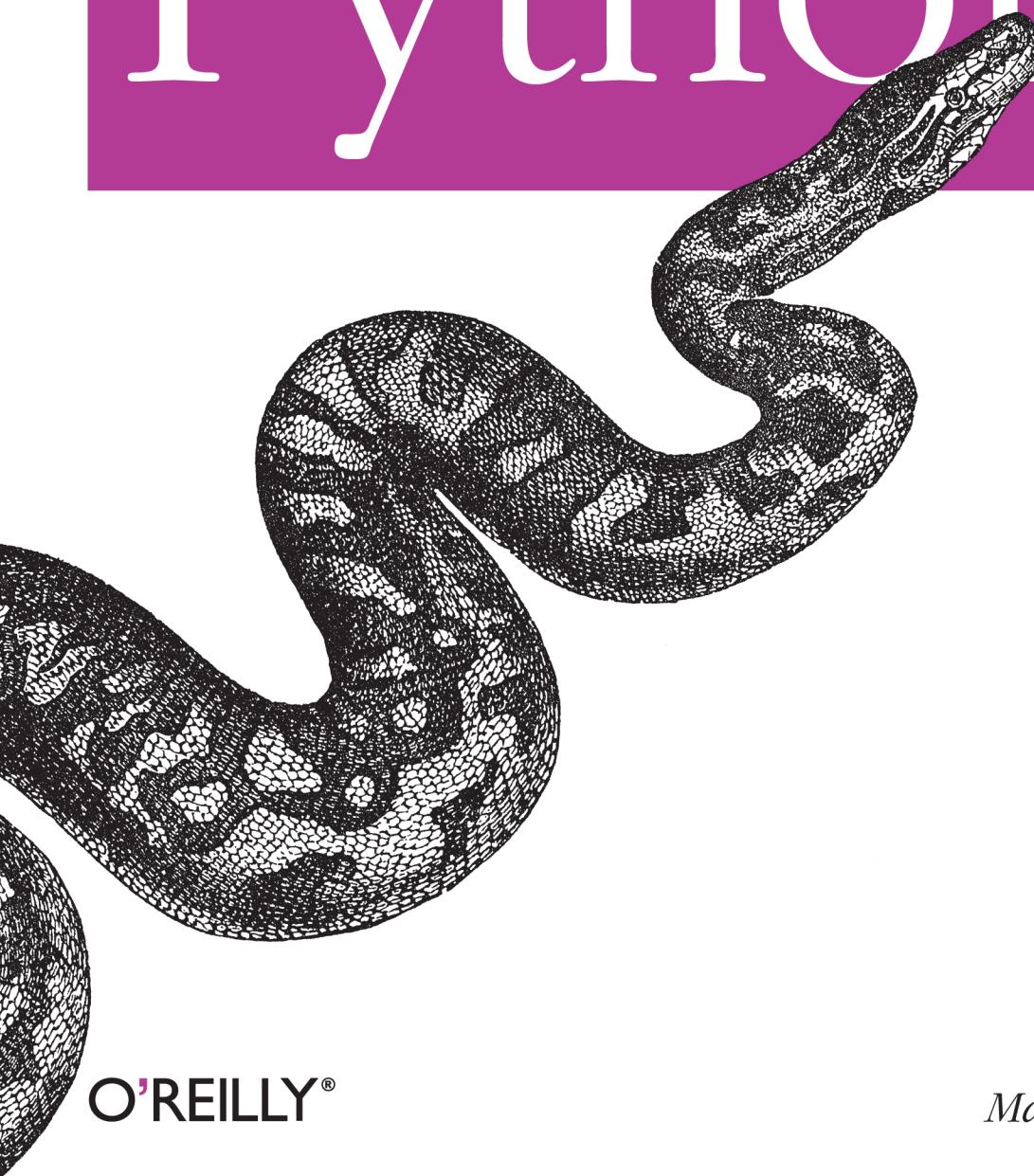


Powerful Object-Oriented Programming

Programming

4th Edition
Covers Python 3.x

Python



O'REILLY®

Mark Lutz

Programming Python

How do you apply Python once you've mastered its fundamentals? This book provides in-depth tutorials on the language's primary application domains—system administration, GUIs, and the Web—and explores its use in databases, networking, front-end scripting layers, text processing, and more. By focusing on commonly used tools and libraries, you'll gain an in-depth understanding of Python's roles in practical, real-world programming.

