide a type code that's useful; therefore, the column name is the only one that can be relied upon. If a database doesn't provide a given piece of information, that particular item will be set to `None`.

The meaning of all fields except the column name is database dependent. That is, the meaning of the fields depends on the database you're using, and in some cases, on the specific data type.

Counting Rows

Sometimes, it's useful to know how large a result set you have to deal with. For instance, you might present a progress bar widget in a graphical interface while

you download a particularly large set of results. Or you might want to check to see if you have enough disk space to proceed, depending on what you're doing.

With the techniques presented so far in this chapter, there's no way to find out how many rows are available until you read them all in and count them. However, there's a special cursor variable called `rowcount`. The `rowcount` attribute holds the total number of rows retrieved from the last command, regardless of how many rows you have actually read in.

If you're running on a database that doesn't provide this information, the value of `rowcount` will be `-1`. Also, some databases don't provide the row count information until after the first call to a `fetch...()` function. In those cases, you would want to fetch a single row first, then the `rowcount`, then process your result set. Here's a sample use of `rowcount`:

```
#!/usr/bin/env python
# rowcount example - Chapter 14 - rowcount.py
# Adjust the connect() call below for your database.

import psycopg
dbh = psycopg.connect('dbname=jgoerzen user=jgoerzen')
print "Connection successful."

cur = dbh.cursor()
cur.execute("SELECT * FROM ch14")
cur.fetchone()

print "Obtained %d rows" % cur.rowcount

dbh.close()
```

When run, this program will connect to the database and display the number of rows in the `ch14` table, or `-1` if your database doesn't provide that information.

Retrieving Data as Dictionaries

Provided that you're careful, you may use the column name information from the metadata to construct dictionaries out of the rows returned. It can be cumbersome to pick out specific columns by number in wide tables, and names are easier to remember. Here's a program that demonstrates that concept:

```

#!/usr/bin/env python
# fetchone() with dictionary - Chapter 14
# Adjust the connect() call below for your database.

import psycopg

def dictfetchone(cur):
    seq = cur.fetchone()
    if seq == None:
        return seq
    result = {}
    colnum = 0
    for column in cur.description:
        result[column[0]] = seq[colnum]
        colnum += 1
    return result

dbh = psycopg.connect('dbname=jgoerzen user=jgoerzen')
print "Connection successful."

cur = dbh.cursor()
cur.execute("SELECT * FROM ch14")
while 1:
    row = dictfetchone(cur)
    if row == None:
        break
    print row
dbh.close()

```

When run, this program produces the following output:

```

$ ./fetchonedict.py
Connection successful.
{'mystring': 'Zero', 'mynum': 0}
{'mystring': 'Item One', 'mynum': 1}
{'mystring': 'Item Two', 'mynum': 2}
{'mystring': 'Three', 'mynum': 3}

```

NOTE Some database modules, such as `psycopg`, already provide `dictfetchone()` and `dictfetchall()` methods. You could use those, but your code won't be portable to database modules that don't provide these optional functions.

Using Data Types

Both SQL and Python have type systems. For instance, SQL defines INTEGER and VARCHAR whereas Python defines int and str. Many simple types map directly between the two systems; INTEGER to int, for instance. However, some more complex types don't map so easily. What's worse, different databases may represent more complex types such as dates in different ways.

DB-API does present a solution for this problem. It defines seven different functions (or class constructors, depending on the database) that convert Python data into a representation suitable as a parameter to SQL statements like SELECT or INSERT. These seven functions are

- `Binary()`. Takes a string and generates a binary database object. The database object is one specifically designed to hold large amounts of binary data. Depending on your server, these may be called things such as blobs, memo fields, or other names.
- `Date()`. Takes an integer year, month, and day of the month and generates a date object. Two-digit year abbreviations shouldn't be used.
- `DateFromTicks()`. Takes an integer or float representing the number of seconds since the UNIX epoch and generates a database date object from it. The argument is in the format of `time.time()`.
- `Time`. Takes an hour (in 24-hour format), minute, and second, all integers, and generates a time object.
- `TimeFromTicks()`. Takes an integer or float representing the number of seconds since the UNIX epoch and generates a database time object from it. The argument is in the format of `time.time()`.
- `Timestamp`. Takes a year (two-digit abbreviations aren't allowed), month, day of month, hour (24-hour format), minute, and second. It generates a timestamp object for the database.
- `TimestampFromTicks()`. Takes an integer or float representing the number of seconds since the UNIX epoch and generates a database timestamp object from it. The argument is in the format of `time.time()`.

Here's a program that demonstrates the use of several of these functions:

```

#!/usr/bin/env python
# Using types to insert data - Chapter 14 - typeinsert.py
# Adjust the connect() call below for your database.

import psycopg, time

dsn = 'dbname=jgoerzen user=jgoerzen'
print "Connecting to %s" % dsn
dbh = psycopg.connect(dsn)
print "Connection successful."

cur = dbh.cursor()
cur.execute("""CREATE TABLE ch14types (
    mydate    DATE,
    mytimestamp TIMESTAMP,
    mytime   TIME,
    mystring varchar(30))""")
query = """INSERT INTO ch14types VALUES (
    %(mydate)s, %(mytimestamp)s, %(mytime)s, %(mystring)s)"""
rows = ( \
    {'mydate': psycopg.Date(2000, 12, 25),
     'mytimestamp': psycopg.Timestamp(2000, 12, 15, 06, 30, 00),
     'mytime': psycopg.Time(6, 30, 00),
     'mystring': 'Christmas - Wake Up!'},
    {'mydate': psycopg.DateFromTicks(time.time()),
     'mytime': psycopg.TimeFromTicks(time.time()),
     'mytimestamp': psycopg.TimestampFromTicks(time.time()),
     'mystring': None})
cur.executemany(query, rows)
dbh.commit()
dbh.close()

```

You can see several different functions being used. Note also the use of the Python special object `None`. When you set a column to `None` in Python, it becomes `NULL` in SQL.

This example creates a new table, `ch14types`, and inserts two rows in it. The first row contains various items representing a December 25, 2000 date. The second row derives most data from the current system time.

Retrieving Typed Data

When you retrieve data back from the database, objects will be created to store that data in Python. Objects similar to the ones created earlier will be created at data retrieval time. If you modify the earlier `fetchone()` example to use the table `ch14types` instead of `ch14`, and then run it, you'll see results like this:

```
$ ./fetchone.py
Connection successful.
(<DateTime object for '2000-12-25 00:00:00.00' at 4021c640>,
 <DateTime object for '2000-12-15 06:30:00.00' at 4021c918>,
 <DateTime object for '1970-01-01 06:30:00.00' at 4021c950>,
 'Christmas - Wake Up!')
(<DateTime object for '2004-01-19 00:00:00.00' at 4021c988>,
 <DateTime object for '2004-01-19 20:08:36.00' at 4021c9c0>,
 <DateTime object for '1970-01-01 20:08:36.00' at 4021c9f8>,
 None)
```

Although it conveys the idea, this output isn't the most useful. With some databases, calling `str()` on the object will convert it into a format more useful. However, that format will be database specific.

Summary

Database modules let you interface with SQL database servers from Python. The DB-API 2.0 specification defines a universal interface. Authors of database interface modules are expected to adhere to this specification. If they do, and your code also follows the specification, your programs are mostly portable from one database server to the next.

Before issuing any commands, you must connect to the database server. The process of connecting is the largest thing that varies from one database server to the next. This chapter presented examples for MySQL, PostgreSQL, and zxJDBC. For more details, consult your database module's documentation. Once connected, you're provided a database handle object through which all commands are sent.

The `execute()` method is used to issue simple commands—ones that don't return any meaningful output. It's typically used to insert data into the database. If you're inserting large amounts of data, you should use `executemany()` or `execute()` with format strings.

Most SQL servers support transactions, thereby allowing you to commit data to the database only after all database modifications composing a logical transaction are complete. The `commit()` and `rollback()` calls commit changes to the database, or forget about them, respectively.

There are several methods to retrieve data from the database. The `fetchone()` method retrieves a single row, `fetchall()` retrieves all resulting rows, and `fetchmany()` retrieves results in chunks. These all operate on database cursors.

Metadata is also available. The `description` variable in a cursor contains information about the columns returned. The `rowcount` variable contains a count of rows in the result set if the database server provides that information.

CHAPTER 15

SSL

AS THE USE of both local networks and the Internet has risen over the years, so too has the importance of these networks to businesses and individuals. Today, people routinely use credit card numbers to make purchases online. Businesses communicate with suppliers over the Internet. Corporate networks provide access to sensitive information.

Unfortunately, along with this rising importance, attempts to subvert security measures and cause trouble have also risen. The consequences of security breaches can be devastating. Therefore, security is of great importance to many people. Securing network services is a battle that must be fought on multiple fronts. It's important to make sure that your systems and network equipment are physically secure by making sure they're locked and monitored to the greatest extent possible. Then, attention must be paid to developing security software. Buggy software, or programs designed without constantly considering security, can also be a threat. Another important concern revolves around transmitting sensitive data across an insecure network, and that is the focus of this chapter. Secure Sockets Layer (SSL), also known as Transport Layer Security (TLS), is designed to help make network communications more secure by taking advantage of modern encryption and cryptographic authentication techniques. Newer versions of SSL are technically known as TLS, but the terms are commonly used interchangeably.

In this chapter, you'll first learn about some of the most common attacks on network systems and how easy it can be for attackers to use them. Then, you'll learn ways that SSL can help defeat these attacks and how to use two different SSL systems for Python to deploy these systems in your programs. Finally, you'll see how to obtain information about the SSL connection itself.

But before I begin, some caution is in order. No security system should be considered foolproof. Techniques described in this chapter—or anywhere else, for that matter—are hints that are designed to help you, but aren't guaranteed to prevent any type of attack. When working with security, a healthy sense of paranoia is a must. New vulnerabilities may also be discovered that necessitate new ways of securing networks. Please also keep in mind that the techniques described in this chapter address only part of the larger security landscape.

Understanding Network Vulnerabilities

Before learning how to secure your programs, it's important to understand the common vulnerabilities that lead to the exploitation of program functionality and data alike. Many of these attacks have something in common: Someone who isn't necessarily physically present at either end of the communication can carry them out.

As traffic travels across the Internet, it may pass through a dozen or more different networks. Each network is controlled by a different company, and isn't necessarily trustworthy. Networks that are physically near a client or a server, such as the LAN at a workplace or university, may also be vulnerable to attacks by disgruntled staff or students. Certain broadband services, especially cable modems and wireless Internet access, have been notorious for exposing network traffic to anyone.

Even if all the networks are trusted, outside attackers can sometimes find ways to breach security and divert or intercept connections. Here are *some* of the ways that outside (or inside) attackers can cause security problems. Many of the attacks described in the following sections are collectively known as man-in-the-middle (MITM) attacks, since the attacker would normally reside on a network somewhere between your computer and the remote one.

Sniffing

Sniffing occurs when an outsider is able to read network traffic as it travels to its destination. A sniffer doesn't modify the traffic or interrupt it; you'll generally have no idea that any sort of security breach is occurring.

Traffic sniffing allows an attacker to carry out illicit activities such as stealing passwords and credit card numbers, reading e-mail, and essentially watching every network-oriented task you carry out. Once these resources are obtained, they can be exploited to cause more damage. For instance, someone in possession of a stolen password could use it to access systems and order merchandise on someone else's account, or use it to read mail or attempt to gather other passwords.

Even if you take precautions and never use your credit card number or send a password unless a site is secure, sniffers can still gather information. For instance, if they monitor your web-browsing traffic, they could discover what college you attended (if you visit their web page frequently), what your interests are, and what online shopping sites you frequent. From that, they may attempt to breach security (perhaps trying to guess your password at the shopping site).

Sniffing is probably the most common type of attack.

Insertion Attacks

An insertion attack occurs when someone adds unwanted data to a network stream. Consider, for instance, an application such as FTP that has access to files on a remote system. Once you've logged on, an attacker with control of some network between you and the server could insert a command to delete files from the server, even if the attacker has no knowledge of your password.

Deletion Attacks

A deletion attack is similar to the insertion attack, except this one removes information from the network stream rather than adding. For instance, if you sent a command `rm *.txt` to a UNIX machine, intending to delete all text files, someone attempting a deletion attack may be able to delete the `.txt` part, leaving just `rm *`—a command that deletes all files, not just the text ones. Deletion attacks aren't frequently seen, but remain a matter of concern.

Replay Attacks

A replay attack occurs when someone has been sniffing network traffic and later replays it. This is one reason why a simple password-based encryption scheme isn't a good idea. Consider, for instance, what happens if you log on to an FTP server and issue a command to delete all files. A sniffer may be able to capture all that traffic. But perhaps the sniffer is unable to get passwords from that capture—maybe the communication was encrypted.

However, if that person wishes to cause you harm, he might wait a few days and then connect back to the server. Even though your password isn't known, your encrypted traffic could be sent back, and the command to delete your files could be issued—again deleting files from the FTP server.

Session Hijacking

Session hijacking occurs when you've opened up a connection to a remote service, but while that connection is still open, an attacker is able to impersonate you while communicating with the remote server. This is similar to an insertion or deletion attack. With session hijacking, the attacker will completely take over your session, effectively impersonating you. The attacker will then be able to run any command you could, and see any data you'd be able to see.

Fake Server (Traffic Redirection)

An attacker could sometimes set up a server that mimics a real one. For instance, if you attempt to visit your bank's website, an attacker may set up a fake website that looks like it's your bank's site. Then the attacker may be able to redirect your network traffic away from the real website to the fake one. You may have no idea that you're not communicating with the real bank until you've already supplied your password. Then the attacker can use the stolen password to access the real bank's site.

Compromised Server

Sometimes attackers can subvert program or operating system security and essentially take over a server. Once they have administrator access to a server, all bets are off. They can sniff traffic, access data streams even after they've been decrypted, and generally cause all sorts of havoc. There's little that can be done to secure a program against a compromised server; SSL can be subverted in this way just like virtually any other security mechanism.

This is why, for a secure system, it's vital to not only write secure applications but make sure that the operating environment is also secure.

Human Engineering

Many programmers make a critical mistake when designing secure systems: forgetting the users of the systems. Attackers have been known to use "human engineering"—fooling people into revealing secure information—to gain access to resources. For instance, an attacker trying to penetrate the security on a corporate network might call an employee, claim to work for the Information Services department, and ask for the employee's password.

More mundane things can cause security breaches as well. Reading over somebody's shoulder as a password is typed or writing a password on a Post-it note stuck to a computer monitor are both ways of revealing sensitive information to potential attackers.

Again, SSL is powerless in the face of this sort of attack.

Reducing Vulnerabilities with SSL

SSL is designed to reduce or eliminate many of the vulnerabilities mentioned previously. Here are some of the ways that it does so:

- Authentication of remote servers (and optionally clients) to help thwart fake server attacks.
- Encryption of data streams in both directions to help thwart sniffing.
- Changing encryption keys so that the same data is represented differently at different times. This helps thwart replay attacks.
- Data-integrity checking to help thwart hijacking, insertion, and deletion attacks.

The SSL libraries handle most of these checks automatically for you. However, the problem of authenticating the remote machine requires some help in your application.

Authenticating the Remote

SSL uses public-key cryptography. For encryption, each endpoint (with TCP/IP, a network connection has two endpoints: a client and a server) has a key *pair*: a public and a private key. The private key is known only to the machine it belongs to; the public key can be known everywhere.

Encrypted messages sent to the machine are encrypted with a public key and decrypted with a private key. Likewise, digital signatures are made with the private key and can be verified with the public key.

With traditional cryptography, the password (or other secure token) must be known in advance. That is, there must already be a secure communication channel that can be used to convey the key. That isn't the case with public-key cryptography.

You're still left with the problem, however, of making sure that the public key the server gives out is really the authentic public key for that server, and not the public key for a server that's intercepting and redirecting the traffic. Shipping public keys that are known to be good for millions of hosts with each SSL application isn't practical.

On the Web, this problem is commonly solved through the use of a list of Certificate Authorities (CA). A CA is an organization, such as Thawte or RSA, that provides a digital verification that the public key being presented really is held by the organization claiming it. Thus, all a web browser needs is a list of trusted CAs, and from there, any remote SSL site can have its authenticity validated.

Without authentication of the remote, SSL is still better than nothing, since it can help prevent certain attacks such as sniffing. However, it will be defenseless against a redirection or fake server attack.

Understanding SSL in Python

Two separate Python modules offer developers the ability to incorporate SSL into their applications. First, the built-in `socket` module provides SSL support on many platforms. This support is a compile-time option, however, and may be disabled in your installation. The implementation in `socket` is very basic and isn't capable of authenticating the remote client or server.

The second SSL option is available with `pyOpenSSL`, a third-party add-on module. `pyOpenSSL` is an interface to the popular OpenSSL library, and brings with it both more power and more complexity. This chapter covers both options.

Using Built-in SSL

Python's built-in SSL support is present on most systems. However, it's possible to compile Python without SSL support. If you get error messages complaining of a lack of SSL support, you may need to recompile your version of Python or obtain a newer build.

To start up an SSL session, you first connect a socket like usual, then create an SSL object that communicates over the socket. From that point on, all communication will occur using the new SSL object. Here's a simple example:

```
#!/usr/bin/env python
# Basic SSL example - Chapter 15 - basic.py

import socket, sys

def sendall(s, buf):
    byteswritten = 0
    while byteswritten < len(buf):
        byteswritten += s.write(buf[byteswritten:])

    print "Creating socket...",
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    print "done."

    print "Connecting to remote host...",
    s.connect(("www.openssl.org", 443))
    print "done."

    print "Establishing SSL...",
    ssl = socket.ssl(s)
    print "done."
```

```

print "Requesting document...",
sendall(ssl, "GET / HTTP/1.0\r\n\r\n")
print "done."

s.shutdown(1)

while 1:
    try:
        buf = ssl.read(1024)
    except socket.sslerror, err:
        if (err[0]) in [socket.SSL_ERROR_ZERO_RETURN, socket.SSL_ERROR_EOF]:
            break
        elif (err[0]) in [socket.SSL_ERROR_WANT_READ,
                          socket.SSL_ERROR_WANT_WRITE]:
            continue
        raise
    if len(buf) == 0:
        break
    sys.stdout.write(buf)

s.close()

```

When you run this program (it needs no arguments), you'll notice it connecting to the www.openssl.org website. Then it establishes an SSL connection, and communicates like normal with HTTP. It will print out the homepage for that site.

You'll notice the `sendall()` function in the program. SSL objects provide only two methods: `read()` and `write()`. They correspond roughly to the `recv()` and `send()` methods of sockets. Like `send()`, the `write()` method doesn't guarantee that it will actually write out all the requested data. Unfortunately, the SSL objects don't provide an equivalent for the standard socket `sendall()` method described in Chapter 1, so that must be implemented in your program itself. This version of `sendall()` simply ensures that the entire string gets transmitted, just like the standard `sendall()` method.

Notice the exception handling surrounding the call to `read()`. Python's built-in SSL support can raise exceptions on end-of-file or even while reading. This code makes sure to exit the loop when an appropriate end-of-file is received, and to just ignore the other exceptions that don't signify an error.

Many network protocols these days are line-oriented. SSL objects don't provide a `readline()` method, which makes working with line-oriented protocols difficult. Here's an SSL wrapper object that adds some missing functions:

```

#!/usr/bin/env python
# Basic SSL example with wrapper - Chapter 15 - basic-wrap.py

import socket, sys

class sslwrapper:
    def __init__(self, sslsock):
        self.sslsock = sslsock
        self.readbuf = ''
        self.eof = 0

    def write(self, buf):
        byteswritten = 0
        while byteswritten < len(buf):
            byteswritten += self.sslsock.write(buf[byteswritten:])

    def _read(self, n):
        retval = ''
        while not self.eof:
            try:
                retval = self.sslsock.read(n)
            except socket.sslerror, err:
                if (err[0]) in [socket.SSL_ERROR_ZERO_RETURN,
                               socket.SSL_ERROR_EOF]:
                    self.eof = 1
                elif (err[0]) in [socket.SSL_ERROR_WANT_READ,
                                 socket.SSL_ERROR_WANT_WRITE]:
                    continue
                else:
                    raise
            break

        if len(retval) == 0:
            self.eof = 1
        return retval

    def read(self, n):
        if len(self.readbuf):
            # Return the stuff in readbuf, even if less than n.
            # It might contain the rest of the line, and if we try to
            # read more, it might block waiting for data that is not
            # coming to arrive.
            bytesfrombuf = min(n, len(self.readbuf))
            retval = self.readbuf[:bytesfrombuf]
            self.readbuf = self.readbuf[bytesfrombuf:]
        else:
            retval = ''
        return retval

```

```
    self.readbuf = self.readbuf[bytesfrombuf:]
    return retval
    retval = self._read(n)
    if len(retval) > n:
        self.readbuf = retval[n:]
        return retval[:n]
    return retval

def readline(self, newlinestring = "\n"):
    retval = ''
    while 1:
        linebuf = self.read(1024)
        if not len(linebuf):
            return retval
        nlindex = linebuf.find(newlinestring)
        if nlindex != -1:
            retval += linebuf[:nlindex + len(newlinestring)]
            self.readbuf = linebuf[nlindex + len(newlinestring):] \
                + self.readbuf
            return retval
        else:
            retval += linebuf

print "Creating socket...",
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
print "done."

print "Connecting to remote host...",
s.connect(("www.openssl.org", 443))
print "done."

print "Establishing SSL...",
ssl = socket.ssl(s)
print "done.

ssl = sslwrapper(ssl)

print "Requesting document...",
ssl.write("HEAD / HTTP/1.0\r\n\r\n")
print "done.

s.shutdown(1)
```

```

while 1:
    line = ssl.readline("\r\n")
    if not len(line):
        break
    print "Received line:", line.strip()

s.close()

```

While this program simply reads a few lines from the server, you can take the `sslwrapper` class and use it with your own programs. The `sslwrapper` class supports enough to be a drop-in replacement for a standard socket object in many programs. Also note that you may not need to use it at all; some Python modules, such as `urllib2` discussed in Chapter 6, already support Python's built-in SSL.

Using OpenSSL

In addition to the built-in SSL support, there's also a binding for OpenSSL available for Python called `pyOpenSSL`. Using `pyOpenSSL` is similar to the build-in SSL capabilities in the sense that it, too, creates a wrapper around the socket. However, `pyOpenSSL`'s wrapper is more powerful and full of features than the default one, and notably will not require the sort of add-on glue that you saw in `basic-wrap.py` for `socket.ssl`.

Before you can use OpenSSL in your program, you'll need to obtain the `pyOpenSSL` distribution. If your operating system doesn't provide it, you may download it from <http://pyopenssl.sourceforge.net/>. Windows users may download a prebuilt version from <http://twistedmatrix.com/products/download>. Install that before running programs in this section. These examples should work with version 0.5.1 and above.

Here's a basic example for use with OpenSSL:

```

#!/usr/bin/env python
# Basic OpenSSL example - Chapter 15 - osslbasic.py

import socket, sys
from OpenSSL import SSL

# Create SSL context object
ctx = SSL.Context(SSL.SSLv23_METHOD)

```

```
print "Creating socket...",  
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)  
print "done."  
  
# Create SSL connection object  
ssl = SSL.Connection(ctx, s)  
  
print "Establishing SSL...",  
ssl.connect(('www.openssl.org', 443))  
print "done."  
  
print "Requesting document...",  
ssl.sendall("GET / HTTP/1.0\r\n\r\n")  
print "done."  
  
while 1:  
    try:  
        buf = ssl.recv(4096)  
    except SSL.ZeroReturnError:  
        break  
    sys.stdout.write(buf)  
  
ssl.close()
```

If you run this example (you can just use `./osslbasic.py`), you'll see it connect to `www.openssl.org` using SSL, and dump that site's homepage. To do that, it first creates a `Context` object by calling `SSL.Context`. Next, a socket is created as usual. After that, an `SSL Connection` object is created. From this point on, all operations will take place using this `Connection` object; the `socket` object is no longer needed. A connection is opened, and communication proceeds as normal—just as it would with a standard socket. In fact, you could pass the `Connection` object to just about any function that expects a `socket` object. Once the connection is established that existing code should be able to work with this object with only a small modification to the reading code.

Verifying Server Certificates with OpenSSL

The previous example connected to an SSL server, but it didn't verify the authenticity of that server. In this section, you'll learn how to do server verification in the same manner that web browsers do.

Obtaining Root Certificate Authority Certificates

The first thing you need to do is obtain the certificates for the root (or master) certificate authorities. These organizations are recognized for doing a good job signing keys and verifying identities. There's no formal standard for who is allowed to be a CA. If you don't have the certificates already, you can download a set of certificates using the latest tar.gz file from <http://ftp.debian.org/debian/pool/main/c/ca-certificates>. You'll need to unpack the file and run the included Makefile to generate the certificate files. If this doesn't work for you, you may be able to export the certificates from your web browser. Browsers such as Mozilla support exporting certificates.

Next, you'll want to generate one master file with the certificates. That's easy to do; you can simply concatenate all the certificate files together (`cat *.crt > filename` will do the trick on UNIX systems if you put the files in a single directory; on Windows you could use `copy file1.crt+file2.crt+... dest.crt`). You should wind up with one big file with many BEGIN CERTIFICATE and END CERTIFICATE blocks.

If you still have trouble, you can use the `certfiles.crt` file included online with the example files for this book. However, it isn't kept up to date, so you should still seek out one of the other methods if possible.

Verifying the Certificates

Here's an example program that connects to a remote site and verifies its certificate:

```
#!/usr/bin/env python
# OpenSSL example with verification - Chapter 15 - osslverify.py
#
# Command-line arguments -- root CA file, remote host

import socket, sys
from OpenSSL import SSL

# Grab the command-line parameters
cafile, host = sys.argv[1:]
```

```
def printx509(x509):
    """Display an X.509 certificate"""
    fields = {'country_name': 'Country',
              'SP': 'State/Province',
              'L': 'Locality',
              'O': 'Organization',
              'OU': 'Organizational Unit',
              'CN': 'Common Name',
              'email': 'E-Mail'}
```

```
for field, desc in fields.items():
    try:
        print "%30s: %s" % (desc, getattr(x509, field))
    except:
        pass
```

```
# Whether or not the certificate name has been verified
cnverified = 0
```

```
def verify(connection, certificate, errnum, depth, ok):
    """Verify a given certificate"""
    global cnverified

    subject = certificate.get_subject()
    issuer = certificate.get_issuer()

    print "Certificate from:"
    printx509(subject)

    print "\nIssued By:"
    printx509(issuer)

    if not ok:
        # OpenSSL could not verify the digital signature.
        print "Could not verify certificate."
        return 0
```

```
# Digital signature verified. Now make sure it's for the server
# we connected to.
if subject.CN == None or subject.CN.lower() != host.lower():
    print "Connected to %s, but got cert for %s" % \
        (host, subject.CN)
else:
    cnverified = 1

if depth == 0 and not cnverified:
    print "Could not verify server name; failing."
    return 0

print "-" * 70
return 1

ctx = SSL.Context(SSL.SSLv23_METHOD)
ctx.load_verify_locations(cafile)

# Set up the verification. Notice we pass the verify function to
# ctx.set_verify()
ctx.set_verify(SSLVERIFY_PEER | SSLVERIFY_FAIL_IF_NO_PEER_CERT, verify)

print "Creating socket...",
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
print "done."

ssl = SSL.Connection(ctx, s)

print "Establishing SSL...",
ssl.connect((host, 443))
print "done."

print "Requesting document..."
ssl.sendall("GET / HTTP/1.0\r\n\r\n")
print "done."

while 1:
    try:
        buf = ssl.recv(4096)
    except SSL.ZeroReturnError:
        break
    sys.stdout.write(buf)

ssl.close()
```

Let's go through this code. The `printx509()` function simply displays information about a certificate. It uses the different attributes (CN, OU, and so on) as a key into the object. Each time `verify()` is called, it will call `printx509()` twice: once for the subject (the certificate itself) and once for the issuer (the name of the organization that issued the certificate).

OpenSSL handles verification of the cryptographic signatures itself, based on the filename supplied on the command line. However, there's one more detail that it doesn't handle: verifying that the certificate given to you actually was issued to the server the program is connecting to. This is important. Otherwise, an attacker could simply obtain his own valid certificate, redirect traffic to his server, and present the substitute certificate. By convention, the common name (CN) attribute on a certificate must correspond exactly with the hostname used to connect.

But there's a trick—the `verify()` function could be called multiple times, and as long as the common name passes at least once, that's all that's necessary. Therefore, the global variable `cnverified` defaults to false, but is set to true if the common name is finally verified.

The `verify()` function is passed several parameters by OpenSSL, though I don't use them all. The function starts by displaying information about its certificates. Then it checks the `ok` parameter. If `ok` is false that means that OpenSSL didn't manage to verify things on its end. An error message is printed, and false is returned, telling OpenSSL to abort.

Next, the common name is compared as described earlier. If it matches, `cnverified` is set to a true value.

Finally, if depth is zero (meaning that this is the last time `verified` will be called), the `cnverified` status is checked. If `cnverified` has never been set to true, an error message is printed, and a false value is returned. Otherwise, a true value is returned.

After the `verify()` definition, the code defines the SSL context. This code is mostly the same as the previous example, but there are two extra calls. The call to `load_verify_locations()` specifies the name of the file that holds the CA information. The call to `set_verify()` defines what kind of verification OpenSSL is to do, and says that the verification callback function is `verify`. The remainder of the code is the same as the previous example.

Here are some examples of this program in action:

```
$ ./osslverify.py certfiles.crt www.openssl.org
Creating socket... done.
Establishing SSL... done.
Requesting document...
Certificate from:
    Common Name: www.openssl.org
        Locality: None
        Organization: The OpenSSL Project
        Organizational Unit: None

Issued By:
    Common Name: OpenSSL CA
        Locality: None
        Organization: The OpenSSL Project
        Organizational Unit: Certificate Authority

Could not verify certificate.
Traceback (most recent call last):
  File "./osslverify.py", line 74, in ?
    ssl.sendall("GET / HTTP/1.0\r\n\r\n")
SSL.Error: [('SSL routines', 'SSL3_GET_SERVER_CERTIFICATE',
  'certificate verify failed')]
```

In the previous example, I was presented with a certificate. However, that certificate was signed by the OpenSSL's own CA, which isn't recognized in the program's list of root CAs. Notice that this error caused an exception to be raised, which stopped the program before any data was actually exchanged. That's a different behavior from the earlier examples with Python's built-in SSL libraries, which didn't complain at all. A real web browser would generally prompt the user at this point, asking whether to proceed anyway.

Here's an example of a successful verification:

```
$ ./osslverify.py crtfiles.crt www.accountonline.com
Creating socket... done.
Establishing SSL... done.
Requesting document...
Certificate from:
    Common Name: None
        Locality: None
        Organization: VeriSign, Inc.
        Organizational Unit: Class 3 Public Primary Certification Authority
```

Issued By:

Common Name: None
Locality: None
Organization: VeriSign, Inc.

Organizational Unit: Class 3 Public Primary Certification Authority

Connected to www.accountonline.com, but got cert for None

Certificate from:

Common Name: None
Locality: None
Organization: VeriSign Trust Network
Organizational Unit: VeriSign, Inc.

Issued By:

Common Name: None
Locality: None
Organization: VeriSign, Inc.

Organizational Unit: Class 3 Public Primary Certification Authority

Connected to www.accountonline.com, but got cert for None

Certificate from:

Common Name: www.accountonline.com
Locality: Weehawken
Organization: Citigroup
Organizational Unit: WHG-weproxy6

Issued By:

Common Name: None
Locality: None
Organization: VeriSign Trust Network
Organizational Unit: VeriSign, Inc.

done.

HTTP/1.1 200 OK

Server: JavaWebServer/2.0

Content-length: 164

Content-type: text/html

Last-modified: Mon, 05 Nov 2001 14:27:17 GMT

Connection: close

Date: Tue, 27 Jan 2004 01:22:11 GMT

In this case, `verify()` was called three times. The first two times, the certificate in question didn't present a common name. The third time the common name matched, so the verification was successful. You can see the output from the server just past the verification results.

As you work on SSL projects, I suggest you start with the code for `osslverify.py`. With it, you'll be able to make any of your programs SSL-aware. You can also start with any of your existing code and add SSL support to it by copying the connection code and `verify()` from `osslverify.py`. This is all you need to use SSL wherever you like.

Summary

There are many different ways that attackers can breach the security of networks and systems. SSL, the Secure Sockets Layer, is designed to help prevent many different attacks, though like any security technology, it isn't foolproof. SSL provides two basic services: encryption of the communications and authentication of the remote server or client.

Two SSL implementations exist for Python: the built-in SSL support and the `pyOpenSSL` library. The built-in SSL library has the advantage that many Python users and developers already have it, but it contains fewer features than a complete SSL library and requires more work to use.

`pyOpenSSL` is a full interface to the popular OpenSSL library for using SSL encryption. Python programmers can use its `Context` objects almost interchangeably with standard socket objects, but need to adjust network reading code.

`pyOpenSSL` also provides support for verification of peer certificates—a critical part of securing network communications. In `osslverify.py`, you saw the function `verify()`, which handled the authentication of the remote machine.



Part Five

Server-Side Frameworks

SocketServer

In the first 15 chapters of this book, I focused mostly on writing network clients with Python. I'd now like to turn to writing network servers in Python. While protocol-specific server modules are more rare than client modules, there's the generic `SocketServer` framework and a few protocols based upon it.

`SocketServer` is a Python framework for handling requests from clients in a server. Python includes `SocketServer` already. `SocketServer` takes advantage of the object-oriented nature of Python to help you implement a server protocol. To write a program that uses `SocketServer`, you actually define classes that inherit from a `SocketServer` base class. Python also provides classes that implement HTTP in order to help get you started.

`SocketServer` is well suited for server applications that receive one request from a client and send back one reply. Some servers may have more advanced needs than a basic `SocketServer` application; Chapters 20–22 discuss writing those types of servers.

The protocol-specific `SocketServer` implementations that come with Python all relate to HTTP servers in some way. In this chapter, you'll be introduced to the basic HTTP servers, and will then learn how to write a server for your own protocol using `SocketServer`.

Using `BaseHTTPServer`

As I mentioned in the introduction to this chapter, Python ships with some `SocketServer` classes to help you get started more quickly with certain protocols. The `BaseHTTPServer` module provides the basic support you may need to write your own HTTP (web) server. Like the other `SocketServer`-related classes, it defines two classes: a server object and a request handler. For `BaseHTTPServer`, these classes are `HTTPServer` and `BaseHTTPRequestHandler`. Here's an example showing their usage. This simple HTTP server will send out the same page to every client, but it does demonstrate the use of `BaseHTTPServer`:

```

#!/usr/bin/env python
# Basic HTTP Server Example - Chapter 16 - basichttp.py

from BaseHTTPServer import HTTPServer, BaseHTTPRequestHandler

class RequestHandler(BaseHTTPRequestHandler):
    def _writeheaders(self):
        self.send_response(200)
        self.send_header('Content-type', 'text/html')
        self.end_headers()

    def do_HEAD(self):
        self._writeheaders()

    def do_GET(self):
        self._writeheaders()
        self.wfile.write("""<HTML><HEAD><TITLE>Sample Page</TITLE></HEAD>
<BODY>This is a sample HTML page. Every page this server provides
will look like this.</BODY></HTML>""")

serveraddr = ('', 8765)
srvr = HTTPServer(serveraddr, RequestHandler)
srvr.serve_forever()

```

To implement your own HTTP server, you'll subclass `BaseHTTPRequestHandler`. The class in this example doesn't do much; it always returns a successful value and always returns the same document to the client, no matter what the client requested.

The `BaseHTTPRequestHandler` class provides some convenient methods for you, such as the `send_response()`, `send_header()`, and `end_headers()` methods, which are used in this example. You can also use the `rfile` and `wfile` variables to access the data stream directly, as I did here, and send the document back.