You'll learn language syntax and programming techniques in a clear and concise manner, with lots of examples that illustrate both correct usage and common idioms. Completely updated for version 3.x, *Programming Python* also delves into the language as a software development tool, with many code examples scaled specifically for that purpose.

TOPICS INCLUDE:

- **Quick Python tour:** Build a simple demo that includes data representation, object-oriented programming, object persistence, GUIs, and website basics.
- **System programming:** Explore system interface tools and techniques for command-line scripting, processing files and folders, running programs in parallel, and more.
- **GUI programming:** Learn to use Python's tkinter widget library to build complete user interfaces.
- **Internet programming:** Access client-side network protocols and email tools, use CGI scripts, and learn website implementation techniques.
- **More ways to apply Python:** Implement data structures, parse text-based information, interface with databases, and extend and embed Python.

Previous programming experience is recommended.

US \$64.99

CAN \$74.99

ISBN: 978-0-596-15810-1



Safari
Books Online

Free online edition
for 45 days with purchase of
this book. Details on last page.

“Here are chapters offering everything from troubleshooting to design specs, with an eye to realistically scaled problems and avoiding common burdens.”

—Diane Donovan
California Bookwatch

Mark Lutz is the world leader in Python training, the author of Python's earliest and bestselling texts, and a pioneering figure in the Python community since 1992. Mark has been a software developer for 25 years and is the author of previous editions of *Programming Python*, as well as O'Reilly's *Learning Python* and *Python Pocket Reference*.

O'REILLY®
oreilly.com

Programming Python

FOURTH EDITION

Programming Python

Mark Lutz

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Programming Python, Fourth Edition

by Mark Lutz

Copyright © 2011 Mark Lutz. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://my.safaribooksonline.com>). For more information, contact our corporate/institutional sales department: (800) 998-9938 or corporate@oreilly.com.

Editor: Julie Steele

Production Editor: Teresa Elsey

Proofreader: Teresa Elsey

Indexer: Lucie Haskins

Cover Designer: Karen Montgomery

Interior Designer: David Futato

Illustrator: Robert Romano

Printing History:

October 1996:	First Edition.
March 2001:	Second Edition.
August 2006:	Third Edition.
December 2010:	Fourth Edition.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. *Programming Python*, the image of an African rock python, and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc., was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 978-0-596-15810-1

[QG]

1292258056

Table of Contents

Preface	xxiii
---------------	-------

Part I. The Beginning

1. A Sneak Preview	3
“Programming Python: The Short Story”	3
The Task	4
Step 1: Representing Records	4
Using Lists	4
Using Dictionaries	9
Step 2: Storing Records Persistently	14
Using Formatted Files	14
Using Pickle Files	19
Using Per-Record Pickle Files	22
Using Shelves	23
Step 3: Stepping Up to OOP	26
Using Classes	27
Adding Behavior	29
Adding Inheritance	29
Refactoring Code	31
Adding Persistence	34
Other Database Options	36
Step 4: Adding Console Interaction	37
A Console Shelve Interface	37
Step 5: Adding a GUI	40
GUI Basics	40
Using OOP for GUIs	42
Getting Input from a User	44
A GUI Shelve Interface	46
Step 6: Adding a Web Interface	52
CGI Basics	52

Running a Web Server	55
Using Query Strings and urllib	57
Formatting Reply Text	59
A Web-Based Shelve Interface	60
The End of the Demo	69

Part II. System Programming

2. System Tools	73
“The os.path to Knowledge”	73
Why Python Here?	73
The Next Five Chapters	74
System Scripting Overview	75
Python System Modules	76
Module Documentation Sources	77
Paging Documentation Strings	78
A Custom Paging Script	79
String Method Basics	80
Other String Concepts in Python 3.X: Unicode and bytes	82
File Operation Basics	83
Using Programs in Two Ways	84
Python Library Manuals	85
Commercially Published References	86
Introducing the sys Module	86
Platforms and Versions	86
The Module Search Path	87
The Loaded Modules Table	88
Exception Details	89
Other sys Module Exports	90
Introducing the os Module	90
Tools in the os Module	90
Administrative Tools	91
Portability Constants	92
Common os.path Tools	92
Running Shell Commands from Scripts	94
Other os Module Exports	100
3. Script Execution Context	103
“I’d Like to Have an Argument, Please”	103
Current Working Directory	104
CWD, Files, and Import Paths	104
CWD and Command Lines	106

Command-Line Arguments	106
Parsing Command-Line Arguments	107
Shell Environment Variables	109
Fetching Shell Variables	110
Changing Shell Variables	111
Shell Variable Fine Points: Parents, <code>putenv</code> , and <code>getenv</code>	112
Standard Streams	113
Redirecting Streams to Files and Programs	114
Redirected Streams and User Interaction	119
Redirecting Streams to Python Objects	123
The <code>io.StringIO</code> and <code>io.BytesIO</code> Utility Classes	126
Capturing the <code>stderr</code> Stream	127
Redirection Syntax in Print Calls	127
Other Redirection Options: <code>os.popen</code> and <code>subprocess</code> Revisited	128
4. File and Directory Tools	135
“Erase Your Hard Drive in Five Easy Steps!”	135
File Tools	135
The File Object Model in Python 3.X	136
Using Built-in File Objects	137
Binary and Text Files	146
Lower-Level File Tools in the <code>os</code> Module	155
File Scanners	160
Directory Tools	163
Walking One Directory	164
Walking Directory Trees	168
Handling Unicode Filenames in 3.X: <code>listdir</code> , <code>walk</code> , <code>glob</code>	172
5. Parallel System Tools	177
“Telling the Monkeys What to Do”	177
Forking Processes	179
The <code>fork/exec</code> Combination	182
Threads	186
The <code>_thread</code> Module	189
The <code>threading</code> Module	199
The <code>queue</code> Module	204
Preview: GUIs and Threads	208
More on the Global Interpreter Lock	211
Program Exits	213
<code>sys</code> Module Exits	214
<code>os</code> Module Exits	215
Shell Command Exit Status Codes	216
Process Exit Status and Shared State	219

Thread Exits and Shared State	220
Interprocess Communication	222
Anonymous Pipes	224
Named Pipes (Fifos)	234
Sockets: A First Look	236
Signals	240
The multiprocessing Module	243
Why multiprocessing?	243
The Basics: Processes and Locks	245
IPC Tools: Pipes, Shared Memory, and Queues	248
Starting Independent Programs	254
And Much More	256
Why multiprocessing? The Conclusion	257
Other Ways to Start Programs	258
The os.spawn Calls	258
The os.startfile call on Windows	261
A Portable Program-Launch Framework	263
Other System Tools Coverage	268
6. Complete System Programs	271
“The Grets of Wrath”	271
A Quick Game of “Find the Biggest Python File”	272
Scanning the Standard Library Directory	272
Scanning the Standard Library Tree	273
Scanning the Module Search Path	274
Scanning the Entire Machine	276
Printing Unicode Filenames	279
Splitting and Joining Files	282
Splitting Files Portably	283
Joining Files Portably	286
Usage Variations	289
Generating Redirection Web Pages	292
Page Template File	293
Page Generator Script	294
A Regression Test Script	297
Running the Test Driver	299
Copying Directory Trees	304
Comparing Directory Trees	308
Finding Directory Differences	309
Finding Tree Differences	311
Running the Script	314
Verifying Backups	316
Reporting Differences and Other Ideas	317