The last three lines of code create and start the server. Each time a client connects, the `serve_forever()` method will receive the connection, create `RequestHandler` instance, and have the `RequestHandler` service the request. The `RequestHandler` code that was inherited from `BaseHTTPRequestHandler` will receive and parse the request. It will then call a `do_...()` method, where the name is derived from the HTTP method used. The most common HTTP methods are `GET`, `HEAD`, and `POST`. Thus, the `do_...()` methods are generally the entry point into the code you write.

You can simply run `./basehttp.py` to invoke the server. It will continue running until it is explicitly terminated, such as by a Ctrl-C on the terminal, Ctrl-Break on a Windows console, or by the machine going down. Exceptions in the request

handler will cause the current connection to be closed, but the server will continue handling other requests.

Here's what it looks like from a client that's connecting to this server:

```
$ telnet localhost 8765
Trying 127.0.0.1...
Connected to heinrich.complete.org.
Escape character is '^]'.
HEAD / HTTP/1.0
```

```
HTTP/1.0 200 OK
Server: BaseHTTP/0.3 Python/2.3.3
Date: Sat, 31 Jan 2004 21:55:01 GMT
Content-type: text/html
```

Connection closed by foreign host.

```
$ telnet localhost 8765
Trying 127.0.0.1...
Connected to heinrich.complete.org.
Escape character is '^]'.
GET / HTTP/1.0
```

```
HTTP/1.0 200 OK
Server: BaseHTTP/0.3 Python/2.3.3
Date: Sat, 31 Jan 2004 22:02:08 GMT
Content-type: text/html
```

```
<HTML><HEAD><TITLE>Sample Page</TITLE></HEAD>
<BODY>This is a sample HTML page. Every page this server provides
will look like this.</BODY></HTML>
Connection closed by foreign host.
```

Note that you'll have to press Enter twice after typing the GET request. For a different view, you can actually use a web browser to connect to this example. You can use the URL <http://localhost:8765/> for your browser.

Handling Requests for Specific Documents

Giving a single document to everyone probably isn't terribly useful. Here's a more complete example. It serves up two documents: a static one, and one that's dynamically generated.

```

#!/usr/bin/env python
# Basic HTTP Server Example with Two Documents - Chapter 16
# basichttpdoc.py

from BaseHTTPServer import HTTPServer, BaseHTTPRequestHandler
import time

starttime = time.time()

class RequestHandler(BaseHTTPRequestHandler):
    """Definition of the request handler."""
    def _writeheaders(self, doc):
        """Write the HTTP headers for the document. If there's no
        document, send a 404 error code; otherwise, send a 200 success code."""
        if doc is None:
            self.send_response(404)
        else:
            self.send_response(200)

        # Always serve up HTML for now.
        self.send_header('Content-type', 'text/html')
        self.end_headers()

    def _getdoc(self, filename):
        """Handle a request for a document, returning one of two different
        pages as appropriate."""
        global starttime
        if filename == '/':
            return """<html><head><title>Sample Page</title></head>
<body>This is a sample page. You can also look at the
<a href="stats.html">server statistics</a>.
</body></html>
"""
        elif filename == '/stats.html':
            return """<html><head><title>Statistics</title></head>
<body>This server has been running for %d seconds.
</body></html>
""" % int(time.time() - starttime)
        else:
            return None

```

```

def do_HEAD(self):
    """Handle a request for headers only"""
    doc = self._getdoc(self.path)
    self._writeheaders(doc)

def do_GET(self):
    """Handle a request for headers and body"""
    doc = self._getdoc(self.path)
    self._writeheaders(doc)
    if doc is None:
        self.wfile.write("""<html><head><title>Not Found</title></head>
<body>The requested document '%s' was not found.</body>
</html>
""" % self.path)
    else:
        self.wfile.write(doc)

# Create the object and serve requests
serveraddr = ('', 8765)
srvr = HTTPServer(serveraddr, RequestHandler)
srvr.serve_forever()

```

The `_getdoc()` function looks up the document to return when it's given a filename. Most real web servers would consult a directory on disk, but this one just has two built-in documents that it can serve. If it doesn't find a document that matches the given filename, it returns `None`.

CAUTION If you do want to serve up files from a disk, I suggest using the `SimpleHTTPServer` module discussed later in this chapter. Attempting to serve files from a disk by yourself can lead to security problems, since the correct algorithms to sanitize the request string can be tricky to get right.

The `_writeheaders()` function receives that document. It sends a 404, File Not Found code if the document was `None`, and a 200 (Document OK) code otherwise. The `do_GET()` function is also slightly modified; it generates an appropriate error document if no document was found.

You can point a web browser to `http://localhost:8765/` after starting this server. Try loading the statistics page and hitting your reload (or refresh) button a few times. Notice how the number on that page increments each time.

You can also connect directly to the server with telnet like before. Here's what that will look like:

```
$ telnet localhost 8765
Trying 127.0.0.1...
Connected to heinrich.complete.org.
Escape character is '^].
GET /nonexistent HTTP/1.0

HTTP/1.0 404 Not Found
Server: BaseHTTP/0.3 Python/2.3.3
Date: Sat, 31 Jan 2004 22:29:34 GMT
Content-type: text/html

<html><head><title>Not Found</title></head>
<body>The requested document '/nonexistent' was not found.</body>
</html>
Connection closed by foreign host.
```

Handling Multiple Requests Simultaneously

The previous examples aren't suitable for use in a production server because they only service one client at a time. From the time that a client connects until the time it disconnects, no other clients can be serviced. Even with this small program, that could be a problem. For example, somebody on a low-quality dial-up link may not even get the request sent for 20 seconds. This is a long time to wait, and on a busy server, could stall hundreds or thousands of other clients.

`SocketServer` (and its subclasses) support two different ways of solving this dilemma: forking and threading. These two solutions are discussed in greater detail in Chapters 20 and 21, respectively. Briefly, forking involves starting a new process to handle each incoming connection; all these processes are then completely separate from each other. Threading involves using Python threads to handle connections, and doesn't separate the different connection handlers as much. A third method involving nonblocking (or asynchronous) communication isn't supported by `SocketServer` and is covered in Chapter 22.

If you're looking for a quick way to make your server do multitasking and are running on a UNIX or Linux platform, I recommend forking. Both forking and threading have their complexities, but forking is supported more widely on different UNIX platforms, and this makes it more difficult for connections to interfere with each other. If your code will need to run on Windows, then you must use threading. Most Python implementations for Windows don't implement forking.

Adding forking or threading support to your program is simple. Here's a modified version of the last example. It's fully multitasking and uses threads.

```
#!/usr/bin/env python
# Basic HTTP Server Example with Two Documents, threading version
# Chapter 16
# basichttpdocthread.py

from BaseHTTPServer import HTTPServer, BaseHTTPRequestHandler
from SocketServer import ThreadingMixIn
import time, threading

starttime = time.time()

class RequestHandler(BaseHTTPRequestHandler):
    """Definition of the request handler."""
    def _writeheaders(self, doc):
        """Write the HTTP headers for the document. If there's no
        document, send a 404 error code; otherwise, send a 200 success code."""
        if doc is None:
            self.send_response(404)
        else:
            self.send_response(200)

        # Always serve up HTML for now.
        self.send_header('Content-type', 'text/html')
        self.end_headers()

    def _getdoc(self, filename):
        """Handle a request for a document, returning one of two
        different pages as appropriate."""
        global starttime
        if filename == '/':
            return """<html><head><title>Sample Page</title></head>
<body>This is a sample page. You can also look at the
<a href="stats.html">server statistics</a>.
</body></html>
"""
        elif filename == '/stats.html':
            return """<html><head><title>Statistics</title></head>
<body>This server has been running for %d seconds.
</body></html>
""" % int(time.time() - starttime)
        else:
            return None
```

```

def do_HEAD(self):
    """Handle a request for headers only"""
    doc = self._getdoc(self.path)
    self._writeheaders(doc)

def do_GET(self):
    """Handle a request for headers and body"""
    print "Handling with thread", threading.currentThread().getName()
    doc = self._getdoc(self.path)
    self._writeheaders(doc)
    if doc is None:
        self.wfile.write("""<html><head><title>Not Found</title></head>
<body>The requested document '%s' was not found.</body>
</html>
""" % self.path)
    else:
        self.wfile.write(doc)

class ThreadingHTTPServer(ThreadingMixIn, HTTPServer):
    pass

# Create the object and serve requests
serveraddr = ('', 8765)
srvr = ThreadingHTTPServer(serveraddr, RequestHandler)
srvr.serve_forever()

```

Notice the new `ThreadingHTTPServer` class. This class declares two base classes: `ThreadingMixIn` and the same `HTTPServer` class used before. The `ThreadingMixIn` class contains code that implements the threading. Then you just instantiate the new class below—instant threading support! You can see that in action; I added a `print` statement to `do_GET()`. The statement displays which thread handles a given connection. This principle works for all the different `SocketServer` subclasses discussed in this chapter.

SimpleHTTPServer

The `SimpleHTTPServer` class extends `BaseHTTPServer`. `SimpleHTTPServer` serves up regular files from the current working directory. It also has support for finding `index.html` files and, in newer versions of Python, generating on-the-fly directory listings. As with any server that accesses files from your disk, make sure you set permissions properly, code is written cleanly, and the server is properly configured. Otherwise, you could inadvertently serve up private data. Here's an example of the simplest `SimpleHTTPServer`:

```
#!/usr/bin/env python
# Basic HTTP Server Example - Chapter 16 - simplehttp.py

from BaseHTTPServer import HTTPServer
from SimpleHTTPServer import SimpleHTTPRequestHandler

serveraddr = ('', 8765)
srvr = HTTPServer(serveraddr, SimpleHTTPRequestHandler)
srvr.serve_forever()
```

When run, this program will simply start serving up the files in the current working directory (and its subdirectories). Like previous examples, you can run this program without any command-line parameters. Just start it with `./simplehttp.py`. You can again connect to port 8765 on localhost with either a telnet client or a web browser to see it in action. This program can also use threads:

```
#!/usr/bin/env python
# Basic HTTP Server Example with threading - Chapter 16
# simplehttpthread.py

from BaseHTTPServer import HTTPServer
from SimpleHTTPServer import SimpleHTTPRequestHandler
from SocketServer import ThreadingMixIn

class ThreadingServer(ThreadingMixIn, HTTPServer):
    pass

serveraddr = ('', 8765)
srvr = ThreadingServer(serveraddr, SimpleHTTPRequestHandler)
srvr.serve_forever()
```

CGIHTTPServer

The `CGIHTTPServer` is similar to the `SimpleHTTPServer`, but takes it one step farther. It can execute CGI scripts among the files it serves. By default, it will consider Python scripts executable files that reside in either the `cgi-bin` or `htbin` directories under the server root as CGI scripts. CGI scripts are covered in Chapter 18. Together with the `CGIHTTPServer`, CGI scripts can offer an elegant and simple pure-Python solution to the problem of providing dynamic content. Implementing `CGIHTTPServer` is just as easy as the `SimpleHTTPServer`. Always remember that CGI scripts are full programs, so they could potentially make your server less secure if you run untrusted code. Here's an example CGI server. In this case, forking is used instead of threading;

this is the usual method of invoking CGI scripts, but if you're running on Windows, you'll still want to use threading.

```
#!/usr/bin/env python
# Basic HTTP CGI Server Example with forking -- Chapter 17

from BaseHTTPServer import HTTPServer
from CGIHTTPServer import CGIHTTPRequestHandler
from SocketServer import ForkingMixIn

class ForkingServer(ForkingMixIn, HTTPServer):
    pass

serveraddr = ('', 8765)
srvr = ForkingServer(serveraddr, CGIHTTPRequestHandler)
srvr.serve_forever()
```

You can test this server with one of the CGI scripts from Chapter 18. Put it in your current working directory, mark it executable, and run the server. Now you can connect to your server with the browser and see the script. If you named the script `myscript.cgi`, you can access it with `http://localhost:8765/myscript.cgi`.

Implementing New Protocols

If one of the existing `SocketServer` classes isn't suitable for you, your own protocol can be implemented using the `SocketServer` module. This module is most appropriate for protocols for which clients connect to a server, make one request, receive an answer, and then disconnect. Here's an example of a server that will give the time to a client in several different formats:

```
#!/usr/bin/env python
# Basic SocketServer Example - Chapter 16 - socketserver.py

from SocketServer import ThreadingMixIn, TCPServer, StreamRequestHandler
import time
```

```

class TimeRequestHandler(StreamRequestHandler):
    def handle(self):
        req = self.rfile.readline().strip()
        if req == "asctime":
            result = time.asctime()
        elif req == "seconds":
            result = str(int(time.time()))
        elif req == "rfc822":
            result = time.strftime("%a, %d %b %Y %H:%M:%S +0000",
                                  time.gmtime())
        else:
            result = """Unhandled request. Send a line with one of the
following words:

```

asctime -- for human-readable time
 seconds -- seconds since the Unix Epoch
 rfc822 -- date/time in format used for mail and news posts"""

```

    self.wfile.write(result + "\n")

```

```

class TimeServer(ThreadingMixIn, TCPServer):
    allow_reuse_address = 1

```

```

serveraddr = ('', 8765)
srvr = TimeServer(serveraddr, TimeRequestHandler)
srvr.serve_forever()

```

The `TimeRequestHandler` does most of the work here. Its base class is `StreamRequestHandler`, which does some work initializing things like `rfile` and `wfile`. Those two instance variables are created for you by using `socket.makefile()`. As you may recall from the discussion of `socket.makefile()` in Chapter 2, results from it are file-like objects that can be manipulated in a way that's similar to standard Python file objects. Here the `rfile` object can be read from and the `wfile` object can be written to. The `handle()` method is called when a connection arrives and things are initialized and ready to go. Thus, `handle()` serves as the entry point into your program.

At the end of the example, the server is created in a manner very similar to the other `SocketServer`-based classes already presented in this chapter. Note the addition of `allow_reuse_address` in the `TimeServer` class. The various HTTP server classes set this automatically for you. This is simply the same as setting `SO_REUSEADDR` (see the “Preparing for Connections” section in Chapter 3).

You can test this server easily. Simply telnet to port 8765 and type `asctime`, `seconds`, or `rfc822` and press Enter. You'll get back a message showing the current time in the requested format. If you supply anything else, you'll get back a help message.

Obtaining Information About the Client

The `StreamRequestHandler` (actually, its base class, `SocketServer.BaseRequestHandler`) initializes a few variables in the class that provide information about the client and the environment. The two most useful are `request`, which is the actual socket object; and `client_address`, the address of the client. The address is in standard (IP, port) form; for instance, ('127.0.0.1', 36414). The IPv6 example in the following section illustrates this.

IPv6

`SocketServer` is designed to be compatible with IPv6, though by default, all its sockets are IPv4-only. (IPv6 is discussed in detail in Chapter 5.) Switching to IPv6 is a simple matter of adjusting the `address_family` variable in your server class.

Here's an example:

```
#!/usr/bin/env python
# SocketServer IPv6 Example - Chapter 16 - ipv6.py

from SocketServer import ThreadingMixIn, TCPServer, StreamRequestHandler
import time, socket

class TimeRequestHandler(StreamRequestHandler):
    def handle(self):
        print "Connection from", self.client_address
        req = self.rfile.readline().strip()
        if req == "asctime":
            result = time.asctime()
        elif req == "seconds":
            result = str(int(time.time()))
        elif req == "rfc822":
            result = time.strftime("%a, %d %b %Y %H:%M:%S +0000",
                                  time.gmtime())
```

```

else:
    result = """Unhandled request. Send a line with one of the
following words:

asctime -- for human-readable time
seconds -- seconds since the Unix Epoch
rfc822 -- date/time in format used for mail and news posts"""
    self.wfile.write(result + "\n")

class TimeServer(ThreadingMixIn, TCPServer):
    allow_reuse_address = 1
    address_family = socket.AF_INET6

serveraddr = ('', 8765)
srvr = TimeServer(serveraddr, TimeRequestHandler)
srvr.serve_forever()

```

Note that it isn't possible for a single `SocketServer` server class to support both IPv4 and IPv6. It's possible for a single *program* to support both, but you'll need one of the techniques in Chapters 20–22 to do so.

Here's the sample output from the server console for this program:

```
$ ./ipv6.py
Connection from ('::1', 36417, 0, 0)
```

The `('::1', 36417, 0, 0)` string reflects the IPv6 address of the client that connected to the server. In this case, `::1` is the IPv6 address for `localhost` and indicates that the connection originated on the local machine.

Summary

`SocketServer` is a Python module that helps simplify writing network servers in Python. To implement a server that uses `SocketServer`, you'll generally create a new subclass of one of its built-in classes.

For HTTP, some of these classes already exist. The `BaseHTTPServer` module provides classes that parse the HTTP request and then leaves the rest up to you. `SimpleHTTPServer` provides classes that serve up plain files from disk, and `CGIHTTPServer` provides classes that add the ability to serve CGI scripts.

If you aren't working with HTTP, you could implement your own `SocketServer` protocol. To do that, you would subclass one of the `SocketServer` classes directly.

To be able to service more than one connection at once, `SocketServer` supports forking and threading. Forking is often a good choice if you'll be running primarily on UNIX or Linux platforms, while threading is a good choice for Microsoft platforms.

The next chapter demonstrates more server modules based on `SocketServer` and are used for writing XML-RPC servers.

SimpleXMLRPCServer

XML-RPC IS A COMMON INTERFACE today. In this chapter, you'll learn how to write XML-RPC servers. These servers might be public servers—perhaps giving out recent news headlines, weather information, search tools, or price quoting. They could also be internal servers for communication between programs on your LAN.

Python makes it easy to set up a basic XML-RPC server. Thanks to the `SocketServer` infrastructure discussed in Chapter 16, an XML-RPC server doesn't require much more code to support. Chapter 8 covers XML-RPC basics from a client perspective, and some examples from that chapter will be used to demonstrate XML-RPC servers.

In general terms, a Remote Procedure Call (RPC) server is a program that exposes certain functions to clients in a way that's designed to be mostly transparent. That is, programmers using those functions on a client might not know that the functions were making calls over the network instead of to other functions in the same program. XML-RPC defines an XML-based protocol for communication between the client and server. You can also use *introspection* with most servers, which is a way for clients to discover what methods are available on the server and also details about those methods.

CAUTION As with any form of RPC, the interface can be deceptively simple, and security must always be remembered. Consider, for instance, a function that simply takes a filename from a client and returns the first 20 lines of that file. This might be fine if the client requests `foo.txt`, but if the client requests `.../etc/passwd` instead, it might well be able to obtain details about the passwords for several users on the system. To maintain security, client requests coming in via XML-RPC must be considered untrustworthy until you've proven otherwise.

Because `SimpleXMLRPCServer` changed in Python 2.3, the examples in this chapter assume that you're running Python 2.3 or later.

In this chapter, you'll first learn how to create a basic XML-RPC server and interact with it. You'll see how you can easily expose an XML-RPC interface to existing Python methods and functions, thereby effectively adding a network

layer atop existing code. You'll also learn how to use DocXMLRPCServer to provide online help for your methods, and CGIXMLRPCServer to run your XML-RPC server as a CGI script under an existing web server. Finally, you'll learn the simple trick necessary to support the multicall optimization.

SimpleXMLRPCServer Basics

Creating a simple XML-RPC program is quite easy. You need to first create a Python class that contains the methods you wish to expose, and then register it with the XML-RPC server. Here's an example:

NOTE If you're running the examples in this chapter on Windows, you'll need to replace Forking with Threading everywhere it appears in this chapter.

```
#!/usr/bin/env python
# SimpleXMLRPCServer Basic Example - Chapter 17 - simple.py
# This program requires Python 2.3 or above

from SimpleXMLRPCServer import SimpleXMLRPCServer, SimpleXMLRPCRequestHandler
from SocketServer import ForkingMixIn

class Math:
    def pow(self, x, y):
        """Returns x raised to the yth power; that is, x ** y.

        x and y may be integers or floating-point values."""
        return x ** y

    def hex(self, x):
        """Returns a string holding a hexadecimal representation of
        the integer x."""
        return "%x" % x

class ForkingServer(ForkingMixIn, SimpleXMLRPCServer):
    pass
```

```

serveraddr = ('', 8765)
srvr = ForkingServer(serveraddr, SimpleXMLRPCRequestHandler)
srvr.register_instance(Math())
srvr.register_introspection_functions()
srvr.serve_forever()

```

This program exposes two methods to the public: `pow()` and `hex()`, which are defined in the standard way for a Python class. Here, the methods are reimplemented in the class; later, you'll see how to serve existing functions without putting them in a class. Notice that you could also use this class as a normal class in your program in the normal way.

To run this server, simply use `./simple.py`. You can use Ctrl-C or Ctrl-Break to terminate it. The remaining examples in this chapter work similarly.

You can use the XML-RPC introspection client from Chapter 8 to interact with the server. You'll want to edit the `url` variable on line six to point to `http://localhost:8765/`. Then, you can run it like this:

```
$ ./xmlrpci.py
Gathering available methods...
```

Available Methods:

```

1: hex
2: pow
3: system.listMethods
4: system.methodHelp
5: system.methodSignature
Select one (q to quit): 2
```

Details for pow

Args: ; Returns: s

...

Help: Returns x raised to the yth power; that is, x^y .

x and y may be integers or floating-point values.

Notice the `Args` lines (omitted). Taken as a group, they show the message “signatures not supported.” Normally, signatures indicate what type of argument (integer, float, string, and so on) is accepted, and what type of value is returned. Python isn’t statically typed, so for some functions, there may not be one particular type that’s accepted. For instance, the `pow()` method will work with two different types of argument.

Therefore, for a Python server, signatures aren’t sensible. However, the function’s docstring is presented as help text.

Testing Your Server

You can use the following program to interactively test your server:

```
#!/usr/bin/env python
# XML-RPC Basic Test Client - Chapter 17 - testclient.py

import xmlrpclib, code

url = 'http://localhost:8765/'
s = xmlrpclib.ServerProxy(url)

interp = code.InteractiveConsole({'s': s})
interp.interact("You can now use the object s to interact with the server.")
```

Using this client to talk to the example XML-RPC server produced the following session output:

```
$ ./testclient.py
You can now use the object s to interact with the server.
>>> s.pow(2, 8)
256
>>> s.hex(255)
'ff'
>>> s.system.listMethods()
['hex', 'pow', 'system.listMethods', 'system.methodHelp',
 'system.methodSignature']
>>> print s.system.methodHelp('pow')
>Returns x raised to the yth power; that is, x ^ y.

x and y may be integers or floating-point values.
>>> import sys
>>> sys.exit()
```

Serving Functions

You're not required to use classes when serving functions. Here's an example that adds two functions to the math example: `int()` and `list.sort()`. Even though `int()` isn't technically a function, it behaves similarly to one. Note, however, that this implementation of `list.sort()` is broken.

```
#!/usr/bin/env python
# SimpleXMLRPCServer Example with functions - Chapter 17 - func.py
# This program requires Python 2.3 or above

from SimpleXMLRPCServer import SimpleXMLRPCServer, SimpleXMLRPCRequestHandler
from SocketServer import ForkingMixIn

class Math:
    def pow(self, x, y):
        """Returns x raised to the yth power; that is, x ^ y.

        x and y may be integers or floating-point values."""
        return pow(x, y)

    def hex(self, x):
        """Returns a string holding a hexadecimal representation of
        the integer x."""
        return "%x" % x

    def sortlist(self, l):
        """Sorts the items in l."""
        l = list(l)
        l.sort()
        return l

class ForkingServer(ForkingMixIn, SimpleXMLRPCServer):
    pass

serveraddr = ('', 8765)
srvr = ForkingServer(serveraddr, SimpleXMLRPCRequestHandler)
srvr.register_instance(Math())
srvr.register_introspection_functions()
srvr.register_function(int)
srvr.register_function(list.sort)      # Won't work!
srvr.serve_forever()
```

Here's an example session:

```
$ ./testclient.py
You can now use the object s to interact with the server.
>>> s.system.listMethods()
['hex', 'int', 'pow', 'sort', 'sortlist', 'system.listMethods',
 'system.methodHelp', 'system.methodSignature']
>>> s.int('5314')
5314
>>> s.sort([5, 3, 1, 8])
Traceback (most recent call last):
  File "<console>", line 1, in ?
    File "/usr/lib/python2.3/xmlrpclib.py", line 1029, in __call__
      return self.__send(self.__name, args)
    File "/usr/lib/python2.3/xmlrpclib.py", line 1316, in __request
      verbose=self.__verbose
    File "/usr/lib/python2.3/xmlrpclib.py", line 1080, in request
      return self.__parse_response(h.getfile(), sock)
    File "/usr/lib/python2.3/xmlrpclib.py", line 1219, in __parse_response
      return u.close()
    File "/usr/lib/python2.3/xmlrpclib.py", line 742, in close
      raise Fault(**self.__stack[0])
Fault: <Fault 1: 'exceptions.TypeError:cannot marshal
None unless allow_none is enabled'>
>>> s.sortlist([5, 3, 1, 8])
[1, 3, 5, 8]
```

From the call to `listMethods()`, you can see that the server now supports three new functions. The `int()` function simply calls the system's built-in function. The `sort()` function, however, raised an exception. The reason is that Python's `sort()` method is an in-place sort; that is, it doesn't return anything. The Python XML-RPC client, by default, raises an exception when `None` is returned from the server since this value usually indicates an error condition. If you really want to receive `None`, you can set `allow_none` to a true value when you create your `ServerProxy` instance.

I also added a `sortlist()` method to illustrate the difference. It works as expected. Note `l = list(l)` in `sortlist()`. This makes sure that it's calling `sort()` as expected. Although this won't likely happen in this example, it's possible that a client could pass an object that would have a method named the same as the one being used that does something entirely different. It's good practice to ensure that all objects are of the type you expect in your XML-RPC server.

Exploiting Class Features

Python's `SimpleXMLRPCServer` acts as a proxy for requests. It receives the requests from the network, decodes them, and then calls the appropriate method just like any other Python code would. When the result is received, it's passed back to the client.

This mechanism lets you use any normal Python features in your classes as long as you're cognizant of the fact that arbitrary Python objects cannot be sent via XML-RPC. That is, as long as your arguments and return values are simple data types, lists, or dictionaries, you can use any Python feature under the hood.

Sending Arbitrary Python Objects

Although XML-RPC doesn't have support for sending arbitrary Python objects across the network, it's possible to do so. Python's `pickle` module provides support to convert almost any Python object to or from a version that can be represented as a string. That string could then be transmitted over the network. However, this approach has many potential security issues. On the receiving side, you may not be receiving the kind of objects you expect. Therefore, protocols based on this sort of conversion are usually discouraged.

Here's an example that demonstrates the use of instance variables and inheritance with XML-RPC. Note that there's a problem with the statistics collection in this program; read on for more details.

In the meantime, here's the example:

```
#!/usr/bin/env python
# SimpleXMLRPCServer Example with extra class features -- Chapter 17
# stat.py
# This program requires Python 2.3 or above

from SimpleXMLRPCServer import SimpleXMLRPCServer, SimpleXMLRPCRequestHandler
from SocketServer import ForkingMixIn
import time

class Stats:
    def getstats(self):
        """Returns a dictionary. The keys are names of the functions,
        and the values are the number of times each function was called."""
        return self.callstats
```

```

def getruntime(self):
    return time.time() - self starttime

def failure(self):
    raise RuntimeError, "This function always raises an error."


class Math(Stats):
    def __init__(self):
        self.callstats = {'pow': 0, 'hex': 0}
        self starttime = time.time()

    def pow(self, x, y):
        """Returns x raised to the yth power; that is, x ^ y.

        x and y may be integers or floating-point values."""
        self.callstats['pow'] += 1      # Doesn't do what you expect!
        return pow(x, y)

    def hex(self, x):
        """Returns a string holding a hexadecimal representation of
        the integer x."""
        self.callstats['hex'] += 1      # Doesn't do what you expect!
        return "%x" % x


class ForkingServer(ForkingMixIn, SimpleXMLRPCServer):
    pass

    serveraddr = ('', 8765)
    srvr = ForkingServer(serveraddr, SimpleXMLRPCRequestHandler)
    srvr.register_instance(Math())
    srvr.register_introspection_functions()
    srvr.serve_forever()

```

This program defines some new methods and some new instance variables. Note that it doesn't matter that the `Math` class now inherits from `Stats`; `SimpleXMLRPCServer` sees the methods from `Stats` just fine—just like any other Python code would.

But the idea of keeping track of how many times a method is called won't work right since you're using a forking server. That's because each time a request comes in, a new, separate process is created to handle that request. It increments `callstats`, then promptly terminates after the response is sent. The parent process's

counter never gets incremented itself because the incrementing always occurs in the child process and the child process cannot modify the parent's memory. Once the child process terminates, as it will as soon as it has finished handling the request, any record of the incremented count is lost. Here's an example of interacting with this program:

```
$ ./testclient.py
You can now use the object s to interact with the server.
>>> s.hex(15)
'f'
>>> s.hex(16)
'10'
>>> s.getstats()
{'pow': 0, 'hex': 0}
>>> s.getRuntime()
269.33166790008545
>>> s.failure()
Traceback (most recent call last):
  File "<console>", line 1, in ?
    File "/usr/lib/python2.3/xmlrpclib.py", line 1029, in __call__
      return self.__send(self.__name, args)
    File "/usr/lib/python2.3/xmlrpclib.py", line 1316, in __request
      verbose=self.__verbose
    File "/usr/lib/python2.3/xmlrpclib.py", line 1080, in request
      return self._parse_response(h.getfile(), sock)
    File "/usr/lib/python2.3/xmlrpclib.py", line 1219, in _parse_response
      return u.close()
    File "/usr/lib/python2.3/xmlrpclib.py", line 742, in close
      raise Fault(**self.__stack[0])
Fault: <Fault 1: 'exceptions.RuntimeError:This function always
raises an error.'>
```

Notice how the values returned by `getstats()` indeed weren't incremented. You can fix the problem by using the `ThreadingMixIn` in place of `ForkingMixIn` (there's an example of that in the "Using DocXMLRPCServer" section later in this chapter). Also, take a look at what happened when `failure()` was called. The server detected the exception and passed it back to the client as a failure string. Python's XML-RPC client library happens to detect these, and raises an exception itself. But notice that the exception raised by the client isn't the same one that was raised on the server side. That's because exception objects cannot be passed across XML-RPC, but strings can.

Using DocXMLRPCServer

The Python `DocXMLRPCServer` is a simple class that adds functionality to `SimpleXMLRPCServer`. The extra features enable a standard web browser to access your server. If it does so, it will be given help information about the functions you've defined. `DocXMLRPCServer` is a drop-in replacement for `SimpleXMLRPCServer`. Here's an example that illustrates a `DocXMLRPCServer`. Also, to address the problems with incrementing statistics in the previous example, this example uses threading instead of forking.

```
#!/usr/bin/env python
# DocXMLRPCServer Example - Chapter 17 - doc.py
# This program requires Python 2.3 or above

from DocXMLRPCServer import DocXMLRPCServer, DocXMLRPCRequestHandler
from SocketServer import ThreadingMixIn
import time

class Stats:
    def getstats(self):
        """Returns a dictionary. The keys are names of the functions,
        and the values are the number of times each function was called."""
        return self.callstats

    def getruntime(self):
        """Returns the number of seconds the class has been
        instantiated."""
        return time.time() - self starttime

    def failure(self):
        """Causes a RuntimeError to be raised."""
        raise RuntimeError, "This function always raises an error."

class Math(Stats):
    def __init__(self):
        self.callstats = {'pow': 0, 'hex': 0}
        self.starttime = time.time()

    def pow(self, x, y):
        """Returns x raised to the yth power; that is, x ^ y. """

```

```

x and y may be integers or floating-point values."""
self.callstats['pow'] += 1
return pow(x, y)

def hex(self, x):
    """Returns a string holding a hexadecimal representation of
the integer x."""
    self.callstats['hex'] += 1
    return "%x" % x

class ThreadingServer(ThreadingMixIn, DocXMLRPCServer):
    pass

serveraddr = ('', 8765)
srvr = ThreadingServer(serveraddr, DocXMLRPCRequestHandler)
srvr.set_server_title("Chapter 18 Example Documentation")
srvr.set_server_name("Chapter 18 Doc")
srvr.set_server_documentation("""Welcome to the sample DocXMLRPCServer from
    Chapter 18.""")
srvr.register_instance(Math())
srvr.register_introspection_functions()
srvr.serve_forever()

```

If you point your web browser to `http://localhost:8765/`, you'll see a description of each of the methods exposed via XML-RPC. You can also adjust the title and introduction of that page via the `set_server_...` family of functions. As you read the generated documentation, notice the `system.listMethods()`, `system.methodHelp()`, and `system.methodSignature()` documentation. These functions were supplied when `srvr.register_introspection_functions()` was called. The documentation for them contains generic examples that are supplied by the default implementation.

Using CGIXMLRPCRequestHandler

It's possible to turn a CGI script into an XML-RPC server. This lets you write a script that runs under an existing web server (which need not be written in Python). The Python script will provide an XML-RPC interface to a client.

This is the job of the `CGIXMLRPCRequestHandler`. While this comes as part of the `SimpleXMLRPCServer` module, it isn't really a server. Here's the most recent example, adjusted for use as a CGI script:

```
#!/usr/bin/env python
# CGI Example - Chapter 17 - cgi.py
# This program requires Python 2.3 or above

from SimpleXMLRPCServer import CGIXMLRPCRequestHandler
import time

class Stats:
    def getstats(self):
        """Returns a dictionary. The keys are names of the functions,
        and the values are the number of times each function was called."""
        return self.callstats

    def getruntime(self):
        """Returns the number of seconds the class has been
        instantiated."""
        return time.time() - self starttime

    def failure(self):
        """Causes a RuntimeError to be raised."""
        raise RuntimeError, "This function always raises an error."

class Math(Stats):
    def __init__(self):
        self.callstats = {'pow': 0, 'hex': 0}
        self starttime = time.time()

    def pow(self, x, y):
        """Returns x raised to the yth power; that is, x ^ y.

        x and y may be integers or floating-point values."""
        self.callstats['pow'] += 1
        return pow(x, y)

    def hex(self, x):
        """Returns a string holding a hexadecimal representation of
        the integer x."""
        self.callstats['hex'] += 1
        return "%x" % x

handler = CGIXMLRPCRequestHandler()
handler.register_instance(Math())
handler.register_introspection_functions()
handler.handle_request()
```

Functionally, this code is identical to others. However, you'll notice that it takes no port number or bind address. That's because it's called by the web server, which takes care of those details itself. Also, it calls `handle_request()` instead of `serve_forever()`. That's because each CGI script handles exactly one request. If you install this under your web server's CGI directory and run it, check out the `getruntime()` result. You'll always get a very small number—likely less than one second. That's because the script itself is executed, handles one request, and then terminates. The statistics will therefore likely be useless, since they will be reset for each request. If you need to keep the statistics, you'll have to write them to a file or devise some other means of persistent storage, such as a database (see Chapter 14).

The exact way to test this example will vary from site to site. If you install the `cgi.py` script in your web server's root directory, and configure it to serve it up as a CGI script, you could adjust `testclient.py` to communicate with it by simply changing the `url` variable to `http://localhost:8765/test.py`.

Supporting Multicall Functions

There's one last feature of the Python XML-RPC modules to mention: multicall functions. Multicall functions are an informal addendum to the XML-RPC standard. They are an optimization that allows clients to submit several XML-RPC requests at once. This can improve performance for clients that send several XML-RPC calls to a server.

With the examples in this chapter, adding multicall support to the server is as simple as adding the additional line `srvr.register_multicall_functions()` before the call to `serve_forever()`. Clients that support multicall functions will then automatically be able to use this feature. This will provide an optimization only, and will not alter functionality.