Searching Directory Trees	319
Greps and Globs and Finds	320
Rolling Your Own find Module	321
Cleaning Up Bytecode Files	324
A Python Tree Searcher	327
Visitor: Walking Directories “++”	330
Editing Files in Directory Trees (Visitor)	334
Global Replacements in Directory Trees (Visitor)	336
Counting Source Code Lines (Visitor)	338
Recoding Copies with Classes (Visitor)	339
Other Visitor Examples (External)	341
Playing Media Files	343
The Python webbrowser Module	347
The Python mimetypes Module	348
Running the Script	350
Automated Program Launchers (External)	351

Part III. GUI Programming

7. Graphical User Interfaces	355
“Here’s Looking at You, Kid”	355
GUI Programming Topics	355
Running the Examples	357
Python GUI Development Options	358
tkinter Overview	363
tkinter Pragmatics	363
tkinter Documentation	364
tkinter Extensions	364
tkinter Structure	366
Climbing the GUI Learning Curve	368
“Hello World” in Four Lines (or Less)	368
tkinter Coding Basics	369
Making Widgets	370
Geometry Managers	370
Running GUI Programs	371
tkinter Coding Alternatives	372
Widget Resizing Basics	373
Configuring Widget Options and Window Titles	375
One More for Old Times’ Sake	376
Packing Widgets Without Saving Them	377
Adding Buttons and Callbacks	379
Widget Resizing Revisited: Expansion	380

Adding User-Defined Callback Handlers	382
Lambda Callback Handlers	383
Deferring Calls with Lambdas and Object References	384
Callback Scope Issues	385
Bound Method Callback Handlers	391
Callable Class Object Callback Handlers	392
Other tkinter Callback Protocols	393
Binding Events	394
Adding Multiple Widgets	395
Widget Resizing Revisited: Clipping	396
Attaching Widgets to Frames	397
Layout: Packing Order and Side Attachments	397
The Packer’s Expand and Fill Revisited	398
Using Anchor to Position Instead of Stretch	399
Customizing Widgets with Classes	400
Standardizing Behavior and Appearance	401
Reusable GUI Components with Classes	403
Attaching Class Components	405
Extending Class Components	407
Standalone Container Classes	408
The End of the Tutorial	410
Python/tkinter for Tcl/Tk Converts	412
8. A tkinter Tour, Part 1	415
“Widgets and Gadgets and GUIs, Oh My!”	415
This Chapter’s Topics	415
Configuring Widget Appearance	416
Top-Level Windows	419
Toplevel and Tk Widgets	421
Top-Level Window Protocols	422
Dialogs	426
Standard (Common) Dialogs	426
The Old-Style Dialog Module	438
Custom Dialogs	439
Binding Events	443
Other bind Events	447
Message and Entry	448
Message	448
Entry	449
Laying Out Input Forms	451
tkinter “Variables” and Form Layout Alternatives	454
Checkbutton, Radiobutton, and Scale	457
Checkbuttons	457

Radio Buttons	462
Scales (Sliders)	467
Running GUI Code Three Ways	471
Attaching Frames	471
Independent Windows	476
Running Programs	478
Images	484
Fun with Buttons and Pictures	487
Viewing and Processing Images with PIL	491
PIL Basics	491
Displaying Other Image Types with PIL	493
Creating Image Thumbnails with PIL	496
9. <i>A tkinter Tour, Part 2</i>	507
“On Today’s Menu: Spam, Spam, and Spam”	507
Menus	507
Top-Level Window Menus	508
Frame- and Menubutton-Based Menus	512
Windows with Both Menus and Toolbars	517
Listboxes and Scrollbars	522
Programming Listboxes	524
Programming Scroll Bars	525
Packing Scroll Bars	526
Text	528
Programming the Text Widget	530
Adding Text-Editing Operations	533
Unicode and the Text Widget	538
Advanced Text and Tag Operations	548
Canvas	550
Basic Canvas Operations	550
Programming the Canvas Widget	551
Scrolling Canvases	554
Scollable Canvases and Image Thumbnails	557
Using Canvas Events	560
Grids	564
Why Grids?	564
Grid Basics: Input Forms Revisited	565
Comparing grid and pack	566
Combining grid and pack	568
Making Gridded Widgets Expandable	570
Laying Out Larger Tables with grid	574
Time Tools, Threads, and Animation	582
Using Threads with tkinter GUIs	584

Using the after Method	585
Simple Animation Techniques	588
Other Animation Topics	593
The End of the Tour	595
Other Widgets and Options	595
10. GUI Coding Techniques	597
“Building a Better Mousetrap”	597
GuiMixin: Common Tool Mixin Classes	598
Widget Builder Functions	598
Mixin Utility Classes	599
GuiMaker: Automating Menus and Toolbars	603
Subclass Protocols	607
GuiMaker Classes	608
GuiMaker Self-Test	608
BigGUI: A Client Demo Program	609
ShellGUI: GUIs for Command-Line Tools	613
A Generic Shell-Tools Display	613
Application-Specific Tool Set Classes	615
Adding GUI Frontends to Command Lines	617
GuiStreams: Redirecting Streams to Widgets	623
Using Redirection for the Packing Scripts	627
Reloading Callback Handlers Dynamically	628
Wrapping Up Top-Level Window Interfaces	630
GUIs, Threads, and Queues	635
Placing Data on Queues	636
Placing Callbacks on Queues	640
More Ways to Add GUIs to Non-GUI Code	646
Popping Up GUI Windows on Demand	647
Adding a GUI As a Separate Program: Sockets (A Second Look)	649
Adding a GUI As a Separate Program: Command Pipes	654
The PyDemos and PyGadgets Launchers	662
PyDemos Launcher Bar (Mostly External)	662
PyGadgets Launcher Bar	667
11. Complete GUI Programs	671
“Python, Open Source, and Camaros”	671
Examples in Other Chapters	672
This Chapter’s Strategy	673
PyEdit: A Text Editor Program/Object	674
Running PyEdit	675
PyEdit Changes in Version 2.0 (Third Edition)	682
PyEdit Changes in Version 2.1 (Fourth Edition)	684

PyEdit Source Code	693
PyPhoto: An Image Viewer and Resizer	716
Running PyPhoto	717
PyPhoto Source Code	719
PyView: An Image and Notes Slideshow	727
Running PyView	727
PyView Source Code	732
PyDraw: Painting and Moving Graphics	738
Running PyDraw	738
PyDraw Source Code	738
PyClock: An Analog/Digital Clock Widget	747
A Quick Geometry Lesson	747
Running PyClock	751
PyClock Source Code	754
PyToe: A Tic-Tac-Toe Game Widget	762
Running PyToe	762
PyToe Source Code (External)	763
Where to Go from Here	766

Part IV. Internet Programming

12. Network Scripting	771
“Tune In, Log On, and Drop Out”	771
Internet Scripting Topics	772
Running Examples in This Part of the Book	775
Python Internet Development Options	777
Plumbing the Internet	780
The Socket Layer	781
The Protocol Layer	782
Python’s Internet Library Modules	785
Socket Programming	787
Socket Basics	788
Running Socket Programs Locally	794
Running Socket Programs Remotely	795
Spawning Clients in Parallel	798
Talking to Reserved Ports	801
Handling Multiple Clients	802
Forking Servers	803
Threading Servers	815
Standard Library Server Classes	818
Multiplexing Servers with select	820
Summary: Choosing a Server Scheme	826