Summary

With Python's `SimpleXMLRPCServer` module, you can write your own XML-RPC server. This server can expose methods of classes or standalone functions. Since it uses `SocketServer`, you can make your server use threading or forking as appropriate.

Like the XML-RPC client, `SimpleXMLRPCServer` converts between Python types and XML-RPC types. However, you can use arbitrarily complex functions or objects on the server side as long as they accept and return only the types supported by XML-RPC.

The DocXMLRPCServer extends the basic SimpleXMLRPCServer functionality, thereby making your server available to standard web browsers as well. People using a standard browser will see documentation on your server.

With CGIXMLRPCServer, you can provide an XML-RPC server that gets called as a CGI script from your web server. The next chapter discusses CGI scripts in greater detail.

CGI

CGI, THE COMMON GATEWAY INTERFACE, is a way to present dynamic content on websites. Originally, websites mostly presented static information; that is, each visitor to a page saw exactly the same page until the authors manually updated it. Then, each visitor would see the same updated page. However, from even the early days of the Web, developers wanted to be able to present more dynamic information to users. CGI is one of the most frequently used mechanisms to accomplish that.

The “common” in CGI stems from two things: it’s server-independent and it’s language-neutral. This means that a CGI script can run under any web server that supports CGI, and that a CGI script may be written in any language. CGI is neither a network protocol nor a library in itself. Rather, it’s a specification for how information is exchanged between the web server and the program that generates data. A program that complies with CGI and gets executed by a web server is typically called a *CGI script*.

CGI is generally tightly intertwined with HTML. This chapter focuses on the Python side of CGI scripting; if you aren’t already familiar with HTML and HTML forms, please consult a HTML reference.

CGI vs. Other Technologies

CGI is a popular choice for generating dynamic web pages and websites. It’s supported by almost every popular web server and programming language. It usually doesn’t require much configuration on the server, and setup is easy.

However, it does have problems, most notably with performance. Performance can be especially bad when interfacing with databases. Other technologies have been developed that provide greater performance if a certain degree of portability is sacrificed. One such technology is known as mod_python and is discussed in Chapter 19. With mod_python, programs must run under the Apache web server and be written in Python.

Setting Up CGI

Unlike many of the other examples in this book, the examples in this chapter aren't designed to be run from the command line. CGI scripts are executed by a web server.

Web servers generally need to be configured to execute CGI scripts. They will often have restrictions on CGI scripts for security reasons. For example, CGI scripts may need to be placed in a particular directory, have a particular file extension, and be marked executable. The process for configuring a server varies from one web server to the next; consult your server documentation for more information.

If you don't already have a web server and want a quick way to run CGI scripts, the Python module `CGIHTTPServer` provides a convenient way to run a server. Chapter 17 offers a simple server, written in Python, that can execute CGI scripts. It requires that scripts exist in a directory named `cgi-bin` and be marked executable.

If you're using the Apache web server, you can enable CGI scripts in a particular directory by using a configuration directive such as `ScriptAlias/webpath/usr/local/cgi`. If you use that, you can access your scripts with a URL such as `http://localhost/webpath/script.cgi` and it will load `/usr/local/cgi/script.cgi`.

Understanding CGI

Suppose that the website `www.example.com` sells mousetraps. On that site, you have a lot of plain HTML files that describe your different types of traps. You'd also like to let customers purchase traps over the Web, probably using the standard shopping cart metaphor.

Each customer's shopping cart is going to be different, so you obviously can't have a plain HTML page for the cart. You also need to collect billing and shipping information—static HTML won't do.

You might add an "Add to Cart" link on each static page. This link will point to a regular URL, just like all the others. But that's where the similarity stops. Instead of sending a plain file to the web browser, the server instead executes the CGI script. The CGI script will be able to access any information passed from the client in a form (such as the number of mousetraps to add to the cart). It will then generate an HTML document and print it to the standard output. The web server makes sure this output gets sent to the client.

Thus, most CGI scripts are very short-lived; the time between their invocation and their termination is often a fraction of a second. They're also called frequently; a CGI script is executed each time the page is displayed. On heavily loaded sites, this could be hundreds of times per minute. This unusual nature, as shown in the following list, has a few implications for Python programmers:

- Initialization times become critically important. Many Python programs have inefficient initialization. That's ordinarily not a problem since an extra second starting up a program that runs for three months isn't a big deal, but with CGI scripts, it is.
- Error handling is different. If a CGI script dies with an exception, the error is typically logged by your web server, but the client will either get a generic error message or an incomplete document rather than a stack trace. However, the error will have no impact on other running instances of the CGI script.
- Interactivity is handled differently. Rather than being able to prompt the user for more information, a CGI script must execute completely with what it's given. If more information is required, it will have to be called again later.

CGI scripts most frequently generate HTML documents, but they can generate data of any type. For instance, a script may generate a graphic containing some sort of custom image in it.

Understanding CGI in Python

Python, unsurprisingly, provides several modules that are useful for CGI scripts. The primary module is named `cgi` and, for the most part, handles the input side of CGI, though it does provide a few useful functions for generating output. Here's a sample CGI script. This script does nothing but display the current time whenever a browser asks for it.

```
#!/usr/bin/env python
# Simple CGI Example - Chapter 18 - simple.cgi

import cgitb
cgitb.enable()

import time
print "Content-type: text/html"
print

print """<HTML>
<HEAD>
<TITLE>Sample CGI Script</TITLE>
</HEAD>
```

```
<BODY>
The present time is %s.
</BODY>
</HTML>"""\ % time.strftime("%I:%M:%S %p %Z")
print
```

Save this example in the appropriate directory for your web server and pull it up from a browser. You'll see a short HTML document displaying the current time. If you hit your Reload or Refresh button, you'll see that time changes. Each time the document is requested, the output is regenerated with the present time at that moment.

Adjusting the Interpreter Path

The first line of the script is used on Linux and UNIX platforms to define which interpreter is used to run it. Some web servers will not work with the `/usr/bin/env python` string, which is normally used to run Python scripts. In those cases, you'll need to replace it with the full path to your Python interpreter. The string might be `/usr/bin/python` in that case.

Let's examine how this script works. This simple CGI script doesn't even require Python's `cgi` module. First, it imports and enables `cgitb`. The `cgitb` module provides CGI traceback support. In most, but not all cases, this will let you send Python stack traces to the web browser or a log file. When debugging your program and writing code, this is great, but you may want to remove the line when the code is deployed; it can divulge some of your code to a visitor should an error occur, thereby potentially unearthing a security hole.

Next, the HTTP headers are generated. CGI scripts must generate at least the `Content-type` header, which tells the browser what type of file it's receiving. Other headers can be useful as well; for instance, one that sets cookies.

After the headers, a blank line must be sent. A single `print` statement accomplishes that. The blank line separates the headers from the document.

Then the document itself is sent. This document is a minimal HTML document, but it's sufficient to illustrate this feature.

Retrieving Environment Information

Part of the CGI specification calls for web servers to load certain environment variables with information about the session. The `cgi` module uses this information as part of its processing. But you'll often encounter situations in which you'll want to access environment variables directly to get information such as URL or path information.

The `cgi` module provides a couple of handy functions that help you see what the environment looks like. Here's one program that uses one; the function that generates HTML code representing the environment as passed in to the CGI script, as shown here:

```
#!/usr/bin/env python
# CGI Environment - Chapter 18 - environ.cgi

import cgitb
cgitb.enable()

import cgi

print "Content-type: text/html"
print

print """<HTML>
<HEAD>
<TITLE>CGI Environment</TITLE>
</HEAD>
<BODY>"""
cgi.print_environ()
print "</BODY></HTML>"
```

On my system, I installed this as `environ.cgi` under the server's `cgi-bin` directory. When I called up `http://localhost:8765/cgi-bin/environ.cgi/foo`, I saw this:

Shell Environment:

```
GATEWAY_INTERFACE
CGI/1.1
```

```
HTTP_ACCEPT
*/*
```

```
HTTP_USER_AGENT
    Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.6)
Gecko/20040506 Firefox/0.8

PATH_INFO
    /foo

PATH_TRANSLATED
    /tmp/htdocs/chapters/19/foo

REMOTE_ADDR
    127.0.0.1

REMOTE_HOST
    localhost

REQUEST_METHOD
    GET

SCRIPT_NAME
    /cgi-bin/environ.cgi

SERVER_NAME
    localhost

SERVER_PORT
    8765

SERVER_PROTOCOL
    HTTP/1.0

SERVER_SOFTWARE
    SimpleHTTP/0.6 Python/2.3.3
```

I'll define some of the more interesting environment variables from this output:

- `REMOTE_ADDR`. Contains the IP address of the remote web client.
- `PATH_INFO`. Contains the component of the URL that follows the CGI script, if any.
- `REMOTE_HOST`. Sometimes holds the hostname of the remote web client, though many web servers either don't set this variable or set it to the same value as `REMOTE_ADDR`.

- SERVER_NAME. Contains the name of the local web server.
- SERVER_PORT. Contains the port of the local web server on which the request was received.

These can all be accessed via `os.environ`. For instance, `os.environ['REMOTE_ADDR']` contains the client's IP address.

Getting Input

Although CGI scripts that generate data without any input from the user may be useful in some circumstances, CGI is most frequently used to help sites become more interactive with users.

There are three primary ways that CGI scripts can get information from the user: via path-like components in the URL, via GET-encoded parameters in the URL, and via forms (which may use either GET or POST). Let's look at each of these methods.

Extra URL Components

In the preceding environment example, I saved the CGI script as `environ.cgi` but called `http://localhost:8765/cgi-bin/environ.cgi/foo` as if `environ.cgi` were a directory. There's no file `foo` on the system, yet the script loaded up properly. Web servers are configured to take everything after the script name and pass it to you in the `PATH_INFO` environment variable.

If you have a URL that specifies a CGI script, you can add whatever else you like at the end after the script name. The one rule that applies is that it must start with a slash, which differentiates it from the CGI script itself. Some scripts will generate URLs that are specifically designed to cause certain values to be passed to CGI via `PATH_INFO`. Here's an example. It presents a simple quiz asking the user what day it is. Answers are passed back to the same script by adding components to the end of the URL.

```
#!/usr/bin/env python
# CGI PATH_INFO example - Chapter 18 - pathinfo.cgi

import cgitb
cgitb.enable()

import cgi, time, os
```

```
monthmap = {1: 'January', 2: 'February', 3: 'March', 4: 'April', 5: 'May',
6: 'June', 7: 'July', 8: 'August', 9: 'September', 10: 'October',
11: 'November', 12: 'December'}
```

```
daymap = {0: 'Monday', 1: 'Tuesday', 2: 'Wednesday', 3: 'Thursday',
4: 'Friday', 5: 'Saturday', 6: 'Sunday'}
```

```
def print_month_quiz():
    print "What month is it?<P>"
    for code, name in monthmap.items():
        print '<A HREF="%s/%d">%s</A><BR>' % (os.environ['SCRIPT_NAME'],
                                                     code, name)
```

```
def print_day_quiz():
    month = time.localtime()[1]
    print "What day is it?<P>"
    for code, name in daymap.items():
        print '<A HREF="%s/%d/%d">%s</A><BR>' % (os.environ['SCRIPT_NAME'],
                                                     month, code, name)
```

```
def check_month_answer(answer):
    month = time.localtime()[1]
    if int(answer) == month:
        print "Yes, this is <B>%s</B>.<P>" % monthmap[month]
        return 1
    else:
        print "Sorry, you're wrong. Try again:<P>"
        print_month_quiz()
        return 0
```

```
def check_day_answer(answer):
    day = time.localtime()[6]
    if int(answer) == day:
        print "Yes, this is <B>%s</B>." % daymap[day]
        return 1
    else:
        print "Sorry, you're wrong. Try again:<P>"
        print_day_quiz()
        return 0
```

```

print "Content-type: text/html"
print

print """<HTML>
<HEAD>
<TITLE>CGI PATH_INFO Example</TITLE></HEAD><BODY>"""

input = os.getenv('PATH_INFO', '').split('/')[1:]

if not len(input):
    print_month_quiz()
elif len(input) == 1:
    ismonthright = check_month_answer(input[0])
    if ismonthright:
        print_day_quiz()
else:
    ismonthright = check_month_answer(input[0])
    if ismonthright:
        check_day_answer(input[1])

print "</BODY></HTML>"

```

There are several interesting things to note about this program. First, it generates links back to itself. It does that by using the `SCRIPT_NAME` environment variable to figure out where it is, then adds on the extra component for `PATH_INFO`. For instance, part of the generated source looks like this for the weekday quiz:

```

What day is it?<P>
<A HREF="/cgi-bin/pathinfo.cgi/2/0">Monday</A><BR>
<A HREF="/cgi-bin/pathinfo.cgi/2/1">Tuesday</A><BR>
<A HREF="/cgi-bin/pathinfo.cgi/2/2">Wednesday</A><BR>
<A HREF="/cgi-bin/pathinfo.cgi/2/3">Thursday</A><BR>
<A HREF="/cgi-bin/pathinfo.cgi/2/4">Friday</A><BR>
<A HREF="/cgi-bin/pathinfo.cgi/2/5">Saturday</A><BR>
<A HREF="/cgi-bin/pathinfo.cgi/2/6">Sunday</A><BR>

```

The `/cgi-bin/pathinfo.cgi` component of the URL was generated from `SCRIPT_NAME`. The remaining components were generated on the fly by the `for` loop.

Towards the end of the script, it looks at the `PATH_INFO` variable, splitting it into components separated by slashes. The first component is always empty (it's the empty string that would precede the first slash, or the entire empty string if no `PATH_INFO` is specified), so the `[1:]` strips it off.

If there's nothing in the `PATH_INFO`, the month quiz screen is displayed. Otherwise, if a month is present, the code checks to see if it's correct. If so, the day quiz is displayed. If both a month and weekday are specified, they're both validated, and appropriate screens are displayed.

The `PATH_INFO` method of interacting with a user has some limitations. It's not possible to use this simple method with submitted forms, which many people wish to do. However, some like the cosmetic purity of the URLs that are generated. This can be especially useful if your URLs convey some meaning (such as a hierarchy of information) or if you need to ensure that browsers get a particular filename (perhaps you're generating files that users download).

The GET Method

Although technically the previous method of interacting also used HTTP's GET method, when CGI programmers talk about the GET method, they normally mean it as a way of sending form submissions back to a server. Since GET values are sent as part of a URL, it's possible to generate GET strings without using an actual form. Here's an example that provides the same program as before, using GET strings instead of `PATH_INFO`.

```
#!/usr/bin/env python
# CGI GET example -- Chapter 18 - get.cgi

import cgitb
cgitb.enable()

import cgi, time, os

monthmap = {1: 'January', 2: 'February', 3: 'March', 4: 'April', 5: 'May',
            6: 'June', 7: 'July', 8: 'August', 9: 'September', 10: 'October',
            11: 'November', 12: 'December'}

daymap = {0: 'Monday', 1: 'Tuesday', 2: 'Wednesday', 3: 'Thursday',
          4: 'Friday', 5: 'Saturday', 6: 'Sunday'}

def print_month_quiz():
    print "What month is it?<P>"
    for code, name in monthmap.items():
        print '<A HREF="%s?month=%d">%s</A><BR>' % (os.environ['SCRIPT_NAME'],
                                                       code, name)
```

```

def print_day_quiz():
    month = time.localtime()[1]
    print "What day is it?<P>"
    for code, name in daymap.items():
        print '<A HREF="%s?month=%d&day=%d">%s</A><BR>' % \
            (os.environ['SCRIPT_NAME'], month, code, name)

def check_month_answer(answer):
    month = time.localtime()[1]
    if int(answer) == month:
        print "Yes, this is <B>%s</B>.<P>" % monthmap[month]
        return 1
    else:
        print "Sorry, you're wrong. Try again:<P>"
        print_month_quiz()
        return 0

def check_day_answer(answer):
    day = time.localtime()[6]
    if int(answer) == day:
        print "Yes, this is <B>%s</B>." % daymap[day]
        return 1
    else:
        print "Sorry, you're wrong. Try again:<P>"
        print_day_quiz()
        return 0

print "Content-type: text/html"
print

print """<HTML>
<HEAD>
<TITLE>CGI GET Example</TITLE></HEAD><BODY>"""

form = cgi.FieldStorage()

if form.getFirst('month') == None:
    print_month_quiz()
elif form.getFirst('day') == None:
    ismonthright = check_month_answer(form.getFirst('month'))
    if ismonthright:
        print_day_quiz()

```

```

else:
    ismonthright = check_month_answer(form.getFirst('month'))
    if ismonthright:
        check_day_answer(form.getFirst('day'))

print "</BODY></HTML>"
```

To the user, this program behaves identically to the previous one. The generated URL looks a little different, and it doesn't get put in PATH_INFO.

To the cgi library, things look exactly as they had if they had been submitted from a form. To access a form, or information from a GET URL, you use the cgi.FieldStorage() class. The cgi library will automatically parse the input and make it conveniently available via a FieldStorage() instance. Forms usually (but not always) use key and value pairs, and in this example, getfirst() is used to retrieve the value of a specific key.

FieldStorage instances commonly use the following two methods: getfirst() and getList(). It's also possible to access them as a dictionary, as you'll see in the upload example in the following section. If accessed as a dictionary, the keys represent the field names in a form or in the URL. The example first checks to see if a month is present (if not, None is returned by getfirst()). Then, it proceeds with the same logic as before, using form calls instead of manually parsing PATH_INFO.

The POST Method

The POST method is used exclusively to receive HTML form submissions. Although GET can also receive those submissions, POST is generally capable of handling larger amounts of data, including uploaded files. However, POST data doesn't show up in the URL, so users cannot bookmark a particular screen in a CGI script that uses POST. Sometimes, this is actually desirable, such as when the submitted data is particularly sensitive.

Here's a version of the GET example modified to use forms and POST:

```

#!/usr/bin/env python
# CGI POST example - Chapter 18 - post.cgi

import cgitb
cgitb.enable()

import cgi, time, os
```

```

monthmap = {1: 'January', 2: 'February', 3: 'March', 4: 'April', 5: 'May',
6: 'June', 7: 'July', 8: 'August', 9: 'September', 10: 'October',
11: 'November', 12: 'December'}

daymap = {0: 'Monday', 1: 'Tuesday', 2: 'Wednesday', 3: 'Thursday',
4: 'Friday', 5: 'Saturday', 6: 'Sunday'}

def print_month_quiz():
    print "What month is it?<P>"
    print '<FORM METHOD="POST" ACTION="%s">' % os.environ['SCRIPT_NAME']

    for code, name in monthmap.items():
        print '<INPUT NAME="month" TYPE="radio" VALUE="%d"> %s<BR>' % \
            (code, name)

    print '<INPUT TYPE="submit" NAME="submit" VALUE="Next &gt;&gt;">'
    print "</FORM>"

def print_day_quiz():
    month = time.localtime()[1]
    print "What day is it?<P>"
    print '<FORM METHOD="POST" ACTION="%s">' % os.environ['SCRIPT_NAME']
    print '<INPUT TYPE="hidden" NAME="month" VALUE="%d">' % month

    for code, name in daymap.items():
        print '<INPUT NAME="day" TYPE="radio" VALUE="%d"> %s<BR>' % \
            (code, name)

    print '<INPUT TYPE="submit" NAME="submit" VALUE="Next &gt;&gt;">'
    print "</FORM>"

def check_month_answer(answer):
    month = time.localtime()[1]
    if int(answer) == month:
        print "Yes, this is <B>%s</B>.<P>" % monthmap[month]
        return 1
    else:
        print "Sorry, you're wrong. Try again:<P>"
        print_month_quiz()
        return 0

```

```

def check_day_answer(answer):
    day = time.localtime()[6]
    if int(answer) == day:
        print "Yes, this is <B>%s</B>." % daymap[day]
        return 1
    else:
        print "Sorry, you're wrong. Try again:<P>"
        print_day_quiz()
        return 0

print "Content-type: text/html"
print

print """<HTML>
<HEAD>
<TITLE>CGI POST Example</TITLE></HEAD><BODY>"""

form = cgi.FieldStorage()

if form.getFirst('month') == None:
    print_month_quiz()
elif form.getFirst('day') == None:
    ismonthright = check_month_answer(form.getFirst('month'))
    if ismonthright:
        print_day_quiz()
else:
    ismonthright = check_month_answer(form.getFirst('month'))
    if ismonthright:
        check_day_answer(form.getFirst('day'))

print "</BODY></HTML>"
```

The logic at the end of this script is exactly the same as the logic from the GET example. In fact, you could replace the two occurrences of POST in this script with GET and wind up with a working script.

The major differences between this most recent example and the last one is the use of HTML forms. Our form presents the user with some radio buttons, one of which can be selected. There's also a Next >> button to accept the selections and advance to the next page.

Notice the “hidden” INPUT element in `print_day_quiz()`. This is used to cause some piece of data to be submitted along with the form without requiring the user to supply it. In the previous examples, the selected month was always trans-

mitted along with the day. The hidden element causes that to continue to happen with this form.

Python's `cgi` module supports both GET and POST for the `FieldStorage` class. In most cases, the two are interchangeable; just set the `METHOD` parameter in your `FORM` tag appropriately, and the `cgi` module will do the rest.

Escaping Special Characters

Both HTML documents and URLs have a set of special characters that can't be used directly. In HTML documents, the characters `<`, `>`, and `&` cannot be inserted literally into a document. Instead, in order to get those characters in the output, you must insert `<`, `>`, or `&` into your document. Similarly, there are characters that cannot be used in URLs or links; such characters include spaces and, depending on your particular situation, the question mark and ampersand. The process of converting strings that contain special characters that use the HTML sequences for them is called *escaping*.

This issue becomes a critical one of security when you display data that's read from users back to them. For instance, a web-based bulletin board will often display data supplied by site visitors. If the data supplied isn't escaped, a malicious visitor could insert codes to redirect visitors to a different page, or surreptitiously capture passwords or other information. This is called a *cross-site scripting* attack.

There are two main utilities that you can use to help with escaping: `cgi.escape()` and `urllib.quote_plus()`. Here's a program that demonstrates their use. It asks for input, and creates both a link and a display that properly escape any necessary characters in the input.

```
#!/usr/bin/env python
# CGI escape example - Chapter 18 - escape.cgi

import cgitb
cgitb.enable()

import cgi, os, urllib

print "Content-type: text/html"
print

print """<HTML>
<HEAD>
<TITLE>CGI Escape Example</TITLE></HEAD><BODY>"""

```

```

form = cgi.FieldStorage()

if form.getFirst('data') == None:
    print "No submitted data.<P>"
else:
    print "Submitted data:<P>"
    print '<A HREF="%s?data=%s"><TT>%s</TT></A><P>' % \
        (os.environ['SCRIPT_NAME'],
         urllib.quote_plus(form.getFirst('data')),
         cgi.escape(form.getFirst('data')))

print """<FORM METHOD="GET" ACTION="%s">
Supply some data:
<INPUT TYPE="text" NAME="data" WIDTH="40">
<INPUT TYPE="submit" NAME="submit" VALUE="Submit">
</FORM>
</BODY></HTML>"""\n os.environ['SCRIPT_NAME']

```

If you save this program as `escape.cgi` and run it, you can experiment with the results of escaping. Load it up and type `<&?+>` test into the box, then click Submit. That string should be echoed back to you in the generated page. But view the source of that page, and you'll see something like this:

```

<HTML>
<HEAD>
<TITLE>CGI Escape Example</TITLE></HEAD><BODY>
Submitted data:<P>
<A HREF="/cgi-bin/escape.cgi?data=%3C%26%3F%22%3E+test">
<TT>&lt;?+&gt; test</TT></A><P>
<FORM METHOD="GET" ACTION="/cgi-bin/escape.cgi">
Supply some data:
<INPUT TYPE="text" NAME="data" WIDTH="40">
<INPUT TYPE="submit" NAME="submit" VALUE="Submit">
</FORM>
</BODY></HTML>

```

The text within the `A` tag is escaped as necessary for URLs—that method of escaping is different from what's used in HTML itself; note the use of hex codes for the special characters and the plus for a space.

Then, the text between the `<TT>` tags is escaped for a HTML document. Notice that fewer characters are escaped, and they're escaped differently. It's important to use the proper method of escaping in each case. In general, you'll want to use `urllib.quote_plus()` for components of a URL and `cgilib.escape()` for everything else.

If you click on the link, you should get a page back that redisplays the same string you typed in so that you can verify that the escape worked. Your browser simply uses the escaped text in the URL, passing it back to the CGI script.

Handling Multiple Inputs per Field

There are ways that multiple values can be specified for a single name in HTML forms. You can supply checkboxes or “multiple” SELECT boxes. Or, you could assign the same name to more than one INPUT element.

The `cgi` module handles these cases by providing a `getlist()` method that's available to `FieldStorage` instances. The `getlist()` method returns a list of all values supplied for a particular field name. Here's an example that presents some lists to the user and shows what was selected.

```
#!/usr/bin/env python
# CGI list example - Chapter 18 - list.cgi

import cgitb
cgitb.enable()

import cgi, os, urllib

print "Content-type: text/html"
print

print """<HTML>
<HEAD>
<TITLE>CGI List Example</TITLE></HEAD><BODY>"""

form = cgi.FieldStorage()
print "You selected: "
selections = form.getlist('data')
printable = [cgi.escape(x) for x in selections]
print ", ".join(printable)
```

```

print """<FORM METHOD="GET" ACTION="%s">
Select some things:<P>"""
% os.environ['SCRIPT_NAME']
for item in ['Red', 'Green', 'Blue', 'Black', 'White', 'Purple',
    'Python', 'Perl', 'Java', 'Ruby', 'K&R', 'C++', 'OCaml', 'Haskell',
    'Prolog']:
    print '<INPUT TYPE="checkbox" NAME="data" VALUE="%s">' % cgi.escape(item)
    print '%s<BR>' % cgi.escape(item)

print """<INPUT TYPE="submit" NAME="submit" VALUE="Submit">
</FORM>
</BODY></HTML>"""

```

To test this program, run it, and select C++, Haskell, and Blue. Then click Submit.

Notice that the program listed all three items in the You Selected area. Notice also what the URL looks like (`http://localhost:8765/cgi-bin/list.cgi?data=Blue&data=C%2B%2B&data=Haskell&submit=Submit`) on my system. The data item was simply repeated three times. Other ways to generate multiple instances of the same field name will result in a URL looking like this as well.

Uploading Files

You may have encountered websites that allow you to upload files as part of an HTML form. These can be retrieved with a CGI script, and Python's `cgi` module supports this.

Here's an example that demonstrates uploading. It will allow the user to upload a file, and then displays the size and MD5 sum of the uploaded file. An MD5 sum is a unique checksum for a file; UNIX and Linux users can use the `md5` or `md5sum` command to check the MD5 sum of a file in advance, as follows:

```

#!/usr/bin/env python
# CGI file example - Chapter 18 - file.cgi

import cgitb
cgitb.enable()

import cgi, os, urllib, md5

print "Content-type: text/html"
print

```

```

print """<HTML>
<HEAD>
<TITLE>CGI File Example</TITLE></HEAD><BODY>"""

form = cgi.FieldStorage()
if form.has_key('file'):
    fileitem = form['file']
    if not fileitem.file:
        print "Error: not a file upload.<P>"
    else:
        print "Got file: %s<P>" % cgi.escape(fileitem.filename)
        m = md5.new()
        size = 0
        while 1:
            data = fileitem.file.read(4096)
            if not len(data):
                break
            size += len(data)
            m.update(data)
        print "Received file of %d bytes.  MD5sum is %s<P>" % \
              (size, m.hexdigest())
else:
    print "No file found.<P>

print """<FORM METHOD="POST" ACTION="%s" enctype="multipart/form-data">
File: <INPUT TYPE="file" NAME="file">
""" % os.environ['SCRIPT_NAME']
print """<INPUT TYPE="submit" NAME="submit" VALUE="Submit">
</FORM>
</BODY></HTML>"""

```

Note that this script uses POST for the form. This is required; file uploads aren't possible with GET. Also, observe the `enctype` attribute of the form. Again, this is required for file uploads to work.

The interface to work with uploaded files is the “old-style” or low-level `cgi` interface. Instead of using `getfirst()` or `getlist()` (which will load the entire file into memory, and then hand it over—this probably isn't desirable), you should instead look a little closer at the underlying data structures.

You can detect whether or not you have a file upload by looking at the `file` attribute. In this case, the `fileitem.file` test (equivalent to `form['file'].file`) does that. The name of the file, as supplied by the browser, is in the `filename` attribute.

It's theoretically possible to upload multiple files with the same attribute name. However, support for this is spotty both in browsers and CGI libraries, including Python's. Instead, if you need to receive multiple files from a single screen, you should instead provide multiple file INPUT tags, each with a different name. For instance, you may have file, file2, file3, and so on.

To test this program, first find a suitable file. If you're on a UNIX or Linux machine, use the `md5` or `md5sum` command to get the MD5 sum for the file. Then, pull up the `file.cgi` script in your web browser. You'll see an entry location for the filename. Select the file and click Submit. You'll get back a screen that looks like this:

```
Got file: bash
Received file of 628684 bytes. MD5sum is c7b805fd0322950f66a12a7b664b9cf4
```

Compare the MD5 sum to the one you calculated. You should be able to verify that they're identical.

Using Cookies

CGI authors frequently need to track user sessions. A session would be one continuous interaction between a user and a website. For instance, a user might be using a shopping site, viewing information on 20 separate products and adding 3 to a shopping cart. Although there may be several dozen individual pages requested, viewing the entire interaction as a single session is helpful. One common use for that is to display a count of items in the shopping cart on every page.

HTTP is inherently a *stateless* protocol, which means that each page view is its own session and there's no built-in way to associate one page view with another.

However, CGI authors often need such an association. For instance, if you're developing a shopping cart site, you would need to track a user's movement through the site to make sure that when "Add to Cart" is clicked, the cart for that user is used. This sort of thing often requires persistent storage of some sort—often a database to hold the session information and cart information.

There are several ways to track session information. For example, you could use HTTP authentication. If users must log in to access your site, and you use HTTP authentication, you can access the `REMOTE_USER` environment variable to get the username that was authenticated and use that as a session token.

Another method is to pass a session token around. This token can be a randomly generated item. It would be embedded in a hidden field on each form, or included at the end of each URL link. This works, but can be cumbersome.

Many developers today prefer the cookie mechanism. Cookies are small tokens that are stored on a user's machine. When the user accesses your site, the browser will return the previously stored cookie to you. A cookie can store any short string that you specify. You can then use this to track the session.

Cookies are also useful for other things. For instance, cookies can be used to store preferences for a site, since they're persistent and stored on a user's browser.

Mechanics of Cookies

When you wish to use a cookie, you'll emit an extra HTTP header before you serve up a page. This header contains the cookie you wish to place and some details about it. The user's browser will then store this cookie. On any future visits to the site, the cookie will be sent along as a HTTP header and the web server will place it in the `HTTP_COOKIE` environment variable.

You can actually set more than one cookie at once, and the user's browser can supply more than one cookie also.

Python provides a `Cookie` module to help with both the setting and retrieving of cookies. Each value is a `Morsel` object. Every `Morsel` has a name and a value. It also has a few attributes that are specified as part of the cookie standard in RFC2109. None are required. These attributes are

- `domain`. Gives the server on which the cookie is valid, starting with a dot. If omitted, defaults to the present server. As a security measure, many browsers will not accept cookies set to other domains.
- `max-age`. Specifies the maximum age of the cookie in seconds. If not specified, the cookie lasts until the user closes the browser. If set to 0, the cookie is deleted immediately. You can use this property to delete a previously stored cookie, which might achieve an effect such as logging out of a site.
- `path`. Gives the location on the server in which the cookie is valid. If not specified, it defaults to being valid on the entire server.
- `secure`. If specified, indicates that the cookie may only be transmitted over a secure connection (such as SSL-encrypted HTTP). This doesn't imply, however, that the cookie is stored securely by the web browser.
- `version`. Defaults to 1 and should not be modified.

Using Cookies

Here's an example of using cookies in a CGI program. This example will let you set a new cookie. If cookies are found, it will show them and let you delete them.

```

#!/usr/bin/env python
# CGI cookie example - Chapter 18 - cookie.cgi

import cgitb
cgitb.enable()

import cgi, os, urllib, Cookie

def getCookie():
    "Generates a Cookie object based on input"
    if os.environ.has_key('HTTP_COOKIE'):
        cookiestring = os.environ['HTTP_COOKIE']
    else:
        cookiestring = ''
    return Cookie.SimpleCookie(cookiestring)

def dispCookie():
    "Displays cookies found"
    cookie = getCookie()
    print "Found the following cookies:<UL>"
    foundcookies = 0
    for key in cookie.keys():
        morsel = cookie[key]
        print "<LI>%s: %s" % (cgi.escape(key), cgi.escape(morsel.value))
        foundcookies += 1
    print "</UL><P>"
    if foundcookies:
        print '<A HREF="%s?action=delCookie">Click here</A>' % \
            os.environ['SCRIPT_NAME']
        print ' to delete the testcookie.<P>'

def setCookie(value, maxage):
    "Sets a new cookie, sending appropriate output"
    cookie = getCookie()
    cookie['testcookie'] = value
    cookie['testcookie']['max-age'] = maxage
    print cookie.output()

```

```

print "Content-type: text/html"
form = cgi.FieldStorage()
action = form.getFirst('action')
if action == 'setCookie':
    # User requested setting a cookie
    setCookie(form.getFirst('cookieval'), 60*60*24*365)
    print # Signal end of the headers
    print """<HTML><HEAD><TITLE>Cookie Set</TITLE></HEAD><BODY>
The cookie has been set. Click <A HREF="%s">here</A> to return to the
main page.</BODY></HTML>"""\ % os.environ['SCRIPT_NAME']
elif action == 'delCookie':
    # User requested deleting a cookie
    setCookie('fake', 0)
    print # Signal end of the headers
    print """<HTML><HEAD><TITLE>Cookie deleted</TITLE></HEAD><BODY>
The cookie has been deleted. Click <A HREF="%s">here</A> to return to
the main page.</BODY></HTML>"""\ % os.environ['SCRIPT_NAME']
else:
    # No action requested by user. Just display cookies and offer
    # a new choice.
    print
    print """<HTML><HEAD><TITLE>CGI Cookie Example</TITLE></HEAD><BODY>"""
    dispCookie()
    print """<FORM METHOD="GET" ACTION="%s">"""\ % os.environ['SCRIPT_NAME']
    for value in ['Red', 'Green', 'Blue', 'White', 'Black']:
        print '<INPUT TYPE="radio" NAME="cookieval" VALUE="%s"> %s<BR>' % \
            (value, value)
    print '<INPUT TYPE="submit" NAME="action" VALUE="setCookie">
</FORM>
</BODY>
</HTML>"""

```

Save this as something like `cookie.cgi` and try running it. You'll be able to select a value for a cookie, and when you go back to the main page, you'll see that the cookie has that value. You can either change the value or delete the cookie.

Looking at the code, you can see that the `getCookie()` function fetches the cookie values sent by the client in the environment variable `HTTP_COOKIE` and passes that on to the `SimpleCookie` object.

In Python, a `Cookie` (or `SimpleCookie`) object holds a set of `Morsel` objects. Confusingly, each `Morsel` corresponds to what is normally called a cookie.

The `dispCookie()` function simply treats the `SimpleCookie` as a dictionary and goes over any cookies found within it. The `setCookie()` function will set a cookie. Note that while the `SimpleCookie` object looks like a regular Python dictionary, it

isn't. The line `cookie['testcookie'] = value` doesn't set the key `testcookie` to a string; rather, the `SimpleCookie` object creates a `Morsel` object whose value is `value`. That's why the next line works to set the maximum age of the cookie. Finally, `cookie.output()` is called. This generates the appropriate HTTP header lines to send to the client.

Farther down, take a look at what happens when a cookie is to be deleted. `setCookie()` is called. The value of the cookie is unimportant here. What matters is the maximum age, which is set to 0. A web browser should immediately discard a cookie when it sees the maximum age of 0.