Making Sockets Look Like Files and Streams	827
A Stream Redirection Utility	828
A Simple Python File Server	840
Running the File Server and Clients	842
Adding a User-Interface Frontend	843
13. Client-Side Scripting	853
“Socket to Me!”	853
FTP: Transferring Files over the Net	854
Transferring Files with ftplib	854
Using urllib to Download Files	857
FTP get and put Utilities	860
Adding a User Interface	867
Transferring Directories with ftplib	874
Downloading Site Directories	874
Uploading Site Directories	880
Refactoring Uploads and Downloads for Reuse	884
Transferring Directory Trees with ftplib	892
Uploading Local Trees	893
Deleting Remote Trees	895
Downloading Remote Trees	899
Processing Internet Email	899
Unicode in Python 3.X and Email Tools	900
POP: Fetching Email	901
Mail Configuration Module	902
POP Mail Reader Script	905
Fetching Messages	906
Fetching Email at the Interactive Prompt	909
SMTP: Sending Email	910
SMTP Mail Sender Script	911
Sending Messages	913
Sending Email at the Interactive Prompt	919
email: Parsing and Composing Mail Content	921
Message Objects	922
Basic email Package Interfaces in Action	924
Unicode, Internationalization, and the Python 3.1 email Package	926
A Console-Based Email Client	947
Running the pymail Console Client	952
The mailtools Utility Package	956
Initialization File	957
MailTool Class	958
MailSender Class	959
MailFetcher Class	967

MailParser Class	976
Self-Test Script	983
Updating the pymail Console Client	986
NNTP: Accessing Newsgroups	991
HTTP: Accessing Websites	994
The urllib Package Revisited	997
Other urllib Interfaces	999
Other Client-Side Scripting Options	1002
14. The PyMailGUI Client	1005
“Use the Source, Luke”	1005
Source Code Modules and Size	1006
Why PyMailGUI?	1008
Running PyMailGUI	1010
Presentation Strategy	1010
Major PyMailGUI Changes	1011
New in Version 2.1 and 2.0 (Third Edition)	1011
New in Version 3.0 (Fourth Edition)	1012
A PyMailGUI Demo	1019
Getting Started	1020
Loading Mail	1025
Threading Model	1027
Load Server Interface	1030
Offline Processing with Save and Open	1031
Sending Email and Attachments	1033
Viewing Email and Attachments	1037
Email Replies and Forwards and Recipient Options	1043
Deleting Email	1049
POP Message Numbers and Synchronization	1051
Handling HTML Content in Email	1053
Mail Content Internationalization Support	1055
Alternative Configurations and Accounts	1059
Multiple Windows and Status Messages	1060
PyMailGUI Implementation	1062
PyMailGUI: The Main Module	1063
SharedNames: Program-Wide Globals	1066
ListWindows: Message List Windows	1067
ViewWindows: Message View Windows	1085
messagecache: Message Cache Manager	1095
poputil: General-Purpose GUI Pop Ups	1098
wraplines: Line Split Tools	1100
html2text: Extracting Text from HTML (Prototype, Preview)	1102
mailconfig: User Configurations	1105

textConfig: Customizing Pop-Up PyEdit Windows	1110
PyMailGUIHelp: User Help Text and Display	1111
altconfigs: Configuring for Multiple Accounts	1114
Ideas for Improvement	1116
15. Server-Side Scripting	1125
“Oh, What a Tangled Web We Weave”	1125
What’s a Server-Side CGI Script?	1126
The Script Behind the Curtain	1126
Writing CGI Scripts in Python	1128
Running Server-Side Examples	1130
Web Server Options	1130
Running a Local Web Server	1131
The Server-Side Examples Root Page	1133
Viewing Server-Side Examples and Output	1134
Climbing the CGI Learning Curve	1135
A First Web Page	1135
A First CGI Script	1141
Adding Pictures and Generating Tables	1146
Adding User Interaction	1149
Using Tables to Lay Out Forms	1157
Adding Common Input Devices	1163
Changing Input Layouts	1166
Passing Parameters in Hardcoded URLs	1170
Passing Parameters in Hidden Form Fields	1172
Saving State Information in CGI Scripts	1174
URL Query Parameters	1176
Hidden Form Input Fields	1176
HTTP “Cookies”	1177
Server-Side Databases	1181
Extensions to the CGI Model	1182
Combining Techniques	1183
The Hello World Selector	1183
Checking for Missing and Invalid Inputs	1190
Refactoring Code for Maintainability	1192
Step 1: Sharing Objects Between Pages—A New Input Form	1193
Step 2: A Reusable Form Mock-Up Utility	1196
Step 3: Putting It All Together—A New Reply Script	1199
More on HTML and URL Escapes	1201
URL Escape Code Conventions	1202
Python HTML and URL Escape Tools	1203
Escaping HTML Code	1203
Escaping URLs	1204