Setting Multiple Cookies

It's possible to set multiple cookies for a client, but to do this, you should set multiple `Morsel` objects rather than multiple `Cookie` objects.

Summary

Many people need to make dynamic web pages, and CGI is one of the most popular ways to do so. To generate dynamic pages, a web server invokes the CGI script to render the page each time a request is received for it.

Python provides a `cgi` module that provides assistance for authors of CGI scripts. Information about the user's request is passed in the environment. Variables such as `PATH_INFO` can be used to access data that's passed along.

The `FieldStorage` class can be used to access forms submitted with GET or POST, and also URLs generated using GET-like syntax. By using its `getlist()` method, you can deal with forms that have multiple values for a single field name. `FieldStorage` can also handle file uploads.

When generating your documents, it's important to remember to escape special characters. The `cgi.escape()` function can do that for HTML text, and `urllib.quote_plus()` does the same thing for URLs.

Cookies are short strings stored on a user's computer. They're frequently used to track user sessions to enable things such as shopping carts. The Python `Cookie` module is used to set and access cookies stored in a user's browser.

CGI has some problems, mostly relating to performance. One way to boost performance is to use `mod_python` instead of CGI. The `mod_python` system is the topic of the next chapter.

mod_python

ONE OF THE MOST interesting ways to use Python today is Apache's mod_python module. The mod_python module actually embeds a fully functional Python interpreter inside the Apache web server. This is most frequently used as a powerful and efficient means to generate dynamic web pages, but has other uses as well.

In many ways, writing mod_python programs is similar to writing CGI programs, so familiarity with CGI (discussed in Chapter 18) will be beneficial as you learn about mod_python. Important differences do exist, and they'll be highlighted in this chapter.

Understanding the Need for mod_python

The most common method of generating dynamic web pages is the CGI script, which is discussed in Chapter 18. A CGI script is invoked each time a given page is requested. It reads the request, generates a reply, and then terminates. This mimics the operation of HTTP, which, at its core, works with a single request at a time. The next time a request is received, the CGI script is again invoked from scratch. This design enables CGI scripts to be both language- and server-neutral; indeed, virtually all popular web servers and programming languages support them.

However, this compatibility comes at a price: performance. Starting up a CGI script is slow. There's operating system overhead involved with creating a new process. There's overhead from the Python interpreter when initializing and loading the script. CGI scripts that connect to databases are hit especially hard, since they must establish a new database session each time a page is displayed. For these reasons, CGI scripts aren't suitable for high-traffic sites.

The mod_python module is one answer to these problems. It actually embeds a full Python interpreter inside the Apache web server. Your scripts are loaded once per server process and only initialized then. Database connections can be established at initialization time and kept open throughout the life of the web server process. Whenever a page needs to be generated, a particular function is called, and all the data about the request is passed to it. This function has access to the environment created at initialization time. So, for instance, it can reuse the existing database connection.

While this scheme forces the use of the Apache server, its advantages often outweigh its disadvantages, especially when designing a complete web application from the ground up. Python can be an effective alternative to special-purpose web languages like PHP.

The mod_python module actually can do more than simply serving up pages. It can also interact with the Apache system in various different ways. For instance, Apache provides various authentication handlers that let you authenticate users against a text file or LDAP database that contains usernames and passwords. You can write your own authentication handler in mod_python (perhaps it authenticates users against a remote XML-RPC server) and use that handler anywhere in Apache—even if the pages being used aren't generated by Python code.

Installing and Configuring mod_python

In this section, you'll learn how to install mod_python and configure Apache to use it. There are two popular versions of Apache: 1.3.x and 2.0.x. For Apache 1.3.x, you'll want to use mod_python version 2.7, and for Apache 2.0.x, you should use version 3.1 or above. The instructions and examples in this chapter are written for use with mod_python 3.1 and Apache 2.0. Due to the significant internal architectural changes between Apache 1.3.x and 2.0.x, these examples may not run under an Apache 1.3.x server. If you've a choice of which web server to deploy, I recommend Apache 2.0.x, since that will save you from needing to modify your mod_python code when you upgrade later.

The installation process for mod_python involves the following four steps:

1. Install Python.
2. Install Apache.
3. Install mod_python.
4. Configure Apache to use mod_python.

Since you're reading a book about Python, I'm assuming you already have it installed. Steps two and three vary depending on your operating system. Some operating-system suppliers or third parties supply prebuilt Apache and mod_python packages. If your supplier has done that, installing those prebuilt packages is certainly the quickest and easiest way to get mod_python up and running.

If you're installing manually, you'll first want to obtain and install Apache 2.0. You can download it from <http://httpd.apache.org/download.cgi>. When compiling,

make sure that your Apache is built with Dynamic Shared Objects (DSO) support. Most modern Apache installations are, but some—especially heavily customized ones or older installations—may not be. If you’re building from scratch, passing `--enable-so` to Apache’s `configure` script will usually enable the DSO mechanism.

Next, you’ll need to obtain and install mod_python. You can download it from <http://httpd.apache.org/modules/python-download.cgi>. Compilation and installation instructions for your particular platform can be found in the downloaded file or at www.modpython.org. Installation procedures can vary between different releases of Apache or mod_python, so please check your downloaded file or the mod_python website for the latest instructions.

Once you’ve installed everything, it’s important to verify that the individual components work. Make sure that you know how to make Apache serve up static HTML files, that it can actually do so, and that you know where to go to modify those files. Also, make sure that you’ve a working Python environment. If you have trouble with the mod_python installation later, it’s important to know that these two building blocks are both functional. Problems with either one of them could manifest themselves as mod_python problems later on.

You should also determine where Apache’s configuration files are stored on your system. Common locations include `/etc/apache`, `/etc/apache2`, `/etc/httpd`, `/usr/local/apache2`, `/usr/local/etc/httpd`, `/usr/local/etc/apache2`, or other similar locations. You should also identify the primary Apache configuration file, stored in the configuration directory. It’s usually named either `httpd.conf` or `apache2.conf`. Once you’ve done that, it’s time to configure Apache to use mod_python.

Loading the Module

The first thing to do when configuring mod_python for Apache is to make sure that the mod_python module is being loaded. This will require a `LoadModule` line in your Apache configuration file. It will look something like `LoadModule python_module /path/to/mod_python.so`. If you don’t know the path, consult the output from `make install` as part of the mod_python installation, or the information from your operating-system vendor if you installed mod_python via a package. Some operating systems may place examples in the `mods-available` directory in your Apache configuration area. Other operating systems may let you add something like `-D PYTHON` to `/etc/conf.d/apache2` to enable mod_python support.

You should be able to restart Apache with `apache2ctl restart` or `apachectl restart`. If it’s successful, and you’ve inserted the `LoadModule` line, you’ve successfully loaded the mod_python module into Apache.

Configuring Apache Directories

Now that the mod_python module is enabled, the next step is to activate it for your Python programs. The mod_python module is only activated for the areas and files you ask it to be used for. By default, it's not activated for any areas.

The first thing you need to do is either place your code under a directory that can already be reached via Apache, or set an Alias so that your code can be seen. For instance, if you placed your mod_python code in /usr/local/mod_python, you could use this configuration directive:

```
Alias /py /usr/local/mod_python
```

Requests to files under /py on the web server will actually use code from /usr/local/mod_python.

Now, you need to configure Apache to serve up Python code from there. Here's an example that you can place in your Apache configuration file (or .htaccess files, if you delete the first and last lines):

```
<Directory /usr/local/mod_python>
    AddHandler mod_python .prog
    PythonHandler test
    PythonDebug On
</Directory>
```

Save this file and adjust directory names if necessary. Now you'll need a program to test with. Here's some code that you can use. Name it test.py and place it in your mod_python directory, /usr/local/mod_python in this example. Note that unlike CGI scripts or standalone Python applications, you don't need to mark this script executable on UNIX or Linux platforms. Apache imports it directly into a running Python interpreter instead, as shown here:

```
# mod_python test example - Chapter 19 - test.py

from mod_python import apache
from sys import version

def writeinfo(req, name, value):
    req.write("<DT>%s</DT><DD>%s</DD>\n" % (name, value))
```

```

def handler(req):
    req.content_type = "text/html"
    if req.header_only:
        # Don't supply the body
        return apache.OK

    req.write("""<HTML><HEAD><TITLE>mod_python is working</TITLE>
</HEAD>
<BODY>
<H1>mod_python is working</H1>
You have successfully configured mod_python on your Apache system.
Here is some information about the environment and this request:
<P>
<DL>
""")

    writeinfo(req, "Client IP", req.get_remote_host(apache.REMOTE_NOLOOKUP))
    writeinfo(req, "URI", req.uri)
    writeinfo(req, "Filename", req.filename)
    writeinfo(req, "Canonical filename", req.canonical_filename)
    writeinfo(req, "Path_info", req.path_info)
    writeinfo(req, "Python version", version)

    req.write("</DL></BODY></HTML>\n")

    return apache.OK

```

Save this file. Then stop and start Apache as described earlier in this chapter.

NOTE Apache with mod_python doesn't always restart or reload properly when changes have been made to the mod_python configuration. After altering mod_python settings, you may need to completely stop and then start the Apache server.

You should now be able to access this document. If you used /py as an alias for the mod_python directory, you should be able to access the document at <http://localhost/py/test.prog>. You'll see a screen of information that says "mod_python is working" at the top. On my system, that information screen looked like this:

```
mod_python is working
```

You have successfully configured mod_python on your Apache system. Here is some information about the environment and this request:

Client IP
127.0.0.1

URI
/py/test.prog

Filename
/usr/local/mod_python/test.prog

Canonical filename
/usr/local/mod_python/test.prog

Path_info

Python version
2.3.3 (#1, Feb 24 2004, 09:29:13) [GCC 3.3.3 (Debian)]

You might notice that some of this information looks similar to the information that you can obtain from the environment in a CGI script, and indeed the information available to you via CGI is also available via the Apache API.

Fixing Configuration Problems

If you got an error instead, you'll need to fix your Apache configuration before proceeding with the examples in the remainder of this chapter. The Apache error log file often will have a hint that can help you solve the problem. This file usually is named something like `error.log` or `error_log` and is typically located in `/var/log/apache`, `/var/log/httpd`, or `/var/log/apache2`, though its location varies from system to system.

If you're still stuck, some of these hints may help:

- If Apache failed to even start (or your browser yields a “Connection refused” error), chances are that you have a problem with your configuration. Make sure that the `mod_python` module is being loaded and that there are no typos. The `apache2ctl configtest` or `apachectl configtest` commands can help find problems.

- If Apache started but generated a "4xx" error (such as a "not found" or "permissions" problem), make sure you have an Alias directive and that it points to the proper directory. Also make sure that the directory has permissions that allow the Apache server to access it. If you're using .htaccess files, make sure that Apache is configured to use them and to serve files from the directory in which they're located.
- If you got an internal server error, verify that you accurately typed in the example code and that the configuration file is correct. Then check your error log for a cause.
- If you see Python code instead of HTML output, check your Directory section. Make sure the path has no typos and that all necessary lines are present.
- Try stopping Apache. Wait one minute. Then start it up again. See if that fixes the problem.

If you still cannot solve the problem, try consulting the documentation and FAQ on mod_python's home page at www.modpython.org. You might also ask for assistance on the `comp.lang.python` newsgroup or the mod_python mailing list, which is available from http://mailman.modpython.org/mailman/listinfo/mod_python.

Understanding mod_python Basics

Let's look at the preceding example and see what went on under the hood. The first important pieces are in the Directory section you added to your Apache configuration file. The first line, `AddHandler mod_python .prog`, tells Apache that the `PythonHandler` should be used for a request for *any file ending in .prog* in that directory. That means you could also request `http://localhost/py/fake.prog` and get the same document. This is a powerful capability that actually lets you override Apache's normal document-selection logic; more on that later.

The `PythonHandler test` line effectively causes Apache to run `import test` when it initializes. Whenever a relevant request arrives, the `test.handler()` function will be called, and an Apache request object will be passed in.

The `PythonDebug` line requests that exception traces are sent to both Apache's error log and the client. Normally, these are sent solely to the error log, but you may find it easier to develop programs when you can see those errors in the web browser in real time.

Now let's look at the code for test.py and examine what happens on the Python side. The test.py code itself is loaded once per server process. Apache often creates several server processes when it's initialized; a fresh copy of test.py will be loaded and initialized for each server process. However, this will generally be done only once per server process. Server processes can also be created during the execution of Apache. This may happen, for instance, when server loads increase. Python scripts aren't always loaded immediately when a server process starts, but may instead be loaded when the first request arrives. Whenever a request for a Python program arrives, Apache calls handler(). This function receives the request and processes it. The return value from handler() indicates what sort of response gets sent back to the client.

The first thing handler() does is check to see if the request was for a header only. If so, that's all it hands out. Otherwise, it proceeds to generate the entire body. Many mod_python programs and CGI scripts don't make this check, and hand out the entire body regardless of the type of request. That usually works, but it's more polite for clients to do the right thing when possible.

The bulk of handler() is spent generating and writing the document being sent to the client. Note that you can use req.write() to transmit data to the client. Finally, an OK status code is returned.

The Role of the PythonHandler

While talking about the preceding PythonHandler line, I made the point that the handler is used for a request for *any file ending in .prog* in that directory. This is extremely important and may seem counterintuitive at first glance. For just about any other method of working with a web server, including both static HTML and CGI scripts, you expect the web server to select a different document based on the URL requested. You would also expect it to return an error itself if the requested document didn't exist.

Not so with the PythonHandler. With mod_python, *all* requests that match the AddHandler directive are passed to the Python handler. The Python handler is then left to determine what to do with them—use the request to select among different documents, return an error, or ignore the filename altogether as in the previous example.

This presents a tremendously powerful tool. You can, for instance, present a complete virtual hierarchy for a software download site. Or you can write your own logic to determine whether a given request is valid. Perhaps you use the filename to look up functions in a Python dictionary or to import your own modules.

Here are some different URLs. Given the preceding example program and configuration, see if you can figure out what Apache will return for each of the following:

- `http://localhost/py/test.prog`
- `http://localhost/py/nonexistent.prog`
- `http://localhost/py/somedir/test.prog`
- `http://localhost/py/somedir/nonexistent.prog`
- `http://localhost/py/nonexistent.html`

The first two examples both return the previous sample “mod_python is working” page. The second one does that because the Python handler for it is still `test.py`. You’ll notice that in the generated output, the URI is different. This is your key to differentiating between requests later on.

The last three examples will all give you a 404, File Not Found error. For the “`somedir`” examples, it’s because the handler is only defined for one specific directory in this example, and even though those are virtual subdirectories, the physical subdirectory does not exist. Apache will generate an error. For the last example, an error is generated because no special handler is defined for `.html` files. Processing reverts to Apache’s default handler, which just reads the file and sends it to the client. Since the file doesn’t exist, the client receives the error message.

This single difference between mod_python programs and other types of web programming represents the most common mod_python confusion. Make sure that you always remember that your single `PythonHandler` is called for all requests that match `AddHandler`.

One consequence of this is that the user-visible URLs need not end in `.prog` or `.py`. They could, in fact, have any extension—`.cgi` or even `.html` (though if you do that, make sure you’re really serving up HTML; you could confuse some browsers otherwise).

Handler Return Values

The return value of the handler governs what sort of HTTP status code Apache delivers to the client. These HTTP status codes are things like 200, Success, or 404, File Not Found. The full list of possible return-value constants is defined by Apache and listed in the mod_python documentation. Many are rarely, if ever, used. Here are the more popular ones:

- `apache.HTTP_OK` (200) indicates that the request is valid and that a document or header will be sent.
- `apache.HTTP_MOVED_PERMANENTLY` (301) and `apache.HTTP_MOVED_TEMPORARILY` (302) issue a HTTP redirect.
- `apache.HTTP_UNAUTHORIZED` (401) indicates that HTTP authorization is required or that it failed.
- `apache.HTTP_FORBIDDEN` (403) is a generic “permission denied” error unrelated to HTTP authentication.
- `apache.HTTP_NOT_FOUND` (404) is a generic “not found” message that’s used when a request doesn’t match anything valid.

In the brief example earlier in the chapter, the program always returned `apache.HTTP_OK` to indicate a successful request. Most `mod_python` scripts will, at minimum, also return `apache.HTTP_NOT_FOUND` in some situations.

It’s also possible to raise an exception that causes a particular result code to be transmitted to the client. For instance, consider the situation in which you’re checking the supplied URL to determine what function to perform. If you couldn’t find a match, you could call `raise apache.SERVER_RETURN` or `apache.HTTP_NOT_FOUND`.

Dispatching Requests

Earlier, I mentioned that Apache will call a single Python handler for all Python-related requests in a given directory, regardless of the filename. While this can provide powerful capabilities, for smaller projects, perhaps a more CGI-like interface is what’s desired. Calling a different URL would execute a different Python script. In fact, in this chapter, you’ll see a number of different example programs. It would be nice to be able to put them all in a directory and invoke them at will from a web browser.

By using Python’s dynamic importing features and doing some simple parsing of the URL, you can accomplish that. Here’s an example of a dispatcher program—one that receives requests and then sends them off to the appropriate script. With this example, you can split up your code with the same ease as you can with CGI. You can separate the logic for different pages, even using code from multiple sources in the same directory, and all of it will be invisible to the user.

```
# mod_python dispatcher - Chapter 19 - dispatcher.py

from mod_python import apache
import re

def raise404(logmsg):
    """Log an explanatory error message and send 404 to the client"""
    apache.log_error(logmsg, apache.APLOG_ERR)
    raise apache.SERVER_RETURN, apache.HTTP_NOT_FOUND

def gethandlerfunc(modname):
    """Given a module name from a URL, obtain the handler function from it
    and return the function object."""
    try:
        # Import the module
        mod = __import__(modname)
    except ImportError:
        # No module with this name
        raise404("Couldn't import module " + modname)

    try:
        # Find the handler function
        handler = mod.handler
    except AttributeError:
        # No handler function
        raise404("Couldn't find handler function in module " + modname)

    if not callable(handler):
        # It's not a function
        raise404("Handler is not callable in module " + modname)

    return handler

def gethandlername(URL):
    """Given a URL, find the handler module name"""
    match = re.search("/([a-zA-Z0-9_-]+)\.prog($|/\?)", URL)
    if not match:
        # Couldn't find the requested module
        raise404("Couldn't find a module name in URL " + URL)
    return match.group(1)
```

```

def handler(req):
    """Main entry point to the program. Find the handler function,
    call it, and return the result."""
    name = gethandlername(req.uri)
    if name == "dispatcher":
        raise404("Can't display the dispatcher")
    handlerfunc = gethandlerfunc(name)
    return handlerfunc(req)

```

This program grabs the filename from a URL and makes sure it fits the prescribed pattern (alphanumeric plus dashes or underscores). Then the `gethandlerfunc()` function is called. It imports the specified module name (given by the filename) and finds the `handler()` function in that module, which is then returned. The dispatcher's `handler()` function then calls the new `handler()` function, passing in `req` and returning the result. To the newly found script, in many ways the program behaves as if Apache had called it as a handler directly.

In a setting where persistent data is needed or performance is critical, you may wish to import all possible modules when `dispatcher.py` itself initializes, and hold them throughout the lifetime of the program. In this case, you'll have to know all the modules you'll use in advance.

To implement `dispatcher.py` on your Apache system, simply take the configuration file example from earlier. Change this line:

`PythonHandler test`

to the following:

`PythonHandler dispatcher`

Next, stop and start Apache. After restarting, you should still be able to display `test.prog`, except this time, the dispatcher is loading it. Also, nonexistent URLs will actually generate 404 errors like you would normally expect.

Dispatching and mod_python's Publisher

This dispatcher example uses the same concept as the Publisher handler that is distributed with mod_python. If you'll be doing a lot of work with a dispatcher, you may wish to investigate that handler as well.

In this chapter, a custom dispatcher was developed. The Publisher can lead to insecure code if not used carefully. A custom dispatcher also permits even more flexibility than Publisher.

Handling Input

Most people who wish to build dynamic websites will want to interact with the users. There are two primary ways of gathering input: via form (or form-like) fields, and with extra components on the URL. If you've read Chapter 18 (on CGI), you'll note that CGI presents exactly the same options. (If you're interested in the client side, Chapter 6 also discusses submissions from a client.) To help you understand how input works in mod_python, I've modified the CGI examples from Chapter 18. If you're contemplating a choice between CGI and mod_python, comparing the two examples is a great way to see the differences between the two technologies.

Extra URL Components

As with CGI, you can put whatever you like in the part of the URL that follows the Python handler. The extra part is saved as `req.path_info` in the object passed to your `handler()` function. Here's a version of the CGI demonstration of this same principle, modified to function as a standard mod_python program. This example will quiz you about today's date, as shown here:

```
# mod_python path_info example -- Chapter 19 -- pathinfo.py

from mod_python import apache
import time

monthmap = {1: 'January', 2: 'February', 3: 'March', 4: 'April', 5: 'May',
6: 'June', 7: 'July', 8: 'August', 9: 'September', 10: 'October',
11: 'November', 12: 'December'}

daymap = {0: 'Monday', 1: 'Tuesday', 2: 'Wednesday', 3: 'Thursday',
4: 'Friday', 5: 'Saturday', 6: 'Sunday'}

def getscriptname(req):
    if not len(req.path_info):
        return req.uri
    return req.uri[:-len(req.path_info)]

def month_quiz(req):
    req.write("What month is it? <P>\n")
    for code, name in monthmap.items():
        req.write('<A HREF="%s/%d">%s</A><BR>' % (getscriptname(req),
                                                       code, name))
```

```

def day_quiz(req):
    month = time.localtime()[1]
    req.write("What day is it?<P>\n")
    for code, name in daymap.items():
        req.write('<A HREF="%s/%d/%d">%s</A><BR>' % (getscriptname(req),
                                                       month, code, name))

def check_month_answer(req, answer):
    month = time.localtime()[1]
    if int(answer) == month:
        req.write("Yes, this is <B>%s</B>.<P>\n" % monthmap[month])
        return 1
    else:
        req.write("Sorry, you're wrong. Try again:<P>\n")
        month_quiz(req)
        return 0

def check_day_answer(req, answer):
    day = time.localtime()[6]
    if int(answer) == day:
        req.write("Yes, this is <B>%s</B>.\n" % daymap[day])
        return 1
    else:
        req.write("Sorry, you're wrong. Try again:<P>\n")
        day_quiz(req)
        return 0

def handler(req):
    req.content_type = "text/html"
    if req.header_only:
        return apache.OK

    req.write("""<HTML>
<HEAD>
<TITLE>mod_python PATH_INFO Example</TITLE></HEAD><BODY>""")

    input = req.path_info.split('/')[1:]

    if not len(input):
        month_quiz(req)
    elif len(input) == 1:
        ismonthright = check_month_answer(req, input[0])
        if ismonthright:
            day_quiz(req)

```

```

else:
    ismonthright = check_month_answer(req, input[0])
    if ismonthright:
        check_day_answer(req, input[1])

req.write("\n</BODY></HTML>\n")
return apache.OK

```

Taking a look at the program, you can see it's very similar to `pathinfo.cgi` from Chapter 18. In fact, the only changes necessary were to obtain data from `req` rather than the environment and to use `req.write()` instead of `print` to send data back to the client.

The program works by appending information after the script name in the URL. These new pieces of data are parsed off into `req.path_info` by Apache and are available to the program. Using those new pieces of data, the program can determine what the user supplied and generate an appropriate response.

If you've been using the example Apache configuration from earlier in the chapter, you can run this example by accessing `http://localhost/py/pathinfo.prog`. You'll first be asked what month it is. Once you answer correctly, you'll be asked what day of the week it is. When you finally get both correct, you'll see a message confirming your choices.

Why Not the CGI Handler?

The `mod_python` distribution includes a CGI handler that's designed to emulate the traditional Python CGI environment. However, due to the great differences between the Apache `mod_python` environment and a CGI environment, this emulation is imperfect and, in fact, results in the loss of many of the benefits of using `mod_python` in the first place. Therefore, neither `mod_python`'s authors nor I recommend using it. If you're going to be using `mod_python`, it's far better to rework your code to work with `mod_python` natively.

The GET Method

Instead of adding a virtual file path, the GET method can be used to pass data to the program. GET encodes parameters at the end of the URL. You can either construct the URL manually or use an HTML form to have the browser construct it for you based on input. Here is a modified version of the previous example that

uses the GET method. Like the previous example, this will present a quiz based on today's date:

```
# mod_python GET example -- Chapter 19 -- get.py

from mod_python import apache, util
import time

monthmap = {1: 'January', 2: 'February', 3: 'March', 4: 'April', 5: 'May',
6: 'June', 7: 'July', 8: 'August', 9: 'September', 10: 'October',
11: 'November', 12: 'December'}

daymap = {0: 'Monday', 1: 'Tuesday', 2: 'Wednesday', 3: 'Thursday',
4: 'Friday', 5: 'Saturday', 6: 'Sunday'}

def month_quiz(req):
    req.write("What month is it?<P>\n")
    for code, name in monthmap.items():
        req.write('<A HREF="%s?month=%d">%s</A><BR>' % (req.uri,
                                                               code, name))

def day_quiz(req):
    month = time.localtime()[1]
    req.write("What day is it?<P>\n")
    for code, name in daymap.items():
        req.write('<A HREF="%s?month=%d&day=%d">%s</A><BR>' % \
                  (req.uri, month, code, name))

def check_month_answer(req, answer):
    month = time.localtime()[1]
    if int(answer) == month:
        req.write("Yes, this is <B>%s</B>. <P>\n" % monthmap[month])
        return 1
    else:
        req.write("Sorry, you're wrong. Try again:<P>\n")
        month_quiz(req)
        return 0

def check_day_answer(req, answer):
    day = time.localtime()[6]
    if int(answer) == day:
        req.write("Yes, this is <B>%s</B>. \n" % daymap[day])
        return 1
```

```

else:
    req.write("Sorry, you're wrong. Try again:<P>\n")
    day_quiz(req)
    return 0

def handler(req):
    req.content_type = "text/html"
    if req.header_only:
        return apache.OK

    req.write("""<HTML>
<HEAD>
<TITLE>mod_python GET Example</TITLE></HEAD><BODY>""")

    form = util.FieldStorage(req)

    if form.getFirst('month') == None:
        month_quiz(req)
    elif form.getFirst('day') == None:
        ismonthright = check_month_answer(req, form.getFirst('month'))
        if ismonthright:
            day_quiz(req)
    else:
        ismonthright = check_month_answer(req, form.getFirst('month'))
        if ismonthright:
            check_day_answer(req, form.getFirst('day'))

    req.write("</BODY></HTML>\n")
    return apache.OK

```

In this case, we use mod_python's `util.FieldStorage` class to parse the GET request. This class is designed to be as compatible as possible with CGI's `FieldStorage` class. In fact, in Chapter 18 the code from the CGI GET example that deals with the form can be used almost unmodified in this situation. To this program, it doesn't matter whether things are submitted via a form or by manually generating a URL as was done in this example.

Notice that in this program, `req.uri` held the name of the Python script itself, whereas it didn't in the `pathinfo.py` example. When using the pathinfo style of input, Apache doesn't strip the input data out of `req.uri`, but it does when you're using the GET method of form submission. Therefore, no `getscriptname()` function was necessary in this example.

If you're using the example configuration from earlier in this chapter, you can access this example by loading `http://localhost/py/get.prog`. The interface will be the same as the `pathinfo.py` example.

The POST Method

The POST method receives HTML form submissions exclusively. Its chief advantages over GET tend to lie in the fact that it can handle much larger amounts of data. Sometimes, the fact that a POST result cannot be bookmarked is an advantage as well, such as when you're dealing with sensitive data. Here is yet another version of the month and day quiz example, except this time you'll use POST with mod_python, as follows:

```
# mod_python POST example -- Chapter 19 -- post.py

from mod_python import apache, util
import time

import cgi, time, os

monthmap = {1: 'January', 2: 'February', 3: 'March', 4: 'April', 5: 'May',
6: 'June', 7: 'July', 8: 'August', 9: 'September', 10: 'October',
11: 'November', 12: 'December'}

daymap = {0: 'Monday', 1: 'Tuesday', 2: 'Wednesday', 3: 'Thursday',
4: 'Friday', 5: 'Saturday', 6: 'Sunday'}

def month_quiz(req):
    req.write("What month is it?<P>\n")
    req.write('<FORM METHOD="POST" ACTION="%s">' % req.uri)

    for code, name in monthmap.items():
        req.write('<INPUT NAME="month" TYPE="radio" VALUE="%d"> %s<BR>' % \
                  (code, name))

    req.write('<INPUT TYPE="submit" NAME="submit" VALUE="Next &gt;&gt;">')
    req.write("</FORM>\n")

def day_quiz(req):
    month = time.localtime()[1]
    req.write("What day is it?<P>\n")
    req.write('<FORM METHOD="POST" ACTION="%s">' % req.uri)
    req.write('<INPUT TYPE="hidden" NAME="month" VALUE="%d">' % month)
```

```

for code, name in daymap.items():
    req.write('<INPUT NAME="day" TYPE="radio" VALUE="%d"> %s<BR>' % \
              (code, name))

req.write('<INPUT TYPE="submit" NAME="submit" VALUE="Next &gt;&gt;">')
req.write("</FORM>\n")

def check_month_answer(req, answer):
    month = time.localtime()[1]
    if int(answer) == month:
        req.write("Yes, this is <B>%s</B>.<P>\n" % monthmap[month])
        return 1
    else:
        req.write("Sorry, you're wrong. Try again:<P>\n")
        month_quiz(req)
        return 0

def check_day_answer(req, answer):
    day = time.localtime()[6]
    if int(answer) == day:
        req.write("Yes, this is <B>%s</B>.\n" % daymap[day])
        return 1
    else:
        req.write("Sorry, you're wrong. Try again:<P>\n")
        day_quiz(req)
        return 0

def handler(req):
    req.content_type = "text/html"
    if req.header_only:
        return apache.OK

    req.write("""<HTML>
<HEAD>
<TITLE>mod_python POST Example</TITLE></HEAD><BODY>""")

    form = util.FieldStorage(req)

    if form.getFirst('month') == None:
        month_quiz(req)
    elif form.getFirst('day') == None:
        ismonthright = check_month_answer(req, form.getFirst('month'))
        if ismonthright:
            day_quiz(req)

```

```

else:
    ismonthright = check_month_answer(req, form.getFirst('month'))
    if ismonthright:
        check_day_answer(req, form.getFirst('day'))

req.write("</BODY></HTML>\n")
return apache.OK

```

This program uses the exact same form logic as the GET version did. In fact, the only real change is the HTML code that was generated to present the menu of selections. The mechanics of handling the POST data are the same as for the CGI script thanks to the interface compatibility of FieldStorage.

Since the mod_python FieldStorage is largely compatible with CGI's FieldStorage, many of the principles of handling form data described in the CGI chapter apply to mod_python as well.

You can run this example by loading `http://localhost/py/post.prog` if you're using the example Apache configuration from earlier in this chapter.

Escaping

The mod_python module doesn't directly provide support for escaping HTML data or URLs. However, it's possible to use the functions in the `cgi` and `urllib` modules to do this.

Normally, you should never access anything from the CGI library in a mod_python program. Its `escape()` function is a special-case exemption. Here's a mod_python version of the escaping demonstration presented for CGI.

```

# mod_python escape example -- Chapter 19 -- escape.py

from mod_python import apache, util
import urllib
from cgi import escape

def handler(req):
    req.content_type = "text/html"
    if req.header_only:
        # Don't supply the body
        return apache.OK

```

```

req.write("""<HTML>
<HEAD>
<TITLE>mod_python Escape Example</TITLE></HEAD><BODY>""")

form = util.FieldStorage(req)

if form.getFirst('data') == None:
    req.write("No submitted data.<P>\n")
else:
    req.write("Submitted data:<P>\n")
    req.write('<A HREF="%s?data=%s"><TT>%s</TT></A><P>' % \
              (req.uri,
               urllib.quote_plus(form.getFirst('data')),
               escape(form.getFirst('data'))))

req.write("""<FORM METHOD="GET" ACTION="%s">
Supply some data:
<INPUT TYPE="text" NAME="data" WIDTH="40">
<INPUT TYPE="submit" NAME="submit" VALUE="Submit">
</FORM>
</BODY></HTML>\n""") % req.uri)

return apache.OK

```

Notice the careful import from the `cgi` module. You could, of course, use `import cgi` and then `cgi.escape()` instead of `from cgi import escape`. By using `from cgi import escape`, you remove the possibility that other functions from `cgi` could be accidentally used. Because the `cgi` functions assume the CGI environment of one process per request, they can cause serious conflicts with mod_python's very different environment.

Understanding Interpreter Instances

The mod_python system embeds Python inside Apache, but actually, in many situations, it will use more than one Python instance for the server. In the default configuration, the Python programs in each Apache virtual server execute in their own Python interpreter instance. That is, the Python code in one virtual server is completely unable to interact with the Python code in another virtual server because they exist in separate Python interpreters. This is usually a benefit, because it prevents errant or malicious Python programs from causing trouble with unrelated sites.

But sometimes it can be a problem. For instance, perhaps you have many virtual servers running the same Python program. If each virtual server uses its own Python environment, resource requirements will be increased on the web server. These increased requirements generally mean more RAM since more copies than necessary of your script will be loaded into RAM at once. It would also increase the number of connections to a database server that may be kept open.

In other situations, you may wish to increase the number of separate interpreters. This may occur when you run many different Python programs on a single virtual server. Increasing the number of separate interpreters will make it more difficult for a problem in one Python program to impact another one. For instance, a bug in one program may corrupt a database connection that's also used by another program.

The `mod_python` module defines three Apache configuration directives to control this behavior. `PythonInterpreter` gives fine-grained control over exactly how interpreters are used. It takes a single string parameter. Every Python program that falls under a `PythonInterpreter` with that name will use the same interpreter namespace. You can force the entire system to use one interpreter by placing something like `PythonInterpreter GLOBAL` at the top level of your server configuration. Note that the text `GLOBAL` could be replaced by any other name of your choosing.

There are also two other options: `PythonInterpPerDirectory` and `PythonInterpPerDirective`. These request separate interpreters for each directory or Apache directive area, respectively.

While these directives offer a certain level of control, they still may not necessarily do what you expect. For instance, specifying a single `PythonInterpreter` at the top level of your configuration doesn't necessarily mean that only one interpreter will exist; it just means that new interpreters aren't created based on the location of Python scripts.

Internally, Apache often uses a forking method to handle multiple client requests simultaneously. Each forked Apache process is its own entity, and thus each forked Apache process will have its own Python interpreter set. The rules just described govern how the interpreters *within a single forked process* can interact. With `mod_python`, there's no such thing as a true global variable that can be accessed by all Python programs for all connections. Interpreters within different forked processes are automatically isolated. As a programmer, you have no control over which forked process is called to handle a given connection, and you don't have control over the longevity of a given forked process. Given the forked-process model, you must, for example, be sure that your database server can handle an amount of simultaneous connections equal to the maximum number of forked processes in Apache.

Prebuilt Handlers in mod_python

In the examples in this chapter, handlers for mod_python were designed from scratch. The mod_python distribution ships three handlers that can be useful for your projects.

First, the Publisher handler is a more sophisticated version of the dispatcher example presented in this chapter. The Publisher handler presents not only Python scripts but also functions within them as paths. This can simplify your code in some cases, but unless due care is taken, the Publisher handler can also lead to security risks if certain functions are exposed unintentionally.

The CGI handler is designed to ease the migration from true CGI scripts to mod_python scripts. If you have existing CGI scripts and wish to move to mod_python, this may help you. However, the mod_python authors warn against using this for new development, as many of the benefits of mod_python are forfeited. Additionally, some CGI scripts that take advantage of the nature of CGI to perform process-altering tasks, such as changing directories or the environment, may cause failures when used with the CGI handler.