Escaping URLs Embedded in HTML Code	1205
Transferring Files to Clients and Servers	1209
Displaying Arbitrary Server Files on the Client	1211
Uploading Client Files to the Server	1218
More Than One Way to Push Bits over the Net	1227
16. The PyMailCGI Server	1229
“Things to Do When Visiting Chicago”	1229
The PyMailCGI Website	1230
Implementation Overview	1230
New in This Fourth Edition (Version 3.0)	1233
New in the Prior Edition (Version 2.0)	1235
Presentation Overview	1236
Running This Chapter’s Examples	1237
The Root Page	1239
Configuring PyMailCGI	1240
Sending Mail by SMTP	1241
The Message Composition Page	1242
The Send Mail Script	1242
Error Pages	1246
Common Look-and-Feel	1246
Using the Send Mail Script Outside a Browser	1247
Reading POP Email	1249
The POP Password Page	1250
The Mail Selection List Page	1251
Passing State Information in URL Link Parameters	1254
Security Protocols	1257
The Message View Page	1259
Passing State Information in HTML Hidden Input Fields	1262
Escaping Mail Text and Passwords in HTML	1264
Processing Fetched Mail	1266
Reply and Forward	1267
Delete	1268
Deletions and POP Message Numbers	1272
Utility Modules	1276
External Components and Configuration	1276
POP Mail Interface	1277
POP Password Encryption	1278
Common Utilities Module	1286
Web Scripting Trade-Offs	1291
PyMailCGI Versus PyMailGUI	1292
The Web Versus the Desktop	1293
Other Approaches	1296

Part V. Tools and Techniques

17. Databases and Persistence	1303
“Give Me an Order of Persistence, but Hold the Pickles”	1303
Persistence Options in Python	1303
DBM Files	1305
Using DBM Files	1305
DBM Details: Files, Portability, and Close	1308
Pickled Objects	1309
Using Object Pickling	1310
Pickling in Action	1311
Pickle Details: Protocols, Binary Modes, and <code>_pickle</code>	1314
Shelve Files	1315
Using Shelves	1316
Storing Built-in Object Types in Shelves	1317
Storing Class Instances in Shelves	1318
Changing Classes of Objects Stored in Shelves	1320
Shelve Constraints	1321
Pickled Class Constraints	1323
Other Shelve Limitations	1324
The ZODB Object-Oriented Database	1325
The Mostly Missing ZODB Tutorial	1326
SQL Database Interfaces	1329
SQL Interface Overview	1330
An SQL Database API Tutorial with SQLite	1332
Building Record Dictionaries	1339
Tying the Pieces Together	1342
Loading Database Tables from Files	1344
SQL Utility Scripts	1347
SQL Resources	1354
ORMs: Object Relational Mappers	1354
PyForm: A Persistent Object Viewer (External)	1356
18. Data Structures	1359
“Roses Are Red, Violets Are Blue; Lists Are Mutable, and So Is Set Foo”	1359
Implementing Stacks	1360
Built-in Options	1360
A Stack Module	1362
A Stack Class	1364
Customization: Performance Monitors	1366
Optimization: Tuple Tree Stacks	1367