Finally, the Python Server Pages (PSP) handler is designed to process HTML or XHTML documents, and allows you to embed Python code inside them. This is similar in concept to PHP programming.

Summary

The mod_python module is a way of embedding Python inside the Apache web server. It can often provide increased performance and greater flexibility if you're able to standardize on the Apache web server for your projects.

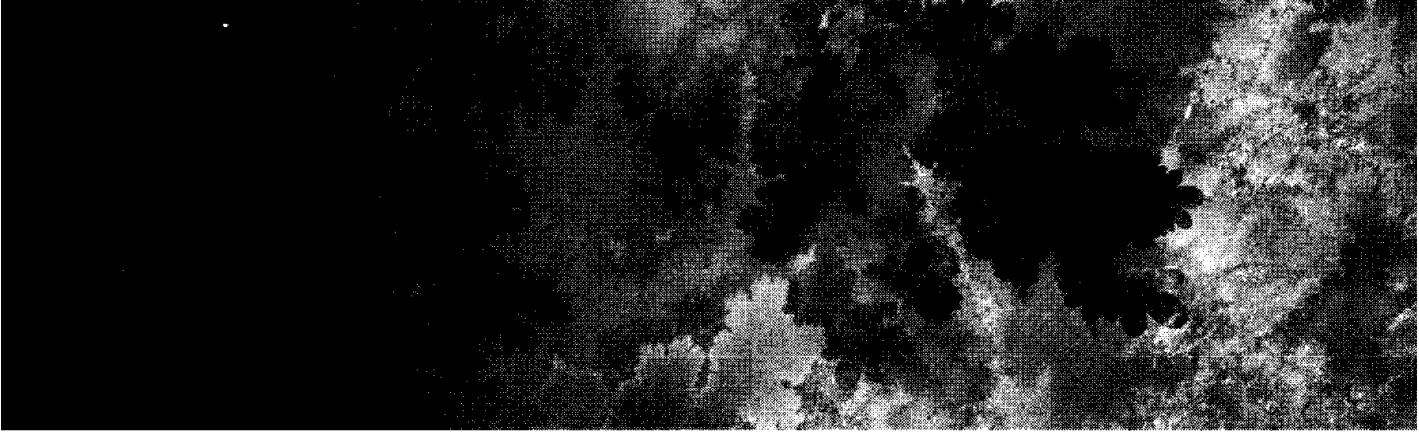
Installation of mod_python varies depending on your system, but generally involves installing Python, Apache, and then the mod_python module. You'll need to add a few lines of code in the Apache configuration file as well.

With mod_python, all requests in a particular directory that match the handler pattern are passed to a single Python function. If you want a more CGI-like behavior, in which programs with different names handle requests for different files, you can use a program similar to the dispatcher example in this chapter to send them off to the appropriate handlers. The mod_python-supplied Publisher handler also performs a similar service.

Programs written with mod_python have options similar to CGI programs for receiving and sending data, though the mechanics of accomplishing these things can differ. The mod_python module provides a `util.FieldStorage` class that's designed to mimic CGI's `FieldStorage` class, which means that form-processing code is often quite similar between CGI and mod_python programs. The `escape()` function can be used directly from the `cgi` module.

Apache normally separates Python programs so that programs running for one virtual server cannot access programs running in another. However, this behavior can be changed. In any case, Apache may create several Python interpreters due to its internal multitasking mechanisms.

The mod_python distribution includes three built-in handlers that may be able to save you some work: Publisher, which is a more sophisticated dispatcher; CGI, which can help migrate existing CGI scripts to a mod_python system; and PSP, which processes Python code embedded in HTML or XHTML documents. When using any of these built-in handlers, make sure you understand the security implications first.



Part Six

Multitasking

Forking

VIRTUALLY ALL AUTHORS of servers, and many authors of clients, need to write programs that can effectively handle multiple network connections simultaneously. As an example, consider a web server. If your server could only handle one connection at a time, you could only be transmitting a single page at a time. If you have a large file on your server and a user on a slow link is downloading it, that user could completely tie up your server for an hour or more. During that time, nobody else would be able to view any pages on that server. Virtually all servers want to be able to serve more than one client at once.

To serve multiple clients simultaneously, you need to have some way to handle several network connections at once. Python provides three primary ways to meet that objective: forking, threading, and asynchronous I/O (also known as nonblocking sockets). I'll cover all three: forking in this chapter, threading in Chapter 21, and asynchronous I/O in Chapter 22.

Of these three, forking is probably the easiest to understand and use. However, it's not completely portable; forking may be unavailable on platforms that aren't derived from UNIX.

Forking involves *multitasking*—the ability to run multiple processes at once, or to simulate that ability. In this chapter, you'll learn how to apply forking to your programs. First, you'll learn about how forking works with your operating system and some common pitfalls to avoid. Next, you'll see how to apply forking to server programs. Finally, the chapter will conclude with information on locking and error handling.

Understanding Processes

Forking is tied in closely with the operating system's nature of a process. A *process* is usually defined as “an executing instance of a program.” When you start up an editor such as Emacs, the operating system creates a new process that runs it. When Emacs terminates, that process goes away. If you open up two copies of Emacs, there will be two Emacs processes running. Although they both may be instances of `/usr/bin/emacs` and may be started up the same way, they may be doing different things—perhaps editing different files. Each process is distinct.

Each process has a unique identification number called a *process ID* (PID). The operating system assigns the PID to a process when it starts. In the preceding Emacs example, the two Emacs processes would each have a unique PID.

NOTE This chapter focuses on UNIX and Linux platforms, since those are platforms for which forking is best supported. The information contained in this chapter may not apply to other operating systems such as Windows.

However, although the details may differ in important ways, all multitasking operating systems (including Windows) have some notion of a process, even if they don't refer to it by that name. Single-tasking operating systems, such as DOS, will usually have no notion of a process.

You can gather information about running processes by using the `ps` command. The syntax for `ps` differs from one UNIX to the next. Here's an example from Linux, which should also work on any BSD operating system and AIX:

```
$ ps x
  PID TTY      STAT   TIME COMMAND
19817 ?          S      0:00 /bin/sh /usr/bin/startkde
19866 ?          Ss     0:00 /usr/bin/ssh-agent startkde
19877 ?          Ss     0:02 kdeinit: Running...
19880 ?          S      0:18 kdeinit: dcopserver --nosid
19882 ?          S      0:01 kdeinit: klauncher
19885 ?          S      0:26 kdeinit: kded
19966 ?          S      34:11 /usr/bin/artsd -F 5 -S 4096 -a alsa -s 60 -m artsmess
...
12096 ?          S      0:00 xterm
12097 pts/668  Ss     0:00 -bash
12154 pts/668  S+    0:00 emacs -nw letter.txt
12155 ?          S      0:00 xterm
12156 pts/669  Ss     0:00 -bash
12163 pts/669  S+    0:00 emacs -nw report.txt
```

The first column in the `ps` output shows the PID of a given process. The last column, in most cases, shows what program that process is executing. In this example, the first processes listed correspond to the graphical environment KDE. They represent things such as the sound system. I've cut out several dozen other processes for this example.

Farther down, you can see two different versions of Emacs running as in the previous example. One has PID 12154 and the other has PID 12163. The first was originally started to edit `letter.txt` and the second was started to edit `report.txt`.

Each process has unique attributes. For instance, PID 12154 may have an open file descriptor for `letter.txt` while PID 12163 may have an open file descriptor for `report.txt`. Processes can also have unique environment variables, data in memory, and open network connections.

The process is the fundamental unit of multitasking. Several processes may be running simultaneously. For instance, my two Emacs processes could be running at the same time as a web browser, a file downloading process, a data analysis process, and a CD burning process. A single process doesn't have more than one thing executing at once. Threading, discussed in Chapter 21, can blur that line.

Understanding `fork()`

The system call used to implement forking is called `fork()`. It's a very unique call. Most functions will return exactly once (with or without a value). The `sys.exit()` function never returns since it terminates the program. By contrast, Python's `os.fork()` is the only function that actually returns *twice*. After calling `fork()`, there are two copies of your program running at once. But the second copy doesn't restart from the beginning; both copies continue directly after the call to `fork()`—the process's entire address space is copied. Errors are possible, and `os.fork()` could raise an exception; see the "Error Handling" section in this chapter for details.

The `fork()` call returns the process ID (PID) of the newly created process to the original ("parent") process. To the new ("child") process, it returns a PID of 0. Therefore, logic like this is common:

```
def handle():
    pid = os.fork()
    if pid:
        # Parent
        close_child_connections()
        handle_more_connections()
    else:
        # Child
        close_parent_connections()
        process_this_connection()
```

When I said before that `os.fork()` is the only function that returns twice, that's not entirely accurate. I could write the following:

```
def dothefork():
    pid = os.fork()
    if pid:
        return "server"
    else:
        return "client"
```

In this instance, `dothefork()` would actually return twice as well. It should be noted, though, that any function that returns twice does, at some point, call `os.fork()` to make that possible.

Forking is one of the most common and best-understood methods of multi-tasking, and using forks is especially common for servers, whereby the server typically forks for each new incoming request.

After a fork, each process has a distinct address space. Modifying a variable in one process will not modify it in another, and that is a key difference from threads (discussed in the next chapter). This leaves your code less vulnerable to errors that may cause the server process for one connection to interfere with that of another.

Forking is used, on UNIX systems, for more than just network purposes. For instance, the typical way (and what Python does under the hood when you call `os.system()`) to execute a program is to fork and then use one of the `os.exec...` functions to start the new program. The parent process can then continue on, monitoring the child, or it can opt to have its execution blocked until the child terminates by using one of the `wait()` functions (which will be described later in the section “The Zombie Problem”).

However, forking is a fairly low-level operation. The process of actually doing a fork takes a little bit of work to make sure that you’re doing everything the operating system expects of you.

Duplicated File Descriptors

There are several side effects of forking. One of the most obvious is that of duplicated file descriptors. A file descriptor can refer to things such as a socket, a file on disk, a terminal (standard input/output/error), or certain other file-like objects.

Since a forked copy of a process is an exact copy, it inherits all the file descriptors and sockets that the parent process had. So you wind up with a situation in which both the parent and child process have a connection open to a single remote host.

That’s bad for several reasons. One is that if both processes try to communicate over that socket, the result will likely be garbled. Another is that a call to `close()` doesn’t actually close the connection until *both* processes have called it.

Therefore, protocols (such as FTP) that use the closing of a socket as a signal that some action has completed will be broken unless sockets are closed both places. Some authors do, on occasion, exploit the fact that two processes can access the socket, but this requires great care and is quite rare.

The solution to this problem is to have whichever process doesn't need a socket close it immediately after forking. For the typical case of a server that forks a new process to handle each incoming request, you'll notice that the parent process will close the socket for the child, and the child will close the master listening socket that the parent uses. This will ensure proper operation for both processes.

Zombie Processes

The semantics of `fork()` are built around the assumption that the parent process is interested in finding out when and how a child process terminated. For instance, a shell script is interested in finding out the exit code from a program that is run. A parent process can find out not just the exit code, but also if a process crashed or terminated due to a signal. The way a parent gathers this information is via `os.wait()` or a similar call.

During the time between the termination of the child process, and the time the parent calls `wait()` in it, the child process is said to be a *zombie* process. It's no longer executing, yet certain memory structures are still present in order to permit the parent to `wait()` on it.

For most servers, the information returned by `wait()` is irrelevant. If a worker process dies, the server will not do anything different; it should still go on servicing requests from other clients.

However, you must still call `wait()` on the child process at some point after it terminates. Otherwise, system resources will be consumed by the vast amount of zombie processes, which could eventually render the server machine unusable.

The operating system makes that job fairly easy, though. Each time a child process terminates, it sends the `SIGCHLD` signal to its parent process. (A signal is a rudimentary way to inform a process of certain events.) The parent process can set a signal handler to receive `SIGCHLD` and clean up any children that have terminated. While this sounds tricky, I'll show you an example in the "The Zombie Problem" section later in this chapter that can accomplish this very easily.

If the parent process dies before its children, the children will continue running. The system re-parents them, setting their parent to be `init` (process 1). The `init` process will then take care of cleaning up zombies.

Performance

You may think that using `fork()` is a slow proposition since it must copy over all of a server each time a client connects. In reality, the performance hit of `fork()` is insignificant and unnoticeable to all but the most heavily loaded systems.

Most modern operating systems, such as Linux, implement `fork()` with copy-on-write memory. That means that memory isn't actually copied until it needs to be (when one process or the other modifies it). The call to `fork()` itself is usually virtually instantaneous.

The `fork()` call is used all over in the system. For instance, when you're using a shell and type `ls`, the shell will fork a copy of itself, and the new process will invoke `ls`. A similar thing happens if you click an icon to launch a program in a graphical environment. The desktop manager or window manager will fork itself, and then call `exec()` to start the new program. When you call `os.system()` from a Python program, there's an internal call for `fork()` and `exec()` in the same manner.

Extremely heavily loaded systems that serve many brief connections, such as web servers for very popular sites, may not want to put up with even the small overhead of forking. These servers sometimes use a *forked pool*, in which the forking is done in advance and processes are reused. They might also choose to use asynchronous I/O, which has no per-process overhead, or threading, which has less of an overhead. For general-purpose use, forking remains a good choice.

Forking First Steps

Here's a simple first example of forking. It's going to fork, and both processes will display some messages.

```
#!/usr/bin/env python
# First fork example - Chapter 20 - firstfork.py

import os, time

print "Before the fork, my PID is", os.getpid()

if os.fork():
    print "Hello from the parent.  My PID is", os.getpid()
else:
    print "Hello from the child.  My PID is", os.getpid() . .

time.sleep(1)
print "Hello from both of us."
```

This program will print out its process ID prior to forking. Then, because `fork()` returns twice, the parent and child each print out a unique message, and they both fall out of the `if`, wait for one second, then display a greeting. Here's what the output looks like:

```
$ ./firstfork.py
Before the fork, my PID is 2700
Hello from the child. My PID is 2701
Hello from the parent. My PID is 2700
... one second later ...
Hello from both of us.
Hello from both of us.
```

On some systems, you may observe that the order of the parent and child messages is different, and they may be different each time you run the program. The operating system makes no guarantee about that, as in fact, both processes should be executing simultaneously.

Notice how `Hello from both of us` is displayed twice, even though it occurs in the code only once. That's because, by the time the execution reaches that point, there are actually two copies of the program running.

The Zombie Problem

Let's take a look at the aforementioned zombie problem in action. The UNIX command `ps` shows a list of active processes. Here's an example that will demonstrate the zombie problem. While it's running, open up another terminal session and take a look at the state of processes.

```
#!/usr/bin/env python
# Zombie problem demonstration - Chapter 20 - zombieprob.py

import os, time

print "Before the fork, my PID is", os.getpid()

pid = os.fork()
if pid:
    print "Hello from the parent. The child will be PID %d" % pid
    print "Sleeping 120 seconds..."
    time.sleep(120)
```

The child process will terminate immediately after the fork (`fork()` returns PID 0 for the child, so it will fail the `if` test, and there's nothing else for it to do). The parent doesn't clean it up, but rather waits around for a while. Run the program as follows:

```
$ ./zombieprob.py
Before the fork, my PID is 2719
Hello from the parent. The child will be PID 2720
Sleeping 120 seconds...
```

Now, in another terminal session, inspect the results without stopping the program:

```
$ ps ax | grep 2719
2719 pts/2      S      0:00 python ./zombieprob.py
$ ps ax | grep 2720
2720 pts/2      Z      0:00 [python] <defunct>
```

You can see that the child process is a zombie; the `Z` in the third column, as well as the `<defunct>` at the end of the output, indicate that. Once the parent terminates, you'll be able to confirm that neither process exists. The shell cleans up the parent process, and the child process gets re-parented to `init`, which will clean it up.

The Role of `init`

The `init` program is always the first process that runs on the system and always has PID 1. Its main roles are starting up and shutting down the system. In this case, there's another special role for `init`. If a process dies, and there are still children of it out there on the system (zombie or not), the operating system will change that process's parent to be PID 1—`init`. The `init` program will watch for zombie children in the same way that normal processes will, so these processes will get cleaned up.

Solving the Zombie Problem with Signals

Here's a program that solves the zombie problem:

```

#!/usr/bin/env python
# Zombie problem solution - Chapter 20 - zombiesol.py

import os, time, signal

def chldhandler(signum, stackframe):
    """Signal handler. Runs on the parent and is called whenever
    a child terminates."""
    while 1:
        # Repeat as long as there are children to collect.
        try:
            result = os.waitpid(-1, os.WNOHANG)
        except:
            break
        print "Reaped child process %d" % result[0]
    # Reset the signal handler so future signals trigger this function
    signal.signal(signal.SIGCHLD, chldhandler)

# Install signal handler so that chldhandler() gets called whenever
# child process terminates.
signal.signal(signal.SIGCHLD, chldhandler)

print "Before the fork, my PID is", os.getpid()

pid = os.fork()
if pid:
    print "Hello from the parent. The child will be PID %d" % pid
    print "Sleeping 10 seconds..."
    time.sleep(10)
    print "Sleep done."
else:
    print "Child sleeping 5 seconds..."
    time.sleep(5)

```

First, the program defines the signal handler `chldhandler()`. This function is called whenever `SIGCHLD` is received. It has a simple loop calling `os.waitpid()`. The first argument to `os.waitpid()`, `-1`, means to wait for any terminated child process, and the second tells it to return immediately if no more terminated processes exist. If there are child processes waiting, `waitpid()` returns a tuple of a process's PID and exit information. Otherwise, it raises an exception. The act of using `wait()` or `waitpid()` to collect information about terminated processes is called *reaping*.

The call is in a loop because a single `SIGCHLD` could indicate multiple child processes have died. Finally, after the loop, the signal handler is reactivated. This

is necessary because some UNIX implementations deactivate the signal handler when it's called. By explicitly reactivating it, you ensure that it gets called again when the next child process terminates. (It won't happen in this example, but it will in real servers.)

The call to `signal.signal()` establishes the signal handler. The first argument is the signal of interest, and the second one names the function that should be called when it arrives. That function must accept two arguments: the signal number and an optional stack frame.

The remainder of the program is fairly typical. When you run the program, you'll see output like this:

```
$ ./zombiesol.py
Before the fork, my PID is 2931
Child sleeping 5 seconds...
Hello from the parent. The child will be PID 2932
Sleeping 10 seconds...
Reaped child process 2932
Sleep done.
```

You'll notice that the parent reaps the child process only five seconds into its sleep, since that's how long it takes before the child process terminates. The signal handler is called immediately.

You might also notice that the parent process never finishes its sleep. There's a special case with `time.sleep()` in that if any signal handler is called, the sleep will terminate immediately, rather than continue waiting the remaining amount of time. Since you'll rarely need to use `time.sleep()` with networking code, this shouldn't be an issue.

Solving the Zombie Problem with Polling

Another approach to solving the zombie problem is to periodically check for zombie children. This method doesn't involve a signal handler, and as such, will not cause problems for `sleep()`. Signal handlers can also cause problems with I/O functions on some operating systems, which is a larger problem for network clients.

Here's another solution to the zombie problem. Instead of using a signal handler, it will periodically try to collect any zombie processes.

```
#!/usr/bin/env python
# Zombie problem solution with polling - Chapter 20 - zombiepoll.py

import os, time

def reap():
    """Try to collect zombie processes, if any."""
    while 1:
        try:
            result = os.waitpid(-1, os.WNOHANG)
        except:
            break
        print "Reaped child process %d" % result[0]

    print "Before the fork, my PID is", os.getpid()

pid = os.fork()
if pid:
    print "Hello from the parent. The child will be PID %d" % pid
    print "Parent sleeping 60 seconds..."
    time.sleep(60)
    print "Parent sleep done."
    reap()
    print "Parent sleeping 60 seconds..."
    time.sleep(60)
    print "Parent sleep done."
else:
    print "Child sleeping 5 seconds..."
    time.sleep(5)
    print "Child terminating."
```

This program will simply call `reap()` to gather up the child processes. This function is very similar to the signal handler in the previous example. A server process would probably call `reap()` at the bottom of its primary `accept()` loop. While there will sometimes be zombie processes out there, they won't build up, since new ones would be created only after cleaning up the older ones.

When you run this problem, you'll see output like this:

```
$ ./zombiepoll.py
Before the fork, my PID is 3667
Child sleeping 5 seconds...
Hello from the parent. The child will be PID 3668
Parent sleeping 60 seconds...
Child terminating.
Parent sleep done.
Reaped child process 3668
Parent sleeping 60 seconds...
Parent sleep done.
```

If you run the program, you'll notice several differences between it and the previous one. First of all, the child process wasn't reaped immediately when it terminated. Secondly, the call to `time.sleep()` wasn't interrupted. Finally, if you do a `ps` during the 55 seconds between the time the child exits and the time it's reaped, you'll see it listed as a zombie. But you can see that it's been cleaned up during the last 60 seconds of the program.

Forking Servers

Forking is most commonly used for network servers. I presented code for several different servers in Chapter 3, but each sample shared a common problem: It could only serve one client at a time. This is rarely an acceptable limitation, and forking is one of the most common ways to solve the problem. The concepts demonstrated earlier can be applied to the server code. Here's an example of an echo server that uses forking. Because it uses forking, it can echo text back to several clients at once.

```
#!/usr/bin/env python
# Echo Server with Forking - Chapter 20 - echoserver.py
# Compare to echo server in Chapter 3

import socket, traceback, os, sys

def reap():
    # Collect any child processes that may be outstanding
    while 1:
        try:
            result = os.waitpid(-1, os.WNOHANG)
            if not result[0]: break
```

```
except:  
    break  
print "Reaped child process %d" % result[0]  
  
host = ''                      # Bind to all interfaces  
port = 51423  
  
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)  
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)  
s.bind((host, port))  
s.listen(1)  
  
print "Parent at %d listening for connections" % os.getpid()  
  
while 1:  
    try:  
        clientsock, clientaddr = s.accept()  
    except KeyboardInterrupt:  
        raise  
    except:  
        traceback.print_exc()  
        continue  
  
    # Clean up old children.  
    reap()  
  
    # Fork a process for this connection.  
    pid = os.fork()  
  
    if pid:  
        # This is the parent process. Close the child's socket  
        # and return to the top of the loop.  
        clientsock.close()  
        continue  
    else:  
  
        # From here on, this is the child.  
        s.close()                      # Close the parent's socket  
  
        # Process the connection
```

```

try:

    print "Child from %s being handled by PID %d" % \
          (clientsock.getpeername(), os.getpid())
    while 1:
        data = clientsock.recv(4096)
        if not len(data):
            break
        clientsock.sendall(data)
    except (KeyboardInterrupt, SystemExit):
        raise
    except:
        traceback.print_exc()

    # Close the connection

    try:
        clientsock.close()
    except KeyboardInterrupt:
        raise
    except:
        traceback.print_exc()

    # Done handling the connection. Child process *must* terminate
    # and not go back to the top of the loop.
    sys.exit(0)

```

Let's look at this program, which is the TCP echo server from Chapter 3 with forking added in. Now it can handle multiple clients simultaneously.

First, the function `reap()` is defined similarly to the previous examples. However, there's an additional test: to see whether or not the PID returned by `waitpid()` is zero. In the previous cases, this test was skipped, since we always knew that `reap()` was called when there was at least one zombie process, but that might not be the case here.

Then, the code proceeds unmodified until after the call to `accept()`. The first new call is to `reap()`. This will clean up any zombie processes that have terminated since the last time a client connected. Next, the program forks and uses the usual `if pid` design.

If the process post-fork is the parent, it will close the child's socket and return to the top of the loop with `continue` to list for more connections. If we're in the child process, it closes the parent process's socket and then processes the connection as usual. However, there's a change at the end—the child calls `sys.exit(0)` when it's done processing. This is vitally important. If it didn't do this, execution would

return to the top of the while loop, and the child would try to accept new connections as well as the parent. In this particular case, it will generate an error since the client closed its copy of the master socket. The `sys.exit()` makes sure that the client terminates when it should.

Try running the program. You can then connect to port 51423 and observe that it echoes text back to you. On the console, the server will print out some status messages. Here's what it looked like for me:

```
$ ./echoserver.py
Parent at 16271 listening for connections
Child from ('127.0.0.1', 37708) being handled by PID 16273
Child from ('127.0.0.1', 37709) being handled by PID 16285
```

This shows two incoming connections being handled by two different processes.

Locking

A simple program like an echo server never needs to write to any files on the local system. However, this isn't necessarily the case for all servers. When using forking, you have to be wary of concurrency issues that don't occur if you only service one connection at once.

For instance, if part of the task of your server is to write lines to a file, it would be a problem to have two servers writing to the file at once. Changes could be lost or corrupted, and the two processes could overwrite each other's changes.

To solve this problem, you'll need to use locking. In forking programs, locking is most frequently used to control access to files. Locking lets you force only one process to perform certain actions at a time. Here's an example of a forking server that uses locking:

```
#!/usr/bin/env python
# Locking server with Forking - Chapter 20 - lockingserver.py
# NOTE: lastaccess.txt will be overwritten!

import socket, traceback, os, sys, fcntl, time

def getlastaccess(fd, ip):
    """Given a file descriptor and an IP, finds the date of last access
    from that IP in the file and returns it. Returns None if there was
    never an access from that IP."""
    # ... (implementation of getlastaccess function)
```

```

# Acquire a shared lock.  We don't care if others are reading the file
# right now, but they shouldn't be writing it.
fcntl.flock(fd, fcntl.LOCK_SH)

try:
    # Start at the beginning of the file
    fd.seek(0)

    for line in fd.readlines():
        fileip, accesstime = line.strip().split("|")
        if fileip == ip:
            # Got a match -- return it
            return accesstime
    return None
finally:
    # Make sure the lock is released no matter what
    fcntl.flock(fd, fcntl.LOCK_UN)

def writelastaccess(fd, ip):
    """Update file noting new last access time for the given IP."""

    # Acquire an exclusive lock.  Nobody else can modify the file
    # while it's being used here.
    fcntl.flock(fd, fcntl.LOCK_EX)
    records = []

    try:
        # Read the existing records, *except* the one for this IP.
        fd.seek(0)
        for line in fd.readlines():
            fileip, accesstime = line.strip().split("|")
            if fileip != ip:
                records.append((fileip, accesstime))

        fd.seek(0)

        # Write them back out, *plus* the one for this IP.
        for fileip, accesstime in records + [(ip, time.asctime())]:
            fd.write("%s|%s\n" % (fileip, accesstime))
        fd.truncate()
    finally:
        # Release the lock no matter what
        fcntl.flock(fd, fcntl.LOCK_UN)

```

```

def reap():
    """Collect any waiting child processes."""
    while 1:
        try:
            result = os.waitpid(-1, os.WNOHANG)
            if not result[0]: break
        except:
            break
        print "Reaped child process %d" % result[0]

host = ''                                # Bind to all interfaces
port = 51423

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind((host, port))
s.listen(1)
fd = open("lastaccess.txt", "w+")

while 1:
    try:
        clientsock, clientaddr = s.accept()
    except KeyboardInterrupt:
        raise
    except:
        traceback.print_exc()
        continue

    # Clean up old children.
    reap()

    # Fork a process for this connection.
    pid = os.fork()

    if pid:
        # This is the parent process. Close the child's socket
        # and return to the top of the loop.
        clientsock.close()
        continue
    else:
        # From here on, this is the child.
        s.close()                      # Close the parent's socket

```

```

# Process the connection

try:
    print "Got connection from %s, servicing with PID %d" % \
        (clientsock.getpeername(), os.getpid())
    ip = clientsock.getpeername()[0]
    clientsock.sendall("Welcome, %s.\n" % ip)
    last = getlastaccess(fd, ip)
    if last:
        clientsock.sendall("I last saw you at %s.\n" % last)
    else:
        clientsock.sendall("I've never seen you before.\n")

    writelastaccess(fd, ip)
    clientsock.sendall("I have noted your connection at %s.\n" % \
        getlastaccess(fd, ip))

except (KeyboardInterrupt, SystemExit):
    raise
except:
    traceback.print_exc()

# Close the connection

try:
    clientsock.close()
except KeyboardInterrupt:
    raise
except:
    traceback.print_exc()

# Done handling the connection. Child process *must* terminate
# and not go back to the top of the loop.
sys.exit(0)

```

This is a fairly basic server. It simply notes the last time a connection was received from a given IP and notes that in a file. The algorithm used to do that is rather inefficient and vulnerable to *race conditions*—situations in which the outcome depends on which process happens to get to the data first.

To combat that, it uses `fcntl.flock()` to restrict access to the file. The `getlastaccess()` function starts out by calling `fcntl.flock(fd, fcntl.LOCK_SH)`. This requests a shared lock on the file. Any number of processes can hold a shared lock as long as no process holds an exclusive lock. That's fine for this function, because it's only reading. It's OK if other processes are reading at the same time, but you don't want to be reading while someone else is writing.

If another process tries to acquire a lock while this process holds it, the other process will stall at the `flock()` call until the lock can be acquired. Therefore, this is a *blocking* call because execution is blocked until a lock is acquired.

At the end of `getlastaccess()`, `flock()` is called again, this time with an argument of `LOCK_UN`, which means “unlock” and effectively releases the lock held. It's *vital* that all acquired locks must be released. Failure to do so can result in *deadlock*, where processes are waiting on each other. The only time a lock is automatically released for you is when your process terminates.

TIP Notice that the unlocking occurs in a `finally` clause. This means that whether an exception was caught or not, the `unlock` command is always run. A common error is to fail to use `try...finally` around locks. Unless you use `try...finally`, an unexpected exception can cause the `unlock` command to be skipped, resulting in deadlock.

The `writelastaccess()` function uses a pattern similar to `getlastaccess()`, except that it acquires an exclusive lock with `LOCK_EX`. When a process holds an exclusive lock, it guarantees that no other process can have a lock of any type on the file. That's what you want here, since you want to lock out all the other readers as well as other instances of `writelastaccess()`.

After the lock is acquired, `writelastaccess()` loads the file from disk, then writes it back out with the new information. You may be wondering why I didn't first acquire a shared lock for reading, followed by an exclusive lock for writing. The answer is that this would introduce a race condition. If I used that approach, then between the time the lock for reading is released and the lock for writing is acquired, another process could have written out data. My process would not know about this data (having just read the file premodification), and the change would be lost. That's why it's important to use a single lock for this entire function.

Let's look at what this program does when it's run. You can just use `./lockingserver.py` to start it. Then, you can telnet to the server. Here's an example of a client-side session:

```
$ telnet localhost 51423
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^>'.
Welcome, 127.0.0.1.
I've never seen you before.
I have noted your connection at Thu Jul  1 06:06:42 2004.
Connection closed by foreign host.

$ telnet localhost 51423
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^>'.
Welcome, 127.0.0.1.
I last saw you at Thu Jul  1 06:06:42 2004.
I have noted your connection at Thu Jul  1 06:08:44 2004.
Connection closed by foreign host.
```

Here, the first time the client connected, the server didn't have a record of it in its `lastaccess.txt` file. It recorded the connection time. For the second connection, the server reports the saved connection time and records the new connection time.

While these connections were occurring, the server was reporting this:

```
$ ./lockingserver.py
Got connection from ('127.0.0.1', 37742), servicing with PID 16848
Reaped child process 16848
Got connection from ('127.0.0.1', 37743), servicing with PID 16850
```

In this particular case, the second child process wasn't yet reaped even though it had terminated. When a third child would connect, it would be reaped.

Error Handling

Strange as it may seem, `os.fork()` can fail. This is rare but does happen. The cause of a failure would be a resource limitation of some kind—the operating system may be out of memory, it may be out of space in its process table, or you may run up against a limit on the maximum number of processes set by an administrator.

There's no good way to deal with this situation. If you don't check for an error, a failure on `os.fork()` will terminate the program. For a client, that's OK, but for a server, it means your server completely dies.

A better way is to kill off just the one connection that caused the problem, and hope that the administrator notices the problem or that the thing causing the problem (a wayward program, for instance) goes away. If so, then when later

clients connect, the fork should succeed. This way, the server process itself need not be restarted.

Remember at the beginning of the chapter I said that `fork()` returns twice. To be more specific, `fork()` either returns twice or raises an exception due to an error. If there's an error, there's no PID returned and execution doesn't fork off—after all, that's why you're getting the exception.

Here's a modified version of the forking echo server that handles problems with `os.fork()`:

```
#!/usr/bin/env python
# Echo Server with Forking and Forking Error Detection - Chapter 20
# errorserver.py

import socket, traceback, os, sys

def reap():
    while 1:
        try:
            result = os.waitpid(-1, os.WNOHANG)
            if not result[0]: break
        except:
            break
        print "Reaped child process %d" % result[0]

host = ''                      # Bind to all interfaces
port = 51423

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind((host, port))
s.listen(1)

while 1:
    try:
        clientsock, clientaddr = s.accept()
    except KeyboardInterrupt:
        raise
    except:
        traceback.print_exc()
        continue

    # Clean up old children.
    reap()
```

```

# Fork a process for this connection.
try:
    pid = os.fork()
except:
    print "BAD THING HAPPENED: fork failed"
    clientsock.close()
    continue

if pid:
    # This is the parent process. Close the child's socket
    # and return to the top of the loop.
    clientsock.close()
    continue
else:
    print "New child", os.getpid()
    # From here on, this is the child.
    s.close()                                # Close the parent's socket

    # Process the connection

    try:
        print "Got connection from", clientsock.getpeername()
        while 1:
            data = clientsock.recv(4096)
            if not len(data):
                break
            clientsock.sendall(data)
        except (KeyboardInterrupt, SystemExit):
            raise
    except:
        traceback.print_exc()

    # Close the connection

    try:
        clientsock.close()
    except KeyboardInterrupt:
        raise
    except:
        traceback.print_exc() . . .

    # Done handling the connection. Child process *must* terminate
    # and not go back to the top of the loop.
    sys.exit(0)

```

You'll notice that this program is mostly the same as the previous example. If `fork()` fails, the server displays an error message, closes the client socket, and returns to the top of the loop. It's important to close that client socket—it will never be used, and this ensures that the client knows not to try communicating over it. More importantly, imagine a scenario in which there was a prolonged time in which `fork()` fails—perhaps the system has run out of memory. It might have to turn away thousands of client requests. If it doesn't close those sockets, each discarded request will continue consuming resources. (In this particular case, Python's garbage collector will likely keep that problem from getting very bad, but it's bad practice to rely upon that behavior).

This program is also notable for what it *doesn't* do. It sends no message whatsoever to the client. The client will simply see a connection reset by peer message if the server cannot fork. This isn't particularly friendly to the client, but consider the alternative. If the server cannot fork, everything it does is taking place in the master process. If it takes a while to communicate with a poorly connected client—say, three minutes—then during that time the server isn't accepting connections at all. A few clients that are attempting to connect when the server can't fork could cause the server to be rendered effectively no better than if it had crashed.

Unfortunately, testing your `os.fork()` error-handling code isn't something that's easily done. Causing `os.fork()` to fail means enforcing administrative restrictions on process counts (not always easily done), or actually causing a system problem.

Summary

Most server programs have a need to handle more than one client at once. There are several methods available to the server designer who wants to accomplish this. The easiest is forking, which is available primarily on Linux and UNIX platforms.

To fork, you call `os.fork()`, which returns twice. That function returns the process ID of the child to the parent, and returns 0 to the child.

When a process terminates, information about its termination remains on the system until its parent calls `wait()` or `waitpid()` on it. Therefore, programs using forking must make sure to call `wait()` or `waitpid()` when a child process terminates. One way to do that is via a signal handler. Alternatively, you could use polling, and periodically check for terminated child processes.

Forking servers usually will use `fork()` to create a new process to handle each incoming connection. It's important for both the parent and child to close any file descriptors that won't be used in that particular process.

If files will be modified, locking is important. Locking prevents data corruption that could occur if multiple processes attempt to modify a file at once, or if one process reads a file while another is writing to it.

The `os.fork()` function can raise an exception if the system cannot perform a fork. Though rare, this exception must be handled to prevent a server crash.

Threading

FORKING, WHICH I DISCUSSED in Chapter 20, is a means of permitting multiple requests to be simultaneously handled. Forking works by creating two completely separate processes out of one. Python also offers another mechanism, known as threading. Threading can be looked at abstractly as having different parts of a single process execute simultaneously. You’re probably wondering why you would ever need an alternative method for managing multiple requests. This is best explained by contemplating a few scenarios that may arise when working with applications that are capable of supporting multiple processes.

In some cases, each connection to a server is totally independent of every other connection. For instance, an FTP server doesn’t need to have any communication between processes that serve clients; each one simply dishes out files or receives uploads. Yet in other cases, that’s not quite so true. For instance, a database server may need to block clients from accessing certain tables while other clients are updating them. Although there are ways to communicate between processes that use `fork()`, threading often makes the use of such methods unnecessary. With threads, you really have only one instance of your program running—it’s just running multiple times. That means that if you change a global variable in one thread, all the other threads will see that change instantly. That’s because the global variable—and indeed, all variables—is shared between all the threads of the program. With a forked program, each process gets its own copy of the variables, so changing a variable in one process has no impact on other processes.

However, this is somewhat of a mixed blessing. While communication between threads is easier than communication between processes, that’s not always good. It also means that the thread serving one client could accidentally mess up the thread serving a different client. Extra care must be taken to ensure that threads don’t trample each other. Such problems are often difficult to detect and debug.

Throughout this chapter, you’ll learn how to take advantage of the easy communication provided with threading and how to make sure this mixed blessing doesn’t turn out to be problematic. The chapter begins with an introduction to threading in Python, and then provides solutions for several common threading issues. Next, you’ll see an example of multithreaded servers. Finally, the chapter concludes with a discussion on multithreaded network clients.

TIP Here's a quick terminology tip: Traditional programs that do not explicitly use threading are said to have only one thread and are called *single-threaded*. Programs that use threading are said to be *multithreaded*. Using more than one thread in your program is called *multithreading*.

Threading in Python

Python exposes two modules that can be used for multithreading: `thread` and `threading`. The `thread` module implements the low-level interface to threaded programming, and `threading` provides a higher-level view. Most new Python programs will use `threading` since it automates some tasks that would otherwise need to be done manually.

Multithreading is available on most platforms that Python supports, though certain versions of UNIX may not support it (or may not support it by default). All of the most common Python platforms, including Java (via Jython), support threading.

Here's an example of threading in Python. This example simply starts a thread and displays some messages to illustrate multithreading, as follows:

```
#!/usr/bin/env python
# First thread example - Chapter 21 - firstthread.py
import threading, time

def sleepandprint():
    time.sleep(1)
    print "Hello from both of us."

def threadcode():
    stdout.write("Hello from the new thread. My name is %s\n" %
                threading.currentThread().getName())
    sleepandprint()

    print "Before starting a new thread, my name is", \
          threading.currentThread().getName()

# Create new thread.
t = threading.Thread(target = threadcode, name = "ChildThread")

# This thread won't keep the program from terminating.
t.setDaemon(1)
```

```

# Start the new thread.
t.start()
stdout.write("Hello from the main thread. My name is %s\n" %
             threading.currentThread().getName())
sleepandprint()

# Wait for the child thread to exit.
t.join()

```

The program begins by creating a new `Thread` object. The `target` parameter to the constructor points to the code to run once the thread starts. The `name` parameter is optional and simply sets a value that you can retrieve later via `getName()`. In this case, it is used to set the text that will later be displayed. The original thread that your program starts with is always named `MainThread`.

One question to consider: Exactly what constitutes termination of the application when multiple threads are concerned? By default, the application will not terminate until all threads have terminated. Usually, when writing network code, it's preferable to have all threads die when the main (control) thread dies. If you call `setDaemon(1)` on a thread, Python pretends that the thread is already dead when considering whether or not to shut everything down. This program calls `setDaemon(1)` for the thread it creates, so the entire application will terminate when `MainThread` terminates. Effectively, this is the same behavior you would have seen if `setDaemon()` were never called.

Finally, the new thread is started with the call to `start()`. Because of the earlier target setting, the new thread calls the `threadcode()` function as soon as it's created. This raises another difference between threads and forking: Threading ensures that the new thread exits when `threadcode()` returns, rather than simply returning and proceeding until an exit instruction appears somewhere as is the case with forking.

At the end of the program, there's a call to `join()`. This call isn't required in general (unlike the need to `wait()` with forked programs). But in this case, it avoids a race condition. Since the new thread is set to daemon mode, if the parent happens to exit before the child has had a chance to print out its message, then the child will be immediately terminated. By calling `join()`, the parent's execution is blocked until the child thread has terminated.

If you run this example, you'll see output like this:

```

$ ./firstthread.py
Before starting a new thread, my name is MainThread
Hello from the main thread. My name is MainThread
Hello from the new thread. My name is ChildThread
Hello from both of us.
Hello from both of us.

```

Using Shared Variables

You'll recall my earlier statement that variables in multithreaded programs are shared between all threads. With that in mind, can you predict the output of the following program?

```
#!/usr/bin/env python
# Threading with variables - Chapter 21 - vars.py
import threading, time

a = 50
b = 50
c = 50
d = 50

def printvars():

    print "a =", a
    print "b =", b
    print "c =", c
    print "d =", d

def threadcode():
    global a, b, c, d
    a += 50
    b = b + 50
    c = 100
    d = "Hello"
    print "[ChildThread] Values of variables in child thread:"
    printvars()

print "[MainThread] Values of variables before child thread:"
printvars()

# Create new thread.
t = threading.Thread(target = threadcode, name = "ChildThread")

# This thread won't keep the program from terminating.
t.setDaemon(1)

# Start the new thread.
t.start()
```

```
# Wait for the child thread to exit.
t.join()

print "[MainThread] Values of variables after child thread:"
printvars()
```

The program begins by setting four variables to 50. It displays the values, then creates a thread. That thread modifies each variable in a slightly different way, outputs the values, and terminates. The main thread then resumes control after the `join()` and prints out the values again. Note that the main thread never makes any changes to the values. Here's what the output looks like:

```
$ ./vars.py
[MainThread] Values of variables before child thread:
a = 50
b = 50
c = 50
d = 50
[ChildThread] Values of variables in child thread:
a = 100
b = 100
c = 100
d = Hello
[MainThread] Values of variables after child thread:
a = 100
b = 100
c = 100
d = Hello
```

You can see every one of those changes in the main thread, because the memory is shared between the two threads. This illustrates the basic method of communication between threads: setting variables. However, as you'll see in the next section, things aren't always that easy.

Being Thread-Safe

Though all this sounds nice, there's a potential down side: race conditions. A race condition occurs whenever the result of a calculation is different depending on the way in which the operating system schedules time. If, in the previous example, two different threads were running the code to add 50 to b at once, the result could be as follows:

- 150 if one thread ran before the other, and each was able to add 50.
- 100 if both threads attempted to perform the calculation simultaneously. In this case, both threads would simultaneously retrieve the current value of b (50), compute the new value by adding 50, and write the new value to b.

In the previous example you don't have to worry about `a += 50`; it will always be 150. That is because the `+=` operation on integers is said to be *atomic*; the system guarantees that the operation will finish before others begin in any thread. However, my advice is to play it safe and not rely on atomic guarantees; it's tough to remember which operations are atomic and which aren't.

To combat the problem of race conditions, *locking* is frequently used. In Chapter 20, I discussed `flock()`'s role in locking files. Locking can also be used arbitrarily—not necessarily connected to any particular file or other system object.

Python's `threading` module provides a `Lock` object. This object can be used to synchronize access to code. The `Lock` object exposes two methods: `acquire()` and `release()`. The `acquire()` method is responsible for acquiring a lock. If no thread is presently holding the lock, the `acquire` method notes the interest in the lock and returns immediately. Otherwise, it waits until the lock is released. In either case, once `acquire()` returns, the thread that called it holds the lock.

The `release()` method releases a lock. If any threads are waiting on the lock (stalled at `acquire()`), *one* of them will be awakened when `release()` is called. That is, `acquire()` in one thread will return.

Here's an example of using locks. This program starts up several threads and uses a lock to protect a global variable.

```
#!/usr/bin/env python
# Threading with locks - Chapter 21 - locks.py
import threading, time

# Initialize a simple variable
b = 50

# And a lock object
l = threading.Lock()

def threadcode():
    """This is run in the created threads"""
    global b
    print "Thread %s invoked" % threading.currentThread().getName()
```

```
# Acquire the lock (will not return until a lock is acquired)
l.acquire()
try:
    print "Thread %s running" % threading.currentThread().getName()
    time.sleep(1)
    b = b + 50
    print "Thread %s set b to %d" % (threading.currentThread().getName(),
                                      b)
finally:
    l.release()

print "Value of b at start of program:", b

childthreads = []

for i in range(1, 5):
    # Create new thread.
    t = threading.Thread(target = threadcode, name = "Thread-%d" % i)

    # This thread won't keep the program from terminating.
    t.setDaemon(1)

    # Start the new thread.
    t.start()
    childthreads.append(t)

for t in childthreads:
    # Wait for the child thread to exit.
    t.join()

print "New value of b:", b
```

This program creates four new threads. Each thread will display a message that says it exists, acquire a lock, delay for one second, update the value of `b`, release the lock, and then terminate. Putting the lock release in the `finally` clause is good practice and guarantees that it will be released even if an exception is raised. Here's a sample invocation:

```
$ ./locks.py
Value of b at start of program: 50
Thread Thread-1 invoked
Thread Thread-1 running
Thread Thread-2 invoked
Thread Thread-3 invoked
Thread Thread-4 invoked
Thread Thread-1 set b to 100
Thread Thread-2 running
Thread Thread-2 set b to 150
Thread Thread-3 running
Thread Thread-3 set b to 200
Thread Thread-4 running
Thread Thread-4 set b to 250
New value of b: 250
```

Each thread got its own turn to run, and you should notice a one-second delay between the “running” messages.

Managing Access to Shared and Scarce Resources

Sometimes there are certain resources that several threads must access. There may be more than one instance of the resource available, so a simple Lock will not do. One scenario might involve server thread pools. Once a server starts, it will create a number of threads. These are worker threads that are responsible for processing clients. In this scenario, the scarce resource is client connections. The threads will wait for the main thread to receive a connection, process it, and then restart before waiting for another connection. If no threads are available to process something, the server should just add it to a queue.

A synchronization object called a *semaphore* is useful in this situation. Semaphores are designed to manage access to limited resources. Like a Lock, a Semaphore has an `acquire()` and a `release()` method. But the mechanics are different. A semaphore has an internal counter that (by default) starts at one. Each time `release()` is called, that counter is incremented. Each time `acquire()` is called, the counter is decremented. If `acquire()` is called when the counter is zero, it doesn’t return until the counter is equal to or greater than one (that is, it doesn’t return until someone else calls `release()`). Here’s a simple example of semaphores. This example provides a function `numbergen()` that simulates a limited resource of numbers (this could be thought of as client connections to a server). Other threads consume those numbers and act on them, as shown here:

```
#!/usr/bin/env python
# Threading with semaphores - Chapter 21 - sem.py
import threading, time, random

def numbergen(sem, queue, qlock):
    while 1:
        time.sleep(2)          # Simulate a complex I/O load
        if random.randint(0, 1):
            # Generate something half the time.
            value = random.randint(0, 100)
            qlock.acquire()
            try:
                queue.append(value)
            finally:
                qlock.release()
            print "Placed %d on the queue." % value

        sem.release()

def numbercalc(sem, queue, qlock):
    while 1:
        sem.acquire()
        qlock.acquire()
        try:
            value = queue.pop(0)
        finally:
            qlock.release()
        print "%s: Got %d from the queue." % \
              (threading.currentThread().getName(), value)
        newvalue = value * 2

        time.sleep(3)          # Simulate a complex calculation

childthreads = []

sem = threading.Semaphore(0)
queue = []
qlock = threading.Lock()
# Create the number generator.
t = threading.Thread(target = numbergen, args = [sem, queue, qlock])
t.setDaemon(1)
t.start()
childthreads.append(t)
```

```

# Create the two threads that work with the numbers.
for i in range(1, 3):
    t = threading.Thread(target = numbercalc, args = [sem, queue, qlock])
    t.setDaemon(1)
    t.start()
    childthreads.append(t)

while 1:
    # Sleep forever
    time.sleep(300)

```

This program consists of four threads: the main thread, a number generator thread, and two number processor threads. The main thread takes care of creating all the other threads, then effectively does nothing. The number generator generates a slow, intermittent stream of numbers. The number processor threads take these numbers and process them.

The `Semaphore` object is initially set to zero. Whenever the number generator has another number available, it will place it on the queue (using a `Lock` to make sure that this operation is safe), then signal its availability by calling `release()` on the `Semaphore`. Note that this doesn't guarantee that the item will be immediately processed (though in this case it usually does); it just signals the availability of data to the processor threads.

The processor threads call `acquire()` at the top of their loop. They then lock the queue, retrieve the item off it, and unlock the queue again.

Here's sample output from this program (you'll have to terminate it with Ctrl-C if you run it):

```

$ ./sem.py
Placed 56 on the queue.
Thread-2: Got 56 from the queue.
Placed 62 on the queue.
Thread-3: Got 62 from the queue.
Placed 54 on the queue.
Thread-2: Got 54 from the queue.
Placed 7 on the queue.
Thread-3: Got 7 from the queue.
Placed 77 on the queue.
Thread-2: Got 77 from the queue.
Traceback (most recent call last):
  File "./sem.py", line 54, in ?
    time.sleep(300)
KeyboardInterrupt

```

This particular example is an instance of a more general problem known as the *producer/consumer* problem. A producer/consumer problem consists of a set of threads that are producing objects, and another set of threads that are consuming them. In this example, the producer was the generator thread, and the consumers were the calculator threads. Producer/consumer is often used when different sets of threads use different resources; for instance, some threads may require lots of I/O to load data, while other threads require lots of CPU to process that data. By splitting those tasks out, you can keep more threads busy.

The producer/consumer model provides a good way to look at many different problems. Later in this chapter, you'll see an implementation of producer/consumer with thread pools.

Alternative to Semaphores: Queue

Python provides a module named `Queue` that can also be useful for solving producer/consumer problems. Though it isn't necessarily as flexible as semaphores, the `Queue` module can still solve many different problems and can be easier to use.

Avoiding Deadlock

Deadlock occurs when two or more threads are waiting for resources, but in such a way that it's impossible for their requests to ever be satisfied because they're waiting for each other. The best way to illustrate deadlock is with an example. In this example, you have two variables and two locks. There are two threads involved, both wishing to modify both variables. The first thread acquires both locks, as does the second, but they do so in a different order. See if you can spot the problem.

```
#!/usr/bin/env python
# Deadlock - Chapter 21 - deadlock.py
import threading, time

a = 5
alock = threading.Lock()
b = 5
block = threading.Lock()
```

```
def thread1calc():
    print "Thread1 acquiring lock a"
    alock.acquire()
    time.sleep(5)

    print "Thread1 acquiring lock b"
    block.acquire()
    time.sleep(5)
    a += 5
    b += 5

    print "Thread1 releasing both locks"
    block.release()
    alock.release()

def thread2calc():
    print "Thread2 acquiring lock b"
    block.acquire()
    time.sleep(5)

    print "Thread2 acquiring lock a"
    alock.acquire()
    time.sleep(5)
    a += 10
    b += 10

    print "Thread2 releasing both locks"
    block.release()
    alock.release()

t = threading.Thread(target = thread1calc)
t.setDaemon(1)
t.start()

t = threading.Thread(target = thread2calc)
t.setDaemon(2)
t.start()

while 1:
    # Sleep forever
    time.sleep(300)
```

In this example, Thread1 attempts to acquire locka and then lockb. However, Thread2 simultaneously attempts to acquire lockb and then locka. Deadlock will result. (The calls to sleep() here ensure that it does.)

Thread1 got the lock on locka and Thread2 got the lock on lockb. Now, they turn around. Thread1 tries to get lockb but can't—Thread2 already has it. And Thread2 tries to get locka but can't because Thread1 has it. And neither one of them releases any lock until they've acquired both locks. The program is deadlocked and can only be killed with Ctrl-C. The output looks like this:

```
$ ./deadlock.py
Thread1 acquiring lock a
Thread2 acquiring lock b
Thread1 acquiring lock b
Thread2 acquiring lock a
Traceback (most recent call last):
  File "./deadlock.py", line 50, in ?
    time.sleep(300)
KeyboardInterrupt
```

Deadlock can be very difficult to track down. In this example, if sleep() were not used, deadlock may have only occurred in one out of a hundred executions of the program. There are two simple rules to observe to avoid deadlock:

- First, always obtain locks in a fixed order. In this example, that would mean always obtaining locka before lockb.
- Second, always release locks in the inverse order that they were obtained. So, one would release lockb first, and then locka.

Writing Threaded Servers

One of the classic problems network programmers need to solve is how to write efficient server programs that can process multiple requests simultaneously. Threads provide one convenient way to do that.

Most multithreaded servers use the same architecture: The MainThread is the thread that listens for requests. When a request is received, a new worker thread is created to handle that particular client. The worker thread for that client terminates when the client disconnects.

Here's an example of such a server. This example provides a TCP echo server by modifying the code from Chapter 3 to make it multithreaded, as shown here:

```

#!/usr/bin/env python
# Echo Server with Threading - Chapter 21 - echoserver.py
# Compare to echo server in Chapters 3 and 20

import socket, traceback, os, sys
from threading import *

host = ''                                # Bind to all interfaces
port = 51423

def handlechild(clientsock):
    print "New child", currentThread().getName()
    print "Got connection from", clientsock.getpeername()
    while 1:
        data = clientsock.recv(4096)
        if not len(data):
            break
        clientsock.sendall(data)

    # Close the connection
    clientsock.close()

# Set up the socket.
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind((host, port))
s.listen(1)

while 1:
    try:
        clientsock, clientaddr = s.accept()
    except KeyboardInterrupt:
        raise
    except:
        traceback.print_exc()
        continue

    t = Thread(target = handlechild, args = [clientsock])
    t.setDaemon(1)
    t.start()

```

This program is fairly straightforward. The `handlechild()` function takes care of a given client connection. When a client connects, a new thread is created.

When that thread is started, it calls `handlechild()`, passing the client's socket as an argument.

If you compare this program to the `echoserver.py` example from Chapter 3 or the forking example in Chapter 20, you'll notice that the code that handles all but the `KeyboardInterrupt` exceptions is missing from all the code in the `handlechild()` function. That's possible because threading guarantees that the `MainThread` will be the only one to ever receive UNIX signals—and `KeyboardInterrupt` is generated as a result of a `SIGINT` signal on UNIX. (Windows doesn't use signals like this, so it's not a concern.) Therefore, it's not a concern for the worker threads. Also, the worker threads can just die if an error occurs in this example.

An Exercise: Threaded Chat Server

Since threads make it easy to pass data between each other, they are ideal for servers on which all clients share some sort of state. One example of such a server is a chat server. You could write a simple chat server that extends the echo server concept, thereby transmitting data received by any client to all clients.

You could start with the previous `echoserver.py` example and add this new ability. There are several different ways to go about it, but in any case, you'll want to make sure that the client threads do the transmission, not the main thread.

One way to accomplish that is to have a semaphore and a queue for each client thread. You could maintain a list of these objects, adding and removing entries from the list as threads come and go. When data is to be sent, it's added to the appropriate queues and the semaphore is signaled.

If you want a working example of a chat server with source code, Chapter 22 provides an asynchronous chat-server implementation, which is yet another way to solve the problem.

Using Thread Pools

Although code that follows the previous pattern may work well for many servers, some may have some special needs. One such need is to minimize the performance overhead of creating a new thread. Though that overhead is small, some applications may need to do more initialization work for a new thread—for instance, by connecting to a database server. That could have a serious negative impact on performance.

Another potential problem lies with resource utilization. The previous program will try to handle all incoming requests concurrently. That's fine for most servers. But heavily loaded ones may wish to, for instance, say that only 1,000 threads will exist at any given moment.

One solution to this problem is the use of thread pools. A thread pool design will have each thread servicing only one client at a time, but the thread doesn't die after it finishes servicing a client. Threads in the pool may either be all created up front or may be created as needs dictate.

A program that uses thread pools still serves each client in a separate thread. However, unlike the previous example, the thread doesn't terminate when the client disconnects. Rather, it remains alive, waiting for more connections to service.

A thread pool also typically has an upper limit on the number of threads to use. Clients that attempt to connect once that limit has been reached will usually be turned away with an error. Some servers also use a pool strategy with forking, though it's much more rare because it's more difficult to manage. Apache is such a server.

Thread pool servers typically consist of several components:

- A main listener thread that accepts client connections and dispatches them
- A set of worker threads that process client requests
- A thread management system that handles threads that have died unexpectedly

Here's an implementation of a thread pool echo server. This example maintains a list of busy threads, waiting threads, and a connection queue, and makes sure that threads receive connections appropriately. I'll present this code in pieces, explaining each part of it as you go.

```
#!/usr/bin/env python
# Thread pool - Chapter 21 - threadpool.py

import socket, traceback, os, sys, time
from threading import *

host = ''                                # Bind to all interfaces
port = 51423
MAXTHREADS = 3
lockpool = Lock()
busylist = {}
waitinglist = []
queue = []
sem = Semaphore(0)
```

In the previous code sample, some global variables are defined, including `queue`, which is responsible for holding pending client connections along with two lists for tracking the state of the threads.

```
def handleconnection(clientsock):
    """Handle an incoming connection."""
    lockpool.acquire()
    print "Received new client connection."
    try:
        if len(waitinglist) == 0 and (activeCount() - 1) >= MAXTHREADS:
            # Too many connections. Just close it and exit.
            clientsock.close()
            return
        if len(waitinglist) == 0:
            startthread()

        queue.append(clientsock)
        sem.release()
    finally:
        lockpool.release()
```

The first defined function is `handleconnection()`. It's called by the MainThread's main loop, `listen()`, when a new connection arrives. First, `handleconnection()` acquires the `lockpool` lock. Then, it checks to see if the system is already maxed out. If so, it just closes the client socket and returns. Next, it determines whether all threads are busy. If so, a new thread is created.

Then, the client socket is added to the queue, and the semaphore is released—signaling a processor thread that a new connection is available. Finally, the pool lock is released, as follows:

```
def startthread():
    # Called by handleconnection when a new thread is needed.
    # Note: lockpool is already acquired when this function is called.
    print "Starting new client processor thread"
    t = Thread(target = threadworker)
    t.setDaemon(1)
    t.start()
```

The `startthread()` function contains code that is very similar to thread code that you've already seen in earlier examples. Its job is merely to start a new thread, as shown here:

```

def threadworker():
    global waitinglist, lockpool, busylist
    time.sleep(1) # Simulate expensive startup
    name = currentThread().getName()
    try:
        lockpool.acquire()
        try:
            waitinglist[name] = 1
        finally:
            lockpool.release()

            processclients()
    finally:
        # Clean up if the thread is dying for some reason.
        # Can't lock here -- we may already hold the lock, but it's OK
        print "** WARNING** Thread %s died" % name
        if name in waitinglist:
            del waitinglist[name]
        if name in busylist:
            del busylist[name]

        # Start a replacement thread.
        startthread()

```

The `threadworker()` is the first function called when a new thread is created. It has two main tasks: 1) initializing the `waitinglist`, and 2) handling threads that are dying. Notice that within the `try` block, it calls `processclients()`—that's the function that does all the real work. In this program, since new threads aren't necessarily created when an existing one dies, it's important to handle threads that are about to die for whatever reason (exception, and so on). The `finally` clause does just that. Whenever the thread is about to die, the `finally` clause gets control. It cleans up the data structures (removing references to the almost-perished thread), starts up a new thread, and then dies.

```

def processclients():
    """Main loop of client-processing threads."""
    global sem, queue, waitinglist, busylist, lockpool
    name = currentThread().getName()
    while 1:
        sem.acquire()
        lockpool.acquire()

```

```

try:
    clientsock = queue.pop(0)
    del waitinglist[name]
    busylist[name] = 1
finally:
    lockpool.release()

try:
    print "[%s] Got connection from %s" % \
          (name, clientsock.getpeername())
    clientsock.sendall("Greetings. You are being serviced by %s.\n" %\
                       name)
while 1:
    data = clientsock.recv(4096)
    if data.startswith('DIE'):
        sys.exit(0)
    if not len(data):
        break
    clientsock.sendall(data)
except (KeyboardInterrupt, SystemExit):
    raise
except:
    traceback.print_exc()

# Close the connection

try:
    clientsock.close()
except KeyboardInterrupt:
    raise
except:
    traceback.print_exc()

lockpool.acquire()
try:
    del busylist[name]
    waitinglist[name] = 1
finally:
    lockpool.release()

```

In `processclients()`, you can see a loop similar to the other echo server examples. It starts out by calling `acquire()` on the semaphore. When that returns, it knows that there's a client connection available to process, so it grabs the lock, obtains the connection, updates the data structures, and releases the connection.

Then it processes the connection (and has an extra feature to help you test the thread-exiting scenario: If you send it the string DIE, it will do just that.) Finally, after the connection is closed, `processclients()` once again acquires `lockpool` and updates the data structures.

NOTE There's actually a bug in the previous code sample. It assumes that the text DIE will be sent in a single packet by your telnet client. That may not always happen, and thus the code may not be triggered. For information about maintaining read buffers, which would solve this problem, see Chapter 22.

```
def listener():
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    s.bind((host, port))
    s.listen(1)

    while 1:
        try:
            clientsock, clientaddr = s.accept()
        except KeyboardInterrupt:
            raise
        except:
            traceback.print_exc()
            continue

        handleconnection(clientsock)
```

The `listener()` function runs in `MainThread` and is responsible for receiving connections from clients. It simply does that and hands them off to `handleconnection()`.

And now, here's the last line of the example, which simply starts up the main listening loop:

```
listener()
```

I suggest running this code on your own machine (it's long, so it would be best to download it rather than typing it in). To run it, just run `./threadpool.py`. Try some of the following experiments:

- Notice that it will take one second for the server to respond the first time you connect. That's the simulated cost, using `sleep()`, of creating a new thread. In real life, the overhead will often be unnoticeable, but this way, you can feel the pool in action. If you close the connection and then connect again, there's no delay.
- You'll also experience that delay the first time you have any number of simultaneous connections, but not after that.
- If you try to open a fourth simultaneous connection, the server will close it immediately because it enforces a maximum connection limit.
- If you send a `DIE` string, the client connection will freeze, but the server will start up a new thread. (A production server would want to close the client socket in this situation.)

Writing Threaded Clients

Threading is sometimes also used for clients. One of the most common applications of threading for clients is to separate time-critical user interface code from slow network access. For instance, a user may be frustrated that a menu takes 20 seconds to display because the program is waiting to receive a set of packets from the network.

Other clients may wish to carry out several network activities at once. For instance, some FTP clients are capable of downloading several files simultaneously. Most web browsers download several items at once.

Here's a sample multithreaded client. It includes some extra calls to `sleep()` to illustrate how it handles client connections and simultaneously provides a "spinner" on the screen.

```
#!/usr/bin/env python
# Threaded Client - Chapter 21 - threadclient.py

import socket, sys, time
from threading import *

host = sys.argv[1]
textport = sys.argv[2]
filename = sys.argv[3]
cv = Condition()
spinners = '|/-\\''
spinpos = 0
equeue = []

def fwrite(buf):
    sys.stdout.write(buf)
    sys.stdout.flush()

def spin():
    global spinpos
    fwrite(spinners[spinpos] + "\b")
    spinpos += 1
    if spinpos >= len(spinners):
        spinpos = 0

def uithread():
    while 1:
        cv.acquire()
        while not len(equeue):
            cv.wait(0.15)
            spin()

        msg = equeue.pop(0)
        cv.release()
        if msg == 'QUIT':
            # Terminate the UI thread
            fwrite("\n")
            sys.exit(0)
        fwrite(" \n %s\r" % msg)
```

```
def msg(message):  
  
    cv.acquire()  
    equeue.append(message)  
    cv.notify()  
    cv.release()  
  
t = Thread(target = uithread)  
t.setDaemon(1)  
t.start()  
  
try:  
    msg('Creating socket object')  
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)  
except socket.error, e:  
    print "Strange error creating socket: %s" % e  
    sys.exit(1)  
  
# Try parsing it as a numeric port number.  
  
try:  
    port = int(textport)  
except ValueError:  
    # That didn't work. Look it up instead.  
    try:  
        port = socket.getservbyname(textport, 'tcp')  
    except socket.error, e:  
        print "Couldn't find your port: %s" % e  
        sys.exit(1)  
  
msg('Connecting to %s:%d' % (host, port))  
time.sleep(5)  
try:  
    s.connect((host, port))  
except socket.gaierror, e:  
    print "Address-related error connecting to server: %s" % e  
    sys.exit(1)  
except socket.error, e:  
    print "Connection error: %s" % e  
    sys.exit(1)
```

```

msg('Sending query')
time.sleep(5)
try:
    s.sendall("GET %s HTTP/1.0\r\n\r\n" % filename)
except socket.error, e:
    print "Error sending data: %s" % e
    sys.exit(1)

msg('Shutting down socket')
time.sleep(3)
try:
    s.shutdown(1)
except socket.error, e:
    print "Error sending data (detected by shutdown): %s" % e
    sys.exit(1)

msg('Receiving data')
count = 0
while 1:
    try:
        buf = s.recv(2048)
    except socket.error, e:
        print "Error receiving data: %s" % e
        sys.exit(1)
    if not len(buf):
        break
    count += len(buf)

msg("Received %d bytes" % count)
msg("QUIT")
t.join()

```

This program is a simple client with the addition of a “spinner”—a simple bit of text that appears to rotate. The spinner rotates while time-consuming actions are taking place, even though those actions block the main thread. This is possible by shoving the spinner into a separate user interface thread.

The `fwrite()` and `spin()` functions are simply utilities for the user interface. The `uithread()` function runs the user interface and makes use of a new threading object: `Condition`. In this program, the user interface is implemented as a producer/consumer—the main thread is the producer of messages for the user, and the interface thread consumes and displays them.

The `Condition` object has some interesting properties, and an underlying lock. The `uithread()` function—the consumer—first acquires the lock. Then it

enters a `while` loop, looping until something is present in the queue. Each time through the loop, it calls `wait()`. That function releases the lock and blocks until another thread calls `notify()`. But it always reacquires the lock before returning, thus providing safe access to the queue for free.

In this case, you pass `0.15` to `wait()`. This means that `wait()` should give up after $15/100$ of a second and just return anyway. Every time `wait()` returns, `spin()` is called. Thus, you effectively rotate the spinner every $15/100$ of a second. If there's actually something available on the queue, you retrieve the value (storing it in `msg`), then release the lock. The item is processed (displayed) and the loop repeats.

The `msg()` function is the producer side of the equation. It starts by acquiring the lock. Then it appends the message to the queue, signals the other thread that a message is there (by calling `notify()`), and finally releases the lock.

The remainder of the program looks fairly normal. Calls to `print` are replaced by calls to `msg()`. But other than that, the rest of the program need not even be aware that a separate thread is running (except for the shutdown procedure at the end).

If you run this program, you'll see output like this:

```
$ ./threadclient.py www.google.com 80 /
```

```
Creating socket object
Connecting to www.google.com:80
Sending query
Shutting down socket
- Receiving data
Received 3010 bytes
```

Though, of course, while the program is actually running, there will be a moving spinner on the left side.

Summary

Threading is one way of supporting multiple connections in a server. Like forking, it permits multiple pieces of code to be executed at once. Unlike forking, threads all have the same address space, so a change made in one thread will affect all others.

Most Python applications will use the `threading` module to create and work with threads. Threading requires careful attention to synchronization issues. The `threading` module provides objects to help out: `Lock`, when used properly, allows only one thread to access a piece of code at a time; `Semaphore` helps manage queues that are shared between threads; and `Condition` helps threads to signal each other when an event of interest occurs.

Threading can also be used for clients to permit other tasks to be carried out while the thread is communicating over the network.

In the next chapter, an alternative to forking and threading (asynchronous I/O) is covered. Unlike forking and threading, asynchronous I/O doesn't involve multiple pieces of code that are executing simultaneously.

CHAPTER 22

Asynchronous Communication

IN CHAPTERS 20 AND 21, I introduced methods of handling multiple connections at once through forking and threading. Both methods involve having the operating system execute multiple code paths simultaneously, though each code path itself is more or less the same as the way a single-socket application would work.

There's a different option available. Instead of running several processes (or threads) at once, one process could be used. This one process would watch over the various connections, switching between them and servicing each one as necessary. This is known as *asynchronous* communication. The traditional method, used everywhere else in this book, is *synchronous* communication in which I/O is handled immediately and directly.

To implement asynchronous communication, some new features are needed. One of them is a way to handle network data without stopping everything. With conventional methods, a call to, say, `read()` will not return until data is received off the network. In this case, that's bad since the process is unable to handle anything else until that one `read()` call returns. Sockets can be put into *nonblocking* mode. In nonblocking mode, if an action cannot be carried out immediately, the call will immediately return a special error code. Processing can then continue. Running around constantly trying to send or receive data from sockets that aren't ready is rather inefficient. It's better to have the operating system tell you when sockets are ready for action. In fact, there are two calls designed to do just that: `select()` and `poll()`. To use one of these functions, you first tell the system about a set of sockets that you're interested in. The call will block until one or more of the sockets are ready for you. You can then find out which sockets are ready, process them, and resume waiting. The `poll()` call tends to be preferred on modern systems, so that is the one you'll see used in this chapter.

Asynchronous I/O on Windows

Python on Windows has some undocumented differences from other platforms when dealing with asynchronous I/O. The examples using `poll()` may not function properly on Windows platforms, but the Twisted examples in this chapter will. If you'll be working with Windows, I recommend using Twisted for asynchronous I/O. The Twisted authors have taken the steps necessary to make most Twisted programs work on both Windows and other platforms.

Deciding Whether or Not to Use Asynchronous Communication

Of course, there are trade-offs involved when you use asynchronous communication. One important characteristic of all asynchronous code is that anything that blocks for any period of time must be eliminated. Servers that perform complex calculations or time-consuming operations (for instance, database servers) generally cannot be purely asynchronous. On the other hand, asynchronous communication imposes very little overhead on new connections. This makes it well suited for servers that process many connections requiring little server-side processing. Web and FTP servers both happen to fit that requirement.

In some ways, writing an asynchronous server can be more complex than writing a forking or a threading server. You have to maintain more state information yourself, rather than let the operating system do so for you. And calls such as `sendall()` must be avoided altogether.

On the other hand, since your server is entirely contained within a single process, there are no locking, deadlock, or synchronization issues to consider—ever. Those are frequently the most difficult to track down among the issues facing forking and threading server authors, so that alone could be a big win.

In terms of libraries that ship with Python itself, few are equipped to work asynchronously. There are two modules: `asyncore` and `asynchat` that help you write asynchronous problems in Python. However, the Twisted project provides more such libraries—and they are, in fact, more numerous than the standard Python libraries for servers.

Another important thing to consider is how much state information must be kept. An FTP server, for instance, keeps little state information; it might store the correct working directory, the file being transferred, and its position in that file. A server such as a game system may have a lot more to store.

Using Asynchronous Communication

Let's take the simple echo server example from Chapter 3 and modify it to work as an asynchronous program capable of handling several connections at once. To do that, you need to add several things, the most notable of which is a way to keep track of state.

In the echo server, keeping track of state means tracking two things: the set of clients that you're interested in, and the data that is to be written to each one. With forking or threading, there's no code that explicitly handles this. Each process or thread handles a specific connection, and goes away when the connection goes away. Moreover, those processes and threads can use normal (blocking) I/O, and maintain no long-lived buffer. They simply use `sendall()` and block until the data has all been sent.

Here, data can be sent in smaller chunks, and you can't block until it's all sent. In fact, you can't block *at all!* Therefore, it's stored in a buffer. The call to `send()` will return the number of bytes that were actually sent without blocking, and that number of bytes is subtracted from the buffer when sent. When new data arrives, it's added to the end of the buffer. In this way, the asynchronous echo server actually gains a new feature that's absent from any of the other servers: It can effectively transmit and receive simultaneously. Here's the code for an echo server. This program uses a class to maintain information about what is going on for each connection. Here's the code, and a detailed explanation follows:

```
#!/usr/bin/env python
# Asynchronous Echo Server - Chapter 22 - echoserver.py
# Compare to echo server in Chapter 3

import socket, traceback, os, sys, select

class stateclass:
    stdmask = select.POLLERR | select.POLLHUP | select.POLLNVAL

    def __init__(self, mastersock):
        """Initialize the state class"""
        self.p = select.poll()
        self.mastersock = mastersock
        self.watchread(mastersock)
        self.buffers = {}
        self.sockets = {mastersock.fileno(): mastersock}

    def fd2socket(self, fd):
        """Return a socket, given a file descriptor"""
        return self.sockets[fd]
```

```

def watchread(self, fd):
    """Note interest in reading"""
    self.p.register(fd, select.POLLIN | self.stdmask)

def watchwrite(self, fd):
    """Note interest in writing"""
    self.p.register(fd, select.POLLOUT | self.stdmask)

def watchboth(self, fd):
    """Note interest in reading and writing"""
    self.p.register(fd, select.POLLIN | select.POLLOUT | self.stdmask)

def dontwatch(self, fd):
    """Don't watch anything about this fd"""
    self.p.unregister(fd)

def newconn(self, sock):
    """Process a new connection"""
    fd = sock.fileno()

    # Start out watching both since there will be an outgoing message
    self.watchboth(fd)

    # Put a greeting message into the buffer
    self.buffers[fd] = "Welcome to the echoserver, %s\n" % \
        str(sock.getpeername())
    self.sockets[fd] = sock

def readevent(self, fd):
    """Called when data is ready to read"""
    try:
        # Read the data and append it to the write buffer.
        self.buffers[fd] += self.fd2socket(fd).recv(4096)
    except:
        self.closeout(fd)

    self.watchboth(fd)

def writeevent(self, fd):
    """Called when data is ready to write."""
    if not len(self.buffers[fd]):
        # No data to send? Take it out of the write list and return.
        self.watchread(fd)
        return

```

```

try:
    byteswritten = self.fd2socket(fd).send(self.buffers[fd])
except:
    self.closeout(fd)

# Delete the text sent from the buffer
self.buffers[fd] = self.buffers[fd][byteswritten:]

# If the buffer is empty, we don't care about writing in the future.
if not len(self.buffers[fd]):
    self.watchread(fd)

def errorevent(self, fd):
    """Called when an error occurs"""
    self.closeout(fd)

def closeout(self, fd):
    """Closes out a connection and removes it from data structures"""
    self.dontwatch(fd)
    try:
        self.fd2socket(fd).close()
    except:
        pass

    del self.buffers[fd]
    del self.sockets[fd]

def loop(self):
    """Main loop for the program"""
    while 1:
        result = self.p.poll()
        for fd, event in result:
            if fd == self.mastersock.fileno() and event == select.POLLIN:
                # Mastersock events mean a new client connection.
                # Accept it, configure it, and pass it over to newconn()
                try:
                    newsock, addr = self.fd2socket(fd).accept()
                    newsock.setblocking(0)
                    print "Got connection from", newsock.getpeername()
                    self.newconn(newsock)
                except:
                    pass

```

```

        elif event == select.POLLIN:
            self.readevent(fd)
        elif event == select.POLLOUT:
            self.writeevent(fd)
        else:
            self.error(event)

host = ''                                # Bind to all interfaces
port = 51423

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind((host, port))
s.listen(1)
s.setblocking(0)

state = stateclass(s)

state.loop()

```

Let's look at this code. The majority of the action occurs within the class `stateclass`. The `__init__()` method takes a master socket and stores it off. It also initializes two data structures. The `buffers` structure will be used to store the buffer for each client, and the `sockets` structure stores the socket for each client. Both are indexed by a file descriptor number, which is a number assigned by the operating system that is unique to each socket. Finally, a `poll()` object is created and saved.

The `fd2socket()` method is a simple helper method. It receives a file descriptor (as returned by `poll()`) and yields a socket object.

The four methods with "watch" in their names are helpers. They register interest (or disinterest) in certain file descriptors with the `poll` object. There are several different events that a program may be interested in: reading, writing, and various error conditions. These four methods always mark interest in error conditions, and one or both of reading and writing.

The `newconn()` method is called when a new connection arrives. It notes an interest in the socket, sets an initial buffer value (which will be used as a greeting), and updates the data structures.

When data arrives, `readevent()` is called. It reads data from the socket, adds it to the buffer, and makes sure that both a reading and writing interest is noted for the socket. This is done because as soon as data has been received there will be data to echo back out.

The `writeevent()` method is called when you have data to write out and the system can guarantee that `send()` will accept a chunk of data and then return immediately. Pending data is sent, and the amount of data actually sent is

removed from the buffer. If the buffer is left empty, the interest in the socket is changed to read only, so that the next time through the loop, no write will be attempted.

When an error occurs, `errorevent()` is called. It simply calls `closeout()`. That function removes all interest in the socket from the poll object. Then, it closes the socket itself and removes it from the data structures.

The function where the program spends most of its time is `loop()`. The program will call `self.p.poll()` at the top of the loop. This is the one call in the program that is supposed to block. It will not return until something of interest happens with one of the sockets. This behavior ensures that the server doesn't use CPU resources unless something is actually going on with the network.

When `poll()` returns, it returns a list of tuples. Each tuple in that list corresponds to a connection on which something of interest happened. Our task, therefore, is to examine each tuple and decide what to do.

The tuple consists of a file descriptor of a socket and an event. The code first checks to see if this particular tuple corresponds to a new client connection—a situation signified by a read event on the master socket. If that happens, it will process the new connection the same way as any other server—by calling `accept()`. Then it will pass it over to `newconn()`. The remaining events get passed to `readevent()`, `writeevent()`, and `errorevent()`.

There's a lot of code there. It may be helpful to trace through the program and follow the sequence of events.

When the program starts, it creates the master socket just like other servers do. It then creates the `stateclass` object, passing in the master socket. Then it calls `state.loop()` and enters the main loop. The program will invoke `p.poll()` with a single interesting socket: the master socket. It will delay at `p.poll()` until a client connects.

The first client finally connects. The `p.poll()` call returns a single tuple corresponding to the master socket. The server calls `accept()`, retrieving the new client socket. It then calls `self.newconn()`, which adds the socket to the data structures. It initializes the buffer with the greeting message, and tells the poll object to notify the program whenever data comes in or when data can go out. Then the loop reverts to the top.

When data can be sent, `p.poll()` returns again—this time, the socket for the client is returned, with a event type indicating that writing is now permissible. The `writeevent()` method gets called. It will attempt to `send()` the entire buffer. This won't necessarily happen; `send()` will transmit the amount of data that can be sent without blocking. It then returns the number of bytes it actually sent. The server removes those bytes from the start of the buffer, and then returns. (If the buffer was emptied, the poll object is told to not notify the server about future write events.)

The server will sit at `p.poll()` until someone either connects or the server is told that it can now read some data from the client. If data comes in from the client, `readevent()` is called. It will read up to the first 4,096 bytes of data from the client, adding it to the end of the write buffer. If there's more data than that, no problem; when `p.poll()` is called next, it will again indicate that data is available for reading. After the data is received and added to the buffer, `watchboth()` is called—since you know there's data in the buffer, you're ready to write it at any time.

That's a lot to consider. Let's look at the sequence of events when a client connects.

First, the call to `p.poll()` returns with the master socket among the list of sockets ready for reading. The server calls `accept()` and calls `newconn()` with the new client socket. Then `newconn()` initializes the data structures and places the welcome message in the buffer for the client. Finally, it says to watch for both reading and writing from the client before returning. Control will then return to `p.poll()`.

When the client is ready to receive data, `p.poll()` will return again with the client's socket among the list of sockets ready for writing. The server will transmit some data, and if the entire contents of the buffer were sent, it will remove the client from the list of sockets for writing. Control returns to the loop once again.

When the client sends something to the server, `p.poll()` will return and indicate that there's data to be read from the client. The `readevent()` method is called. It receives the data, adds it to the end of the buffer, and makes sure that the server is ready to write data back to the client. When the client is ready to receive data, it's sent in the same manner as the initial greeting.

When the client closes the connection, the server is notified as an error and calls `errorevent()`, which closes the socket on the server side and removes the client from the data structures.

To run this server, you can simply use `./echoserver.py`. You can connect to localhost on port 51423. You'll see a greeting and, like the other echo server examples in this book, the server will return to you anything you send to it.

Advanced Server-Side Use

Many asynchronous servers will actually have two buffers per client—one for incoming commands and one for outgoing data. This allows the server to account for incoming commands that aren't contained entirely within a single packet. The following is an example of a very primitive chat system. Data received is relayed to all clients connected, but only when the text `SEND` is received, as follows:

```

#!/usr/bin/env python
# Asynchronous Chat Server - Chapter 22 - chatserver.py

import socket, traceback, os, sys, select

class stateclass:
    stdmask = select.POLLERR | select.POLLHUP | select.POLLNVAL

    def __init__(self, mastersock):
        self.p = select.poll()
        self.mastersock = mastersock
        self.watchread(mastersock)
        self.readbuffers = {}
        self.writebuffers = {}
        self.sockets = {mastersock.fileno(): mastersock}

    def fd2socket(self, fd):
        return self.sockets[fd]

    def watchread(self, fd):
        self.p.register(fd, select.POLLIN | self.stdmask)

    def watchwrite(self, fd):
        self.p.register(fd, select.POLLOUT | self.stdmask)

    def watchboth(self, fd):
        self.p.register(fd, select.POLLIN | select.POLLOUT | self.stdmask)

    def dontwatch(self, fd):
        self.p.unregister(fd)

    def sendtoall(self, text, originfd):
        for line in text.split("\n"):
            line = line.strip()
            transmittext = str(self.fd2socket(originfd).getpeername()) + \
                ":" + line + "\n"
            for fd in self.writebuffers.keys():
                self.writebuffers[fd] += transmittext
            self.watchboth(fd)

```

```

def newconn(self, sock):
    fd = sock.fileno()
    self.watchboth(fd)
    self.writebuffers[fd] = "Welcome to the chat server, %s\n" % \
        str(sock.getpeername())
    self.readbuffers[fd] = ""
    self.sockets[fd] = sock

def readevent(self, fd):
    try:
        # Read the data and append it to the write buffer.
        self.readbuffers[fd] += self.fd2socket(fd).recv(4096)
    except:
        self.closeout(fd)

    parts = self.readbuffers[fd].split("SEND")
    if len(parts) < 2:
        # No SEND command received
        return
    elif parts[-1] == '':
        # Nothing follows the SEND command; send what we have and
        # ignore the rest.
        self.readbuffers[fd] = ""
        sendlist = parts[:-1]
    else:
        # The last element has data for which a SEND has not yet been
        # seen; push it onto the buffer and process the rest.
        self.readbuffers[fd] = parts[-1]
        sendlist = parts[:-1]

    for item in sendlist:
        self.sendtoall(item.strip(), fd)

def writeevent(self, fd):
    if not len(self.writebuffers[fd]):
        # No data to send? Take it out of the write list and return.
        self.watchread(fd)
        return

    try:
        byteswritten = self.fd2socket(fd).send(self.writebuffers[fd])
    except:
        self.closeout(fd)

```

```

        self.writebuffers[fd] = self.writebuffers[fd][byteswritten:]

    if not len(self.writebuffers[fd]):
        self.watchread(fd)

def errorevent(self, fd):
    self.closeout(fd)

def closeout(self, fd):
    self.dontwatch(fd)
    try:
        self.fd2socket(fd).close()
    except:
        pass

del self.writebuffers[fd]
del self.sockets[fd]

def loop(self):
    while 1:
        result = self.p.poll()
        for fd, event in result:
            if fd == self.mastersock.fileno() and event == select.POLLIN:
                try:
                    newsock, addr = self.fd2socket(fd).accept()
                    newsock.setblocking(0)
                    print "Got connection from", newsock.getpeername()
                    self.newconn(newsock)
                except:
                    pass
            elif event == select.POLLIN:
                self.readevent(fd)
            elif event == select.POLLOUT:
                self.writeevent(fd)
            else:
                self.errorevent(fd)

host = ''                                # Bind to all interfaces
port = 51423

```

```

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind((host, port))
s.listen(1)
s.setblocking(0)

state = stateclass(s)
state.loop()

```

The framework of this program looks similar to the previous example. However, note the addition of a read buffer and the code that processes it. That code is of particular interest in that it handles three distinct cases of input that might occur: no end-of-command (SEND) received; one or more complete commands terminated by SEND, and one or more complete commands followed by an incomplete command. With asynchronous I/O, the normal commands that are used, for instance to read a full line of input are unavailable. You must therefore buffer input yourself, and be prepared to receive partial lines or multiple lines all at once.

You can run this program by running `./chatserver.py`. To test it, open several telnet clients directed at port 51423. In one of them, your session might look like this:

```

$ telnet localhost 51423
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^>'.
Welcome to the chat server, ('127.0.0.1', 48633)
Hello.
Testing.
SEND
('127.0.0.1', 48633): Hello.
('127.0.0.1', 48633): Testing.
How are you?
SEND
('127.0.0.1', 48633): How are you?

```

Monitoring Multiple Master Sockets

In the previous example, one master socket was used; the server listened on a socket. It only ever handles a single socket that's listening. It's also possible to use a single-tasking server to listen to many different ports. In fact, the standard UNIX “superserver” `inetd` does exactly that.

`inetd` listens on many different ports. When a connection arrives, it will start up the program that is supposed to handle that connection. In this way, a single process can handle dozens of different sockets. On a system where these different processes aren't under constant, heavy use, this is a win; one process listening instead of dozens of different ones.

One way to implement an `inetd`-like server is to use `poll()` to watch a whole set of master sockets. When a connection is received, it's moved to a known file descriptor and handed off to the program that will actually handle it. Chapter 3 contains some examples of programs that use `inetd`.

In actual fact, an `inetd`-like server will be something of a hybrid; it will use `poll()` to monitor the master sockets, but `fork()` to pass them on to the handlers. This will expose an important security consideration for programs like this. Here's the example `inetd` server. This example will not work on Windows due to its use of forking.

```
#!/usr/bin/env python
# Asynchronous Inetd-like Server - Chapter 22 - inetd.py

import socket, traceback, os, sys, select

class stateclass:
    def __init__(self):
        self.p = select.poll()
        self.mastersocks = {}
        self.commands = {}

    def fd2socket(self, fd):
        return self.mastersocks[fd]

    def addmastersock(self, sockobj, command):
        self.mastersocks[sockobj.fileno()] = sockobj
        self.commands[sockobj.fileno()] = command
        self.watchread(sockobj)

    def watchread(self, fd):
        self.p.register(fd, select.POLLIN)

    def dontwatch(self, fd):
        self.p.unregister(fd)
```

```
def newconn(self, newsock, command):
    try:
        pid = os.fork()
    except:
        try:
            newsock.close()
        except:
            pass
        return

    if pid:
        # Parent process
        newsock.close()
        return

    # Child process from here on
    # First, close all the master sockets.
    for sock in self.mastersocks.values():
        sock.close()

    # Next, copy the socket's file descriptor to standard input (0),
    # standard output (1), and standard error (2).

    fd = newsock.fileno()
    os.dup2(fd, 0)
    os.dup2(fd, 1)
    os.dup2(fd, 2)

    # Finally, call the command.
    program = command.split(' ')[0]
    args = command.split(' ')[1:]

    try:
        os.execvp(program, [program] + args)
    except:
        sys.exit(1)
```

```

def loop(self):
    while 1:
        result = self.p.poll()
        for fd, event in result:
            print "Received a child connection"
            try:
                newsock, addr = self.fd2socket(fd).accept()
                self.newconn(newsock, self.commands[fd])
            except:
                pass

host = ''                      # Bind to all interfaces

state = stateclass()
config = open("inetd.txt")
for line in config:
    line = line.strip()
    port, command = line.split(":", 1)
    port = int(port)

    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    s.bind((host, port))
    s.listen(1)
    s.setblocking(0)
    state.addmastersock(s, command)

config.close()
state.loop()

```

Although this server doesn't have all the features of the standard `inetd` server, it nevertheless does do the same basic task. First, it creates an instance of the `stateclass` class. Then it opens its configuration file, `inetd.txt`, and reads it. Each line gives a TCP port number and a command to run when a client connects to that port. So, for each configuration file line, a new socket object is created, bound, configured, and added to the `stateclass` information. Finally, when the configuration file has been completely processed, it's closed, and the main loop is entered.

This loop is simpler than the main loop in the chat server example. The `inetd` loop only has to handle one event—a client connecting. When that happens, the client is passed off to `self.newconn()`, along with the command that will be executed.

The `newconn()` method is where the real action happens. Notice that it starts off by forking. This isn't really standard practice for asynchronous servers, but can be useful, as you see here. (For more details on forking, please refer to Chapter 20.) After the fork, the parent process should go back to processing client connections and the client process will handle this connection.

So, by checking the `pid` value, if you're now in the parent, the new client socket is closed (as is standard practice with forking servers) and the code returns to the loop.

On the child-process side, the first thing it does is close down every single one of the master sockets. In Chapter 20, I mentioned that a child process should close the sockets it won't use so that it won't accidentally communicate on them or cause strange interactions with `close()`. Those reasons still hold. But in this case, there's an added reason to do that: security.

The child process will later be calling an `exec...()` function to execute another program. We might not necessarily trust this other program to be secure and to be able to handle all these master sockets in a secure way. For instance, a malicious program might be able to find the master socket for a particular port and “take over” that port, handling its requests instead of letting the `inetd` server do that. It might be able to record somebody's password—the person may think they're sending it to the real server, but in reality, it's going to a fake one. This problem is known as a *file descriptor leak*. So it's vital to close down all the sockets that the client won't need in this situation.

After doing that, the next thing the client does is call `os.dup2()` three times. You'll recall from Chapter 3 that `inetd` passes the socket to its server program on standard input, standard output, and standard error. Those are file descriptors 0, 1, and 2 on UNIX. The `os.dup2()` call lets you duplicate a socket (or file, or anything else that has a UNIX file descriptor) to a file descriptor with a particular number. We duplicate the client's socket three times so that it's present in standard input, standard output, and standard error.

Finally, the client is executed. The call to `os.execvp()` might have surprising semantics if you're unfamiliar with UNIX. Unless there's an error, it *never* returns. On successful completion, `os.execvp()` (or the other `exec...()` functions) completely replaces the calling process in memory. So the forked child of `inetd.py` ceases to exist. However, the process environment—including its file descriptors—is copied over to the newly executed program. Therefore, the program receives the client socket as it should.

Let's take a look at how this might work. First, here's a sample configuration file that will let a client connect to the server at two different ports to run two different examples from Chapter 3:

```
55100:../03/inetdsocket.py
55101:../03/inetdserver.py
```

This file doesn't use the same format as the system's `/etc/inetd.conf`, but it gets the job done for this example. Now, let's see what happens:

```
$ telnet localhost 55100
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Welcome.
According to our records, you are connected from ('127.0.0.1', 52385).
The local time is Mon Mar  8 10:06:02 2004.
Connection closed by foreign host.

$ telnet localhost 55101
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Welcome.
Please enter a string:
Testing
You entered 7 characters.
Connection closed by foreign host.
```

It worked. By connecting to a different port, a different server process was run. Yet a single process—our example `inetd`-look-alike—handled the connections on both ports.

This “hybrid” technique (using polling and forking in a single program) can have other uses, too. For instance, an asynchronous server may make sense for most of what you do, but certain commands are processor-intensive; executing them in the single-server process may cause undesired blocking. A new process may be forked, or a thread created, to handle that, with the results passed back later.

Using Twisted for Servers

If you think about the chat server example abstractly, you can visualize the program as two main components: the `poll()` loop that takes care of the mechanics of watching for data, and the actual code that receives data dispatched from the loop and processes it. With some work, the main loop could be made generic—the same code could be used to drive various different asynchronous servers.

This is the approach that the Twisted system takes. Twisted provides the core libraries that implement the central loop for the server. It provides base classes

that you override, adding your own functionality where appropriate. They often call this the “Don’t call us, we’ll call you” design. Your application never explicitly receives data anywhere. Rather, when data arrives, a class method is called. You process the data, possibly queue data for transmission, then return. Twisted, of course, handles sending the data out as appropriate, using an internal buffer similar to the chat server. This forms the core of Twisted.

TIP Another description of some Twisted basics can be found in Chapter 12. Many IMAP examples in that chapter use Twisted for client-side communication.

The Twisted system, though, takes this several steps farther. It offers some generic helper classes, such as the `LineOnlyReceiver`. In the chat server, you had to write code that deals with receiving partial or multiple “lines” at once. `LineOnlyReceiver` will take care of this automatically. Twisted also provides server-side modules for some common network protocols, though this chapter doesn’t go into detail on them.

Twisted isn’t part of the standard Python library. You can download it from www.twistedmatrix.com. Your operating system provider may also make available Twisted packages; if so, those packages may also suffice. The examples in this chapter assume you have Twisted 1.1.1 or higher installed.

TIP This Twisted code probably looks complex and hard to follow. However, Twisted is really an elegant system, and with a little experience, it can feel just as easy as more traditional programming.

Here’s an example implementation of a chat server in Twisted. Notice that it doesn’t have to keep track of any state.

```
#!/usr/bin/env python
# Asynchronous Chat Server with Twisted - Chapter 22
# twistedchatserver.py
# Twisted 1.1.1 or above required for this example
#     -- download from www.twistedmatrix.com
```

```

from twisted.internet.protocol import Factory
from twisted.protocols.basic import LineOnlyReceiver
from twisted.internet import reactor

class Chat(LineOnlyReceiver):
    def lineReceived(self, data):
        self.factory.sendAll("%s: %s" % (self.getId(), data))

    def getId(self):
        return str(self.transport.getPeer())

    def connectionMade(self):
        print "New connection from", self.getId()
        self.transport.write("Welcome to the chat server, %s\n" % self.getId())
        self.factory.addClient(self)

    def connectionLost(self, reason):
        self.factory.delClient(self)

class ChatFactory(Factory):
    protocol = Chat

    def __init__(self):
        self.clients = []

    def addClient(self, newclient):
        self.clients.append(newclient)

    def delClient(self, client):
        self.clients.remove(client)

    def sendAll(self, message):
        for proto in self.clients:
            proto.transport.write(message + "\n")

reactor.listenTCP(51423, ChatFactory())
reactor.run()

```

The first thing that might strike you about this example is that it's a lot shorter than the first chat server example. In fact, it's about one-third the size. This difference occurs because Twisted provided a lot of the basics that the first example had to provide for itself. Under the hood, both programs do essentially the same thing.

The Twisted program is based around two main classes: Chat and ChatFactory. The ChatFactory is somewhat of a “master” class that’s instantiated only once in the entire application. The Chat class is instantiated once for each client that connects; its data structures are unique to each client. This structure makes it easy to both maintain data that’s global for the entire system and data that’s unique for each client. Since there’s a separate class instance for each client, there’s no master dictionary like the first chat server example had.

Both classes derive from Twisted-supplied base classes. The Chat class is a child of Twisted’s LineOnlyReceiver, which is itself a child of Protocol. The Protocol class handles the receiving of data and calls the dataReceived() method when data arrives. The idea is that you can subclass Protocol and provide your own implementation of dataReceived() that processes the information.

In fact, that’s exactly what LineOnlyReceiver does. Whenever a complete line of data has arrived, LineOnlyReceiver’s dataReceived() method will call the lineReceived() method. Again, this method is designed to be overridden in a subclass. The Chat class does exactly that. Whenever a line is completely received from the client, the Twisted code calls lineReceived(), passing the line to it. That function, in turn, calls self.factory.sendAll(). Twisted automatically stores a reference to the Factory object in the Protocol object (Chat) for you.

Like lineReceived(), the connectionMade() method is called when an event occurs. Twisted calls connectionMade() when a client first connects. In this example, the connectionMade() method sends the greeting and adds itself to the data structure by calling self.factory.addClient(). Similarly, when the client disconnects, connectionLost() is called, which asks the factory object to remove its information about the client.

The ChatFactory class is fairly simple. Its `__init__()` method simply creates an empty list of clients. The `addClient()` and `delClient()` methods straightforwardly add or remove a client from the list of clients. The `sendAll()` method takes a message as an argument and iterates over the list of all client objects, calling `transport.write()` on each one of them.

This `write()` method is not the same as `socket.send()`, `socket.sendall()`, or the `write()` method of Python file-like objects. Twisted’s `write()` method will store the data to be sent in a buffer, and then returns immediately. Behind the scenes, it asks the internal poll object to notify it when the client can receive data, and will thus send out data until its buffer is exhausted. Internally, `write()` uses the same sort of logic that the example before did. But it does it all behind the scenes; you don’t have to worry about buffering and managing clients that are ready (or not) to receive data.

The program ends with two lines of code that set up and run the server. By calling `listenTCP()`, the listening socket is set up. The `reactor.run()` method is the main loop of the program; it’s not generally set to ever return, and instead must be terminated by something such as a signal with Ctrl-C or Ctrl-Break.

Summary

Asynchronous communication provides a way to handle multiple connections at once. Unlike forking and threading, asynchronous communication doesn't actually have the server executing different code simultaneously. Rather, it uses non-blocking I/O and polling to service each client when it becomes ready.

Asynchronous I/O is centered on a main loop that waits for events to arrive. In this chapter, that loop uses `poll()` to look for events on file descriptors. When an event occurs—such as data that's available to read, or ready-to-write data—the program can see what happened and take the appropriate action. The `poll()` function is designed to watch many sockets at once.

It's also possible to monitor multiple master sockets. In this chapter, an example `inetd` implementation demonstrated doing that.

The Twisted framework provides many tools for writing asynchronous servers. It can save you a great deal of effort. In this chapter, the Twisted implementation of a chat server was approximately one-third the size of the implementation that was done from scratch.

Index

Symbols

& character, 383

> character, 383

< character, 383

* folder pattern, 238

+= operator, 448

< character, 383

<affiliation> child, 153

<broadcast> address (DNS), 95

== self-processing tag, 137

> character, 383

A

A DNS records, 76

AAAA DNS records, 76

accept(), 14, 40, 41, 42, 43, 429, 432, 475, 476

account FTP authentication token, 276

acquire(), 448, 450, 452, 461

addBoth(), 235

addCallback(), 227, 230

addClient(), 488

addFlags(), 252

addGETdata(), 120

AddHandler, 401

address_family variable, 352

adns Python library, 65

advanced network operations, 87–110

binding to specific addresses, 102–3

half-open sockets, 87–88

overview, 87

timeouts, 89–90

transmitting strings, 90–92

leading size indicator, 92

unique end-of-string identifiers, 91–92

understanding network byte order, 93–94

using broadcast data, 95–97

using event notification with poll() or select(), 104–9

working with IPv6, 97–102

handling family preferences, 100–102

resolving addresses, 98–100

AF_INET protocol, 20, 99, 101

AFP (Apple File Sharing) protocol, 20

Alias directive, 396, 399

allow_none variable, 360

allow_reuse_address variable, 351

alternative multipart (MIME), 180

alternative subtype (MIME), 186

answers attribute (PyDNS), 78

ANY query (DNS), 82

Apache API, 397

apache2ctl configtest command, 398

apache2ctl restart command, 395

apachectl configtest command, 398

apachectl restart command, 395

apache.HTTP_FORBIDDEN (403) status code, 402

apache.HTTP_MOVED_PERMANENTLY (301) status code, 402

apache.HTTP_MOVED_TEMPORARILY (302) status code, 402

apache.HTTP_NOT_FOUND (404) status code, 402

apache.HTTP_OK (200) status code, 402

- apache.HTTP_UNAUTHORIZED (401)
 - status code, 402
 - apache.SERVER_RETURN command, 402
 - APOP, 212, 213
 - apop(), 214
 - append(), 270
 - appendChild(), 156
 - application/octet-stream type (MIME), 183
 - arguments parameter, inetd, 47
 - arraysize attribute, 311, 312
 - as_string(), 182
 - ASCII characters, 92, 93
 - ASCII files, downloading with FTP, 278–79
 - asynchat module, 470
 - asynchronous communication, 469–89
 - advanced server-side use, 476–80
 - monitoring multiple master sockets, 480–85
 - overview, 469–70
 - using, 471–76
 - using Twisted for servers, 485–88
 - whether to use, 470
 - asyncore module, 470
 - atomic operation, 448
 - attachment(), 183
 - attacks, 323
 - authentication
 - FTP, 276–77
 - POP (Post Office Protocol), 212–14
 - SMTP, 208–9
 - Web client access, 115–18
 - “Authentication failed” error, 209
-
- B**
 - Base-64 encoding, 181
 - base64 module, 92
 - basehttp.py file, 342
 - BaseHTTPRequestHandler class, 341–42
 - BaseHTTPServer, 341–48
 - handling multiple requests simultaneously, 346–48
 - handling requests for specific documents, 343–46
 - overview, 341–43
 - BEGIN CERTIFICATE block, 332
 - Binary(), 317
 - binary files, downloading with FTP, 279–81
 - binary word, 93
 - bind(), 39, 43
 - bind package (Linux), 68
 - blocking call, 437
 - body of e-mail messages, 169–70
 - broadcast data, 95–97
 - BSD UNIX, 19, 46
 - buffers structure, 474
 - buflen option, 37
 - build_opener(), 117
 - built-in SSL, 326–30
 - byte order, network, 93–94
-
- C**
 - C connect(), 21
 - C language, 10, 11, 12, 25, 53, 55, 93, 159
 - callback function, 225
 - catch statement, 121
 - Cc Header (MIME), 171
 - certfiles.crt file, 332
 - Certificate Authorities (CAs), 325
 - certificates, server. *See* server certificates
 - CGI (Common Gateway Interface), 369–92
 - escaping special characters, 383–85
 - getting input, 375–83
 - extra URL components, 375–78
 - GET method, 378–80
 - overview, 375
 - POST method, 380–83
 - handling multiple inputs per field, 385–86

overview, 369
retrieving environment information, 373–75
scripts, 349–50, 365, 367, 393, 397, 405, 415
setting up, 370
understanding CGI, 370–72
uploading files, 386–88
using cookies, 388–92

CGI handler, 407, 415
cgi interface, 387
cgi library, 380
cgi module, 371, 373, 383, 385, 386, 412–13
cgi-bin directory, 349, 370, 373
cgi.escape(), 383, 413
cgi.FieldStorage(), 380
CGIHTTPServer, 349–50, 370
cgilib.escape(), 385
cgitb module, 372
CGIXMLRPCRequestHandler, 365–67
character references, translating, 132–33
character set, ASCII, 181
Chat class, 488
chat server exercise, threaded, 457
Chatfactory class, 488
checksum, 5
chldhandler(), 427
CLASSPATH variable, 300
cleanse(), 139
client_address variable, 352
clients, database. *See* database clients
clients, network. *See* network clients
client/server networking, 3–18

- Ethernet, 9
- networking in Python, 9–16
 - high-level interface, 15–16
 - low-level interface, 10–15
- overview, 3
- physical transports, 9

TCP basics, 3–6

- addressing, 4
- reliability, 4–5
- routing, 5
- security, 6

user datagram protocol, 7–8

using client/server model, 6–7

close(), 19, 42, 43, 88, 218, 422, 484

closeout(), 475

CN (common name) attribute, 335

CNAME record, 76, 79, 82

cverified variable, 335

codecs module, 194

column names, 314

command channel, 276

commands

- executing, 301–2
- repeating, 305–10
 - executemany(), 307–10
 - parameter styles, 305–7

Comment objects, 150

commit(), 302–3, 304, 305, 310

Common Gateway Interface. *See* CGI (Common Gateway Interface)

common name (CN) attribute, 335

Common Object Request Broker Architecture (CORBA), 159

comp.lang.python newsgroup, 399

composing e-mail. *See* e-mail composition and decoding

compromised server, 324

Condition object, 466

configure script, 395

connect(), 21, 26, 32, 33, 35, 52, 54, 68

connecting, 297–301

- Jython zxJDBC, 299–301
- MySQL, 299
- POP (Post Office Protocol), 212–14
- PostgreSQL, 298

- Connection object, 331
- connection reset by peer message, 441
- `connectionLost()`, 488
- `connectionMade()`, 227, 231, 488
- content types (MIME), 181
- Content-Disposition header (MIME), 183, 184, 186
- Content-Length header (MIME), 123
- Content-type header, 372
- Content-Type line (MIME), 184
- Context object, 331
- continue statement, 41, 42
- conversation debugging (SMTP), 199–202
- Cookie module, 389
- `cookie.output()`, 392
- cookies, CGI, 388–92
- Coordinated Universal Time (UTC), 178
- `copy()`, 270
- CORBA (Common Object Request Broker Architecture), 159
- `create()`, 270
- CR-LF character, 268
- cross-site scripting attack, 383
- cursor object, 301–2
- `cwd()`, 278, 290
- D**
 - daemon log file, 60
 - data channel, 276
 - data command, 202
 - data item, 386
 - database clients, 295–320
 - connecting, 297–301
 - Jython zxJDBC, 299–301
 - MySQL, 299
 - PostgreSQL, 298
 - executing commands, 301–2
 - overview, 295
- reading metadata, 313–16
- counting rows, 313–14
- retrieving data as dictionaries, 315–16
- repeating commands, 305–10
 - `executemany()`, 307–10
 - parameter styles, 305–7
- retrieving data, 310–13
 - using `fetchall()`, 310–11
 - using `fetchmany()`, 311–12
 - using `fetchone()`, 312–13
- SQL and networking, 295
- SQL in python, 296–97
- transactions, 302–5
 - hiding changes until finished, 303–5
 - performance implications of transactions, 303
 - using data types, 317–19
- `datareceived()`, 474, 488
- `datasock.close()`, 283
- `Date()`, 317
- Date header (MIME), 173, 178, 180
- Date string, 179
- `DateFromTicks()`, 317
- date-ID headers, 174–75
- `db` parameter, 299
- DB-API specification, 296–97, 300, 305, 306, 310, 317
- `dbh` (database handle), 297
- Debian GNU/Linux, 77
- `decode()`, 194
- decoding e-mail. *See* e-mail composition and decoding
- `def gethostname(ipaddr)`, 73–74
- Deferred from list(), 238
- Deferred object, 227, 228; 230, 231, 234
- DeferredList object, 248, 259, 260
- `delClient()`, 488
- `dele()`, 218

DELE command, 218
`delete()`, 270, 293
\Deleted flag, 252, 255
`deletemessages()`, 255
deleting
 folders, 293
 messages, 218–21, 252–55
deletion attacks, 323
description variable, 313
dgram socket type, 49
dgram type, 47
DHCP, 77
`dictfetchall()`, 316
`dictfetchone()`, 316
dictionaries, retrieving data as, 315–16
`dir()`, 284, 285, 288, 290
directories. *See* folders
Directory section, 399
`DirEntry` class, 288, 290, 292
`DirScanner` class, 288, 292
dispatching requests (`mod_python`), 402–4
`dispCookie()`, 391
`displayinfo()`, 260
`dlist` list, 248
DNS (Domain Name System), 21, 65–85
 `DiscoverNameServers()`, 77
 making DNS queries, 65–66
 overview, 65
 `Request()`, 77
 using operating system lookup services, 66–75
 obtaining information about your environment, 74–75
 performing basic lookups, 66–70
 performing reverse lookups, 70–74
 using PyDNS for advanced lookups, 76–85
 DNS records, 76–77
installing PyDNS, 77
querying specific name servers, 79–81
resolving lookup results, 82–85
simple PyDNS queries, 77–79
DNS module, 77
`dnslook`, 65
`dnspython`, 65, 77
`do_...()`, 342
`do_GET()`, 345, 348
DocBook, 145, 148
docstring, 358
document type definition (DTD), 145, 148
DocXMLRPCServer, 364–65
DOM
 full parsing with, 151–54
 generating documents with, 154–57
 type reference, 157–58
domain attribute, 389
Domain Name System. *See* DNS (Domain Name System)
`dothefork()`, 422
`downloadaddir()`, 292
`downloadfile()`, 292
`downloadinfo()`, 259, 260
downloading
 ASCII files (FTP), 278–79
 binary files (FTP), 279–81
 messages (IMAP), 243–49
 messages (POP), 216–18
 recursively (FTP), 290–93
DSO (Dynamic Shared Objects) support, 395
DTD (document type definition), 145
`dup()`, 19
`dup2()`, 19, 484
Dynamic Shared Objects (DSO) support, 395

E

echo client, 62, 63
echo server, 61–62, 63
ehlo(), 204, 205, 209
EHLO command, 205
EHLO, getting information from, 202–4
EHLO method, 210
Element objects, 150
e-mail. *See also* IMAP (Internet Message Access Protocol)
e-mail composition and decoding, 169–95
 MIME
 composing alternatives, 185–87
 composing attachments, 182–84
 parsing, 190–95
 understanding, 180–81
 nested multipart, composing, 188–90
 non-English headers, composing,
 187–88
 overview, 169
 traditional messages
 composing, 173–76
 parsing, 176–80
 understanding, 169–73
email module, 169, 176, 190, 192
email package, 173
email.Header module, 193
email.message_from_file(), 177
email_Utils module, 178
email_Utils.formatdate(), 174
email_Utils.make_msgid(), 174
encode(), 194
END CERTIFICATE block, 332
end_headers(), 342
entries.append(), 286
envelope value, 260
errback notion, 232
error handling
 forking, 438–41
 FTP, 283–84
 network clients, 23–31
 errors with file-like objects, 29–31
 missed errors, 26–28
 socket exceptions, 24–26
 network servers, 41–43
 SMTP, 199–202
 Web client access, 121–25
 connection errors, 121–23
 data errors, 123–25
 XML-RPC, 165
error_all.py example, 125
errorevent(), 475, 476
errorhappened(), 234, 235, 236
escape(), 412
escaping mod_python, 412–13
ESMTP, 202–3, 204
Ethernet, 9
Ethernet LAN, 37
event notification, with poll() or select(),
 104–9
event-based parser, 148
event-based programming, 225
examine(), 239, 240, 245
except clause, 60
exec(), 424
exec...() type functions, 422, 484
execute(), 302, 308, 310
executemany(), 307–10
executing commands, 301–2
execvp(), 484
EXISTS summary item, 240
expunge(), 252, 255

F

facility argument, 57
 factory class, 227
 factory object, 231
 Factory object, 488
 failure(), 363
 Failure object, 234, 235, 236
 fake server (traffic redirection), 324
 fcntl(), 216
 fd2socket(), 474
 f.dir(), 286, 288
 FETCH command, 243
 fetch...(), 310, 315
 fetchall(), 310–11
 fetchBodyStructure(), 256, 260
 fetchFlags(), 250
 fetchmany(), 311–12
 fetchone(), 312–13
 fetchSimplifiedBody(), 256, 259, 260
 fetchSpecific(), 245, 248, 249, 256, 260
 fetchUID('1:*'), 248
 FieldStorage class, 409, 412
 FieldStorage instances, 385
 file attribute, 387
 file descriptor, 19
 file descriptor leak, 484
 File Not Found (404) error, 401
 File Transfer Protocol. *See* FTP (File Transfer Protocol)
 file.cgi script, 388
 File-like objects, 23
 filename attribute, 387
 files
 binary, downloading (FTP), 279–81
 moving (FTP), 294
 renaming (FTP), 294
 uploading (CGI), 386–88
 finally clause, 292, 437, 449, 460

finding messages (IMAP), 262–67
 composing queries, 263–65
 running queries, 265–67
 finishprocessing(), 81, 137, 142
 Flag-related search keywords, 263
 flags option, 49
 flags, reading (IMAP), 250–51, 252
 FLAGS summary item, 240
 flock(), 216, 437, 448
 flush(), 30, 46
 folders
 creating
 FTP, 294
 IMAP, 270
 deleting
 FTP, 293
 IMAP, 270
 examining (IMAP), 239–43
 folder list, scanning (IMAP), 236–39
 moving messages between (IMAP), 270
 scanning (FTP), 284–90
 for loop, 377
 fork(), 54, 88, 425, 438, 439, 441, 481
 forked pool, 424
 forking, 419–42
 error handling, 438–41
 first steps, 424–30
 overview, 424–25
 zombie problem, 425–30
 fork(), 421–24
 duplicated file descriptors, 422–23
 overview, 421–22
 performance, 424
 zombie processes, 423
 locking, 433–38
 overview, 419
 processes, 419–21
 servers, 430–33

ForkingMixIn class, 363
 form data, submitting, 118–21
 with GET, 118–20
 with POST, 120–21
 FORM tag, 383
 format style, 306, 307
 FreeBSD search engine, 118
 from cgi import escape command, 413
 From header (MIME), 173
 fromfd(), 51
 fromtimestamp(), 180
 FTP (File Transfer Protocol), 117, 275–94
 authentication and anonymous FTP, 276–77
 communication channels, 276
 creating directories, 294
 deleting files and directories, 293
 downloading ASCII files, 278–79
 downloading binary files, 279–81
 downloading recursively, 290–93
 handling errors, 283–84
 moving and renaming files, 294
 overview, 275
 scanning directories, 284–90
 discovering information without parsing listings, 288–90
 parsing UNIX directory listings, 286–88
 uploading data, 281–83
 using in Python, 277–78
 FTP protocol, 114
 FTP URL, 125
 ftplib module, 277, 279, 283, 284, 293
 ftplib.all_errors tuple, 283
 ftplib.error_perm exception, 290
 function query type, 81
 fwrite(), 466

G

Gadfly, 297
 GET method, 342, 343, 383
 CGI, 378–80
 and mod_python, 407–10
 submitting form data with, 118–20
 getaddrinfo(), 70, 75, 98–100, 101
 getAttribute(), 153
 getCapabilities(), 227
 getCookie(), 391
 getdate(), 180
 _getdoc(), 345
 getdsn(), 298
 getElementsByTagName(), 153
 getfilename(), 288
 getfirst(), 380
 gethandlerfunc(), 404
 gethostbyname(), 67
 getLastAccess(), 437
 getList(), 380, 385
 getName(), 445
 getpeername(), 51
 getrecordsfromnameserver(), 81
 getruntime(), 367
 getscriptname(), 409
 getservbyname(), 21
 getsockopt(), 37, 38
 getstats(), 363
 gettype(), 288
 geturl(), 115
 Gopher handler, 125
 Gopher Protocol, 10–11, 114
 gopherlib module, 15
 gotcapabilities(), 227
 gotmessage(), 249, 255
 gotmessages(), 245

H

H string format, 93
h_errno C exception, 25
 half-open sockets, 87–88
handle(), 351
handle_request(), 367
handlechild(), 456–57
handleconnection(), 459, 462
handler(), 400, 404, 405
handleuids(), 248, 254
 handling input (mod_python), 405–12
 extra URL components, 405–7
 GET method, 407–10
 overview, 405
 POST method, 410–12
\HasChildren flag, 239
\HasNoChildren flag, 239
HEAD method, 342
Header.decode_header(), 194
 Header-related search keywords, 264
headers (MIME), 169
hel0(), 204
HELO command, 203
hex(), 357
hierquery(), 81
 high-level interface, 15–16
hijacking, session, 323
host command, 68
host parameter, 299
hosts file, 66
.htaccess files, 399
htbin directory, 349
HTML, 118, 187
 HTML and XHTML, parsing, 127–43
 handling unbalanced tags, 133–37
 overview, 127–30
 translating character references, 132–33

translating entities, 130–32
 working example, 137–43
HTML code, 114
HTML file, 370
HTML form, 410
htmlentitydefs class, 132
htmllib, 128
HTMLParser module, 127, 128, 130, 137, 142, 148
HTMLParser's feed() method module, 129
htonl(), 94
HTTP, 15, 39, 113, 115, 116, 117, 122, 388
HTTP authentication, 388
HTTP headers, 125, 389
HTTP protocol, 114
HTTP status code, 401
HTTP_COOKIE environment variable, 389, 391
HTTPBasicAuthHandler handler, 117
HTTPError exception, 122, 123
httplib module, 15
HTTPServer class, 341, 348
human engineering, 324
HUP signal, 48

I

I format, 93
IANA (Internet Assigned Numbers Authority), 7
ident parameter, 57
if statement, 424
if test, 426
IMAP (Internet Message Access Protocol), 223–72
 adding messages, 268–70
 creating and deleting folders, 270
 downloading, 243–49
 entire mailbox, 243–45
 messages individually, 245–49

- examining folders, 239–43
 - message numbers vs. UIDs, 239–40
 - message ranges, 240
- finding messages, 262–67
 - composing queries, 263–65
 - running queries, 265–67
- flagging and deleting messages, 249–55
 - deleting messages, 252–55
 - reading flags, 250–51
 - setting flags, 252
- moving messages between folders, 270
- overview, 223–25
- retrieving message parts, 255–62
 - finding message structures, 256–60
 - retrieving numbered parts, 260–62
- scanning folder list, 236–39
- in Twisted, 225–36
 - error handling, 231–36
 - logging in, 228–31
 - overview, 226–28
- IMAP4Client class, 227, 230
- IMAPFactory object, 227, 231
- imaplib module, 224
- IMAPLogic object, 231, 234
- import cgi command, 413
- IN-ADDR.ARPA extension, 84
- INBOX folder, 239, 243, 245
- index('BODY'), 249
- index.html file, 348
- inetd
 - configuring, 47–48
 - handling errors with, 54–55
 - using socket objects with, 51
 - using UDP with, 51–54
 - when not to use, 55
- inetd loop, 484
- inetd server, 480–81, 483, 484
 - inetd.py file, 484
 - infinite loop, 40–41
 - info(), 125
 - init process, 423
 - __init__(), 230, 231, 234, 260, 290, 488
 - initSyslog(), 60
 - INPUT tag, 388
 - In-Reply-To header, 173
 - INSERT INTO command, 305
 - insertion attacks, 323
 - installing and configuring mod_python, 394–99
 - configuring Apache directories, 396–98
 - fixing configuration problems, 398–99
 - loading the module, 395
 - overview, 394–95
 - int(), 359, 360
 - Internet Assigned Numbers Authority (IANA), 7
 - Internet Message Access Protocol.
 - See* IMAP (Internet Message Access Protocol)
 - interpreter instances (mod_python), 413–14
 - introspection, 355
 - invocationtype parameter, 47
 - IOError exception, 283
 - IP address, 4, 9, 21, 76
 - IPv4, 20, 67, 97–98, 99, 100, 101–2
 - IPv6, 26, 67, 97–102, 352–53
 - address, 76
 - handling family preferences, 100–102
 - queries, 77
 - resolving addresses, 98–100
 - IPX/SPX (NetWare) protocol, 20
 - ISO 8859-1, 187, 188
 - isValid(), 288

J

Java, 159, 300
 JDBC driver conversion layer, 297
`join()`, 445, 447
 Jython interpreter, 297
 Jython zxJDBC, connecting, 299–301

K

key pair, 325
 KeyboardInterrupt exception, 32, 40, 42, 43, 457

L

level parameter, 37
 libxml2 library, 149
 LineOnlyReceiver class, 486, 488
`lineReceived()`, 488
 Lines header, 171
 Linux, 7, 14, 16, 38, 46, 68, 172, 216, 346, 372, 386, 388
 Linux platform, 396, 420
`list()`, 215, 238
`listen()`, 14, 39, 43, 54, 459
`listener()`, 462
`listenTCP()`, 488
`listMethods()`, 360
`list.sort()`, 359
`load_verify_locations()`, 335
 LoadModule line, 395
 local area network (LAN), 95, 97
`locale.getpreferredencoding()`, 194
 localhost server, 198
 Lock object, 448, 450, 452, 466
 LOCK_EX argument, 437
 LOCK_UN argument, 437
 locking, 448
 lockpool lock, 459, 462

`LOG_syslog` priority, 59
`LOG_ALERT` syslog priority, 59
`LOG_AUTH` syslog facility, 58
`LOG_CONS` syslog option, 57
`LOG_CRIT` syslog priority, 59
`LOG_CRON` syslog facility, 58
`LOG_DAEMON` syslog facility, 58
`LOG_DEBUG` syslog priority, 59
`LOG_EMERG` syslog priority, 59
`LOG_ERR` syslog priority, 59
`LOG_INFO` syslog priority, 58, 59
`LOG_KERN` syslog facility, 58
`LOG_LOCALx` syslog facility, 58
`LOG_LPR` syslog facility, 58
`LOG_MAIL` syslog facility, 58
`LOG_NDELAY` syslog option, 57
`LOG_NEWS` syslog facility, 58
`LOG_NOTICE` syslog priority, 59
`LOG_NOWAIT` syslog option, 57
`LOG_PERROR` syslog option, 57
`LOG_PID` syslog option, 57
`LOG_USER` syslog facility, 58
`LOG_UUCP` syslog facility, 58
`logexception()`, 60
`logexception(1)` call, 188
`loggedin()`, 230, 231
 logging module, 55, 56
`login()`, 208, 228, 230, 231, 278
`loginerror()`, 234, 235, 236
`logout()`, 231, 235, 248
`loop()`, 475
 loopback interface, 102
 low-level interface, 10–15

- basic client operation, 10–11
- basic server operation, 13–15
- errors and exceptions, 11–12
- file-like objects, 12–13

M

Mac OS 9, 77
 mail from command, 202
 mailbox information (POP), 215–16
 Maildir specification, 216
 MainThread thread, 445, 455, 457, 459, 462
 make install command, 395
 makefile(), 12–13, 15, 29, 30, 31
 man-in-the-middle (MITM) attacks, 322
 \Marked flag, 238
 Math class, 362
 max-age attribute, 389
 md5 command, 386, 388
 MD5 sum, 386, 388
 md5sum command, 386, 388
 Meerkat service, 162
 Message-ID header, 171, 173, 174–75
 message/rfc822 type, 259
 MessageSet object, 240, 254
 metadata, reading, 313–16

- counting rows, 313–14
- retrieving data as dictionaries, 315–16

 method parameter, 118
 METHOD parameter, 383
 MIME header, 173
 MIME message, 192
 MIME (Multipurpose Internet Mail Extensions)

- composing alternatives, 185–87
- composing attachments, 182–84
- parsing, 190–95
- understanding, 180–81

 MIMEBase generic object, 183
 MIMEMultipart object, 182, 183, 186
 MIMEText module, 173
 MIMEText object, 183
 mimetypes module, 183

minidom

Document object, 156
 parse(), 151
 MITM (man-in-the-middle) attacks, 322
 mkd(), 294
 mktime_tz(), 179
 mod_python, 393–416

- basics of, 399–402
- handler return values, 401–2
- overview, 399–400
- role of PythonHandler, 400–401

 dispatching requests, 402–4
 escaping, 412–13
 handling input, 405–12

- extra URL components, 405–7
- GET method, 407–10
- overview, 405
- POST method, 410–12

 installing and configuring, 394–99

- configuring apache directories, 396–98
- fixing configuration problems, 398–99
- loading the module, 395
- overview, 394–95

 interpreter instances, 413–14
 need for, 393–94
 overview, 393
 prebuilt handlers in, 415
 mod_python mailing list, 399
 mods-available directory, 395
 Morsel object, 389, 391, 392
 moving files (FTP), 294
 Mozilla, 122
 msg(), 467
 mtr program, 5
 multipart messages, 180
 multipart/alternative part, 259

multiple inputs per field, handling (CGI), 385–86
M
 Multipurpose Internet Mail Extensions (MIME), 169
 multi-threaded program, 444
 multithreading, 444
 MX record, 76, 78, 82
 mxTidy, 128, 133
 MySQL, connecting, 299
 MySQLdb, 299
 MySQLdb connect(), 299
 MySQL-Python, 299

N

\n line ending, 268
 name argument, 77
 named style, 307
 NAMEINARGS flag, 49
 netmask, 9
 Network applet, 66
 network clients, 19–34
 handling errors, 23–31
 errors with file-like objects, 29–31
 missed errors, 26–28
 socket exceptions, 24–26
 overview, 19
 sockets
 communicating with, 23
 creating, 20–22
 overview, 19–20
 using user datagram protocol, 31–33
N
 Network File System (NFS), 216
 network operations, advanced. *See also* advanced network operations
 network servers, 35–64
 accepting connections, 40–41
 avoiding deadlock, 60–63
 handling errors, 41–43

inetd
 configuring, 47–48
 handling errors with, 54–55
 using socket objects with, 51
 using UDP with, 51–54
 when not to use, 55
 logging with syslog, 55–60
 overview, 35
 preparing for connections, 35–39
 binding the socket, 39
 creating socket object, 36
 listening for connections, 39
 setting and getting socket options, 36–38
 using user datagram protocol, 43–45
 xinetd, configuring, 48–50
n
 network vulnerabilities
 reducing with SSL, 324–25
 understanding, 322–24
 compromised server, 324
 deletion attacks, 323
 fake server (traffic redirection), 324
 human engineering, 324
 insertion attacks, 323
 overview, 322
 replay attacks, 323
 session hijacking, 323
 sniffing, 322
 newconn(), 474, 475, 476, 484
 newline character, 91
 Next >> button, 382
N
 NFS (Network File System), 216
 nlst(), 284, 288
 Node Types, 158
 \Noinferiors flag, 238
 nonblocking mode, 469
 None object, 318
 non-HTTP protocols, 123, 125
 non-UNIX platforms, 55, 56
 \Noselect flag, 238

Not(), 263
notify(), 467
nowait implementation, 52–53
nowait server, 52
nowait type, 47
NS record, 76, 79, 82
nslookup(), 81
NTEventLogHandler(), 55
ntransfercmd(), 279, 280–81, 282, 284
numbergen(), 450
numeric style, 306

O
ODBC driver conversion layer, 297
ok parameter, 335
openlog(), 56
OpenSSL, 330–31
OpenSSL, verifying server certificates with, 331–38
 obtaining root certificate authority certificates, 332
 overview, 331
 verifying the certificates, 332–38
operating system lookup services, 66–75
 obtaining information about your environment, 74–75
 performing basic lookups, 66–70
 performing reverse lookups, 70–74
opportunistic encryption, 206
Or(), 263
O'Reilly's Meerkat service, 160
osslverify.py file, 338

P
paramstyle variable, 306
parse(), 151
parsedate_tz(), 179
parsing HTML and XHTML. *See* HTML and XHTML, parsing
pass_(), 212
passive mode, 276
passwd parameter, 299
path attribute, 389
path parameter, 47
PATH_INFO environment variable, 374, 375, 377–78, 380
peek, 245
PERMANENTFLAGS summary item, 240
physical transports, 9
pickle module, 159, 361
PID (process ID), 420–21, 421, 484
plain encoding, 181
Point to Point Protocol (PPP), 9
poll(), 104–9, 469, 470, 474, 475, 476, 481, 485
POLLERR option, 106
POLLHUP option, 106
POLLIN option, 106
POLLNVAL option, 106
POLLOUT option, 106
POLLPRI option, 106
POP (Post Office Protocol), 211–22
 connecting and authenticating, 212–14
 deleting messages, 218–21
 downloading messages, 216–18
 obtaining mailbox information, 215–16
 overview, 211
POP3 object, 212
poplib module, 211, 216
poplib.error_proto, 212
port name 3–4, 21
port number, 4
port option, 49
port parameter, 47, 299
POST method, 342, 387
 CGI, 380–83
 and mod_python, 410–12
 submitting form data with, 120–21

- Post Office Protocol. *See* POP (Post Office Protocol)
- PostgreSQL, connecting, 298
- `pow()`, 357, 358
- PPP (Point to Point Protocol), 9
- prebuilt handlers, in mod_python, 415
- `print` statement, 372
- `print_day_quiz()`, 382
- `printf()`, 306
- `printpart()`, 260
- `printqueryresult()`, 267
- `printx509()`, 335
- private key, 325
- process ID (PID), 420–21, 484
- `processclients()`, 460, 461, 462
- producer/consumer problem, 453
- protocol class, 227
- Protocol class, 488
- protocol object, 231
- Protocol object, 488
- protocol option, 49
- `ps` command, 420, 424
- PSP (Python Server Pages) handler, 415
- `psycopg connect()`, 298
- `psycopg` module, 298
- PTR record, 76, 82
- public-key cryptography, 325
- Publisher handler, 404, 415
- `pwd()`, 278
- `pwd` module, 159
- PyDNS, 65
- PyDNS, using for advanced lookups, 76–85
 - DNS records, 76–77
 - installing PyDNS, 77
 - querying specific name servers, 79–81
 - resolving lookup results, 82–85
 - simple PyDNS queries, 77–79
- pyformat style, 307, 308
- pyOpenSSL, 326, 330
- Python 2.1, 301
- Python Database Topic Guide, 296
- Python Server Pages (PSP) handler, 415
- `PythonDebug` line, 399
- `PythonHandler`, 400–401
- `PythonHandler` test line, 399
- `PythonInterpPerDirective` configuration directive, 414
- `PythonInterpPerDirectory` configuration directive, 414
- `PythonInterpreter` configuration directive, 414
- Q**
- `qmark` style, 306, 307
- `qtype` argument, 77
- `Query()`, 263
- `quit()`, 212, 218, 278
- Quoted-printable encoding, 181
- R**
- race conditions, 436, 447
- `rcpt to` command, 202
- `reactor.run()`, 227
- `reactor.stop()`, 227, 231
- `read()`, 19, 23, 122, 123, 281, 327
- `readevent()`, 474, 475, 476
- `readline()`, 23, 91, 281
- `readlines()`, 13
- `reap()`, 429, 432
- reaping, 427
- Received headers, 176
- Received headers (MIME), 171
- \Recent flag, 249, 251
- RECENT summary item, 240
- recursive name server, 65
- `recv()`, 19, 23, 26, 29, 33, 38, 63, 90, 104, 105, 281
- `recvfrom()`, 23, 33, 44, 52, 53

Red Hat, 46
release(), 448, 450, 452
reliable protocol, 5
Remote Method Invocation (RMI), 159
Remote Procedure Call (RPC) server, 159, 355
REMOTE_ADDR environment variable, 374
REMOTE_HOST environment variable, 374
REMOTE_USER environment variable, 388
removeFlags(), 252
rename(), 294
renaming files (FTP), 294
replay attacks, 323
Reply-To header (MIME), 173
repr(), 94
req(), 77
req.path_info file, 405, 407
Request object, 114, 115
request variable, 352
RequestHandler instance, 342
req.write(), 400, 407
resolver libraries, 66
retr(), 216, 217
RETR command, 283
retrbinary(), 279, 283
retrieving data, 310–13
 using fetchall(), 310–11
 using fetchmany(), 311–12
 using fetchone(), 312–13
retrieving message parts (IMAP), 255–62
 finding message structures, 256–60
 retrieving numbered parts, 260–62
retrlines(), 278
RFC2109, 389
RFC3501, 225, 228, 249
RFC86, 33
RFC959, 275
rfile object, 351
rfile variable, 342
rmd(), 293
RMI (Remote Method Invocation), 159
\r\n line ending, 268
rollback(), 303, 305
rowcount attribute, 315
RPC (Remote Procedure Call) server, 159, 355
run(), 488
runquery(), 267
RuntimeError exceptions, 60

S

SampleScanner class, 151
SAX, 154, 157
scanning folders (FTP), 284–90
scanNode(), 151
SCRIPT_NAME environment variable, 377
search(), 262–63, 267
search keywords, 264
secure attribute, 389
Secure Sockets Layer. *See* SSL (Secure Sockets Layer)
secure sockets layer, 205–8
Secure Sockets Layer (SSL), 6
\Seen flag, 245
select(), 104–9, 239, 240, 469
self.data, 137
self.logout(), 235
self.processing flag, 136
self.stopreactor callback, 231, 235
self.taglevels list, 136–37
semaphore, 450
Semaphore object, 452
send(), 12, 19, 23, 29, 30, 63, 104, 471, 474, 475
SEND text, 476, 480
send_header(), 342
send_response(), 342
send_selector(), 15
sendall(), 26, 28, 90, 327, 470, 471

sendAll(), 488
Sender line, 171
sendmail(), 198–99, 203, 207, 209
sendmail command line, 172
sendto(), 23, 33, 44
serve_forever(), 342, 367
server certificates, verifying with OpenSSL, 331–38

- obtaining root certificate authority certificates, 332
- overview, 331
- verifying the certificates, 332–38

server option, 49
server_args option, 49
SERVER_NAME environment variable, 375
SERVER_PORT environment variable, 375
server-side port numbers, 7
service declaration, 48
session hijacking, 323
session token, 388
set_server_... functions, 365
set_verify(), 335
setCookie(), 391, 392
setDaemon(), 445
setFlags(), 252
setsockopt(), 37, 96
setsockopt(2) manpage, 38
settimeout(), 25, 89
SGML, 147
SGML Framework, 128
SGML (Standard Generalized Markup Language), 145
SGML tag, 145
shared variables, and threading, 446–47
s.has_extn(), 205, 209
shutdown(), 26–28, 29, 31, 43, 88
SIGCHLD signal, 423, 427
SIGINT signal, 457
signal.signal(), 428
Simple API for XML (SAX), 148
simple message transport protocol. *See*
 SMTP (simple message transport protocol); SMTP (simple message transport protocol)
Simple Object Access Protocol (SOAP), 159
SimpleCookie object, 391, 392
./simplehttp.py file, 349
SimpleHTTPServer, 348–49
SimpleHTTPServer module, 345
./simple.py file, 357
SimpleXMLRPCServer, 355–68

- basics, 356–58
- CGIXMLRPCRequestHandler, 365–67
- DocXMLRPCServer, 364–65
- exploiting class features, 361–63
- overview, 355–56
- serving functions, 359–60

single-threaded program, 444
sleep(), 455, 463
SMTP (simple message transport protocol), 197–210

- authenticating, 208–9
- error handling and conversation debugging, 199–202
- exchange, 171, 172
- getting information from EHLO, 202–4
- overview, 197
- SMTP library, 197–99
- tips, 209–10
- using secure sockets layer and transport layer security, 205–8

smtplib module, 197, 199, 202, 208, 210
smtplib.SMTP object, 198
smtplib.SMTPException, 199
smtpobj.set_debuglevel(1) call, 199
sniffing, 322, 323
SO_BINDTODEVICE option, 37
SO_BROADCAST option, 37
SO_DONTROUTE option, 37

SO_KEEPALIVE option, 38
SO_OOBINLINE option, 38
SO_REUSEADDR flag, 351
SO_REUSEADDR socket object, 36–37, 38
SO_TYPE option, 38
SOA records SOA = Start of Authority, 76
SOCK_DGRAM protocol, 20, 32, 100
SOCK_STREAM protocol family, 20, 32
SOCK_STREAM socket type, 99, 100
sockaddr data, 67, 68
socket(), 19
socket module, 20, 66, 94, 326
Socket objects, 23
socket_type option, 49
socket(7) manpage, 38
socket.AF_INET socket, 98
socket.AF_INET6 socket, 98
socket.connect(), 67
socket.error, 199
socket.error exception, 25, 26, 104, 123, 283
socket.fromfd(), 51
socket.gaierror, 199
socket.gaierror exception, 11, 25, 26
socket.getaddrinfo(), 67–69
socket.getfqdn(), 74–75
socket.gethostbyname(), 67
socket.gethostname(), 74–75
socket.getservbyname(), 32
socket.herror(), 71, 199
socket.herror exception, 25
socket.makefile(), 351
sockets
 binding, 39
 communicating with, 23
 creating, 20–22
 creating socket object, 36
 overview, 19–20
 setting and getting socket options, 36–38
SocketServer, 341–54
BaseHTTPServer, 341–48
 handling multiple requests simultaneously, 346–48
 handling requests for specific documents, 343–46
 overview, 341–43
CGIHTTPServer, 349–50
implementing new protocols, 350–52
IPv6, 352–53
overview, 341
SimpleHTTPServer, 348–49
socket.SOCK_STREAM protocol type, 69–70
socket.socket(), 10, 14, 51, 67
socket.timeout exception, 25, 89, 90
SOL_SOCKET socket options, 37
sort(), 360
sortlist(), 360
spam scanner, 172
special characters, escaping (CGI), 383–85
spin(), 466, 467
spinner, 466
split(), 215
SQL, 295, 296–97
srvr.register_introspection_functions(), 365
srvr.register_multicall_functions(), 367
s.sendall(), 12
s.setsockopt(socket.SOL_SOCKET,
 socket.SO_BROADCAST, 1) call, 95
SSL (Secure Sockets Layer), 321–38
 overview, 321
 in Python, 326
 reducing vulnerabilities with SSL, 324–25
 understanding network vulnerabilities, 322–24
 compromised server, 324
 deletion attacks, 323
 fake server (traffic redirection), 324

human engineering, 324
 insertion attacks, 323
 overview, 322
 replay attacks, 323
 session hijacking, 323
 sniffing, 322
 using built-in SSL, 326–30
 using OpenSSL, 330–31
 verifying server certificates with OpenSSL, 331–38
 obtaining root certificate authority certificates, 332
 overview, 331
 verifying the certificates, 332–38
 SSL-Enabled Communication (HTTPS), 116
 sslwrapper class, 330
 Standard Generalized Markup Language (SGML), 145
 start(), 445
 startthread(), 459
 starttls(), 205, 210
 stat(), 212
 stateclass object, 474, 475, 483
 stateless protocol, 388
 Stats class, 362
 stopreactor(), 230, 231
 storbinary(), 281, 282
 storlines(), 281, 282
 str(), 319
 Stream Protocol, 98
 stream socket type, 49
 stream (TCP) communication, 10
 stream type, 47
 StreamRequestHandler class, 351, 352
 StringIO module, 270
 strings, transmitting, 90–92
 leading size indicator, 92
 unique end-of-string identifiers, 91–92
 struct function, 94
 struct module, 93
 structure value, 260
 Subject header, 173
 Subject line, 178, 188
 synchronous communication, 469
 sys.exc_info(), 60
 sys.exit(), 42, 125, 421, 432–33
 sys.exit(1) call, 60
 syslog, logging with, 55–60
 sys.stdout line, 46
 system(), 424
 SystemExit exception, 42
 system.listMethods(), 161, 365
 system.methodHelp(), 161, 365
 system.methodSignature(), 161, 365

T

target setting, 445
 tar.gz file, 332
 TCP basics, 3–6
 addressing, 4
 reliability, 4–5
 routing, 5
 security, 6
 telnet command, 41
 Telnet Protocol, 15
 TerminalPassword class, 117
 testclient.py file, 367
 testcookie key, 392
 test.handler(), 399
 Text objects, 150
 text/plain component, 262
 text/x-diff, 259
 · thread module, 444
 Thread object, 445
 thread pools, 450

threadcode(), 445
threading, 443–68

- avoiding deadlock, 453–55
- being thread-safe, 447–50
- managing access to shared and scarce resources, 450–53
- overview, 443–45, 444–45
- using shared variables, 446–47
- writing threaded clients, 463–67
- writing threaded servers, 455–63
 - overview, 455–57
 - threaded chat server exercise, 457
 - using thread pools, 457–63

threading module, 444, 448

ThreadingHTTPServer class, 348

ThreadingMixIn class, 348, 363

threads.def handleconnection(), 459

threadworker(), 460

“Tidy” library, 128

Time(), 317

TimeFromTicks(), 317

time.mktime(), 179

timeouts, 89–90

TimeRequestHandler, 351

TimeServer class, 351

time.sleep(), 428, 430

Timestamp(), 317

TimestampFromTicks(), 317

time.time(), 317

TitleParser class, 129

TLS (Transport Layer Security), 6, 205–8, 321

To Header (MIME), 171

To header (MIME), 172

toprettyxml(), 157

toxml(), 157

traceroute program, 5

transactions, 302–5

- hiding changes until finished, 303–5
- performance implications of transactions, 303

transfer encodings, 181

transmitting strings, 90–92

- leading size indicator, 92
- unique end-of-string identifiers, 91–92

Transport Layer Security (TLS), 6, 205–8, 321

tree-based parser, 148

try blocks, 42, 460

- try...except clause, 60
- try...finally blocks, 43, 214, 437

TT tags, 385

Twisted

- and IMAP, 225–36
- error handling, 231–36
- logging in, 228–31
- overview, 226–28
- using for servers, 485–88

Twisted IMAP library, 240, 243

Twisted project, 470

twisted.protocols.imap4 module, 263

TXT records, 76

U

UDP (User Datagram Protocol), 7–8, 20, 23, 31–33, 37, 43–45, 47, 49, 95, 96, 100

udpechoserver.py server, 32

UID, 245, 248

uid parameter, 240, 248

UIDNEXT summary item, 240

UIDVALIDITY summary item, 241, 243

uithread(), 466

UNIX, 45, 46, 55, 159, 172, 218, 346, 386, 388, 396, 420, 422, 424, 427, 457, 484

UNIX-like operating system, 7, 14, 16, 19, 21, 38, 45, 55, 56, 66, 68, 77
UNLISTED type, 49
unlock command, 437
\bUnmarked flag, 238
UNSEEN summary item, 241
url variable, 357, 367
URLError exception, 122, 123
urllib, 16, 113, 119, 120
urllib module, 412
urllib2 module, 113, 114, 115, 116, 118, 121, 122, 125, 277
 HTTPError exception, 118
 HTTPPasswordMgr class, 117
 URLError class, 123
 URLError exception, 121
 urlopen(), 117
urllib.quote_plus(), 383, 385
user(), 212
user parameter, 299
user xinetd option, 49
user-visible URLs, 401
using data types, 317–19
uTidylib library, 133
util.FieldStorage class, 409
UUCP (UNIX-to-UNIX Copy Protocol), 58

V

v-card, 259
verify(), 335, 338
version attribute, 389
virus scanners, 172
voidcmd(), 280
voidresp(), 281
vulnerabilities, network. *See* **network vulnerabilities**

W

wait(), 422, 423, 427, 467
wait server, 51–52, 54
wait type, 47
wait xinetd option, 49
waitpid(), 427, 432
watchboth(), 476
Web client access, 113–26
 authenticating, 115–18
 fetching Web pages, 114–15
 handling errors, 121–25
 connection errors, 121–23
 data errors, 123–25
 overview, 113
 submitting form data, 118–21
 with GET, 118–20
 with POST, 120–21
 using non-HTTP protocols, 125
wfile object, 351
wfile variable, 342
Windows, 46, 55, 66, 68, 69, 77, 104, 346, 356
write(), 13, 19, 23, 28, 218, 279, 327, 488
writeevent(), 474, 475
 _writeheaders(), 345
writelastaccess(), 437
writeline(), 279
writerow(), 142
writing threaded clients, 463–67
writing threaded servers, 455–63
 overview, 455–57
 threaded chat server exercise, 457
 using thread pools, 457–63

X

\xfc code, 188
XHTML, 148. *See* **HTML and XHTML, parsing**
xinetd, configuring, 48–50
XML, 127–28

- XML and XML-RPC, 145–66
 - overview, 145–48
 - summary, 166
 - using DOM, 148–58
 - full parsing with DOM, 151–54
 - generating documents with DOM, 154–57
 - type reference, 157–58
 - using XML-RPC, 159–66
 - error handling, 165
 - full-featured example, 162–65
 - introspection, 160–62
 - type handling, 165–66

Z

- Zolera SOAP Infrastructure (ZSI), 159