Optimization: In-Place List Modifications	1369
Timing the Improvements	1371
Implementing Sets	1373
Built-in Options	1374
Set Functions	1375
Set Classes	1377
Optimization: Moving Sets to Dictionaries	1378
Adding Relational Algebra to Sets (External)	1382
Subclassing Built-in Types	1383
Binary Search Trees	1385
Built-in Options	1385
Implementing Binary Trees	1386
Trees with Both Keys and Values	1388
Graph Searching	1390
Implementing Graph Search	1390
Moving Graphs to Classes	1393
Permuting Sequences	1395
Reversing and Sorting Sequences	1397
Implementing Reversals	1398
Implementing Sorts	1399
Data Structures Versus Built-ins: The Conclusion	1400
PyTree: A Generic Tree Object Viewer	1402
19. Text and Language	1405
“See Jack Hack. Hack, Jack, Hack”	1405
Strategies for Processing Text in Python	1405
String Method Utilities	1406
Templating with Replacements and Formats	1408
Parsing with Splits and Joins	1409
Summing Columns in a File	1410
Parsing and Unparsing Rule Strings	1412
Regular Expression Pattern Matching	1415
The re Module	1416
First Examples	1416
String Operations Versus Patterns	1418
Using the re Module	1421
More Pattern Examples	1425
Scanning C Header Files for Patterns	1427
XML and HTML Parsing	1429
XML Parsing in Action	1430
HTML Parsing in Action	1435
Advanced Language Tools	1438
Custom Language Parsers	1440

The Expression Grammar	1440
The Parser's Code	1441
Adding a Parse Tree Interpreter	1449
Parse Tree Structure	1454
Exploring Parse Trees with the PyTree GUI	1456
Parsers Versus Python	1457
PyCalc: A Calculator Program/Object	1457
A Simple Calculator GUI	1458
PyCalc—A “Real” Calculator GUI	1463
20. Python/C Integration	1483
“I Am Lost at C”	1483
Extending and Embedding	1484
Extending Python in C: Overview	1486
A Simple C Extension Module	1487
The SWIG Integration Code Generator	1491
A Simple SWIG Example	1491
Wrapping C Environment Calls	1495
Adding Wrapper Classes to Flat Libraries	1499
Wrapping C Environment Calls with SWIG	1500
Wrapping C++ Classes with SWIG	1502
A Simple C++ Extension Class	1503
Wrapping the C++ Class with SWIG	1505
Using the C++ Class in Python	1507
Other Extending Tools	1511
Embedding Python in C: Overview	1514
The C Embedding API	1515
What Is Embedded Code?	1516
Basic Embedding Techniques	1518
Running Simple Code Strings	1519
Running Code Strings with Results and Namespaces	1522
Calling Python Objects	1524
Running Strings in Dictionaries	1526
Precompiling Strings to Bytecode	1528
Registering Callback Handler Objects	1530
Registration Implementation	1531
Using Python Classes in C	1535
Other Integration Topics	1538

Part VI. The End

21. Conclusion: Python and the Development Cycle	1543
“That’s the End of the Book, Now Here’s the Meaning of Life”	1544
“Something’s Wrong with the Way We Program Computers”	1544
The “Gilligan Factor”	1544
Doing the Right Thing	1545
The Static Language Build Cycle	1546
Artificial Complexities	1546
One Language Does Not Fit All	1546
Enter Python	1547
But What About That Bottleneck?	1548
Python Provides Immediate Turnaround	1549
Python Is “Executable Pseudocode”	1550
Python Is OOP Done Right	1550
Python Fosters Hybrid Applications	1551
On Sinking the Titanic	1552
So What’s “Python: The Sequel”?	1555
In the Final Analysis...	1555
Index	1557

Preface

“And Now for Something Completely Different...”

This book explores ways to apply the Python programming language in common application domains and realistically scaled tasks. It’s about what you can *do* with the language once you’ve mastered its fundamentals.

This book assumes you are relatively new to each of the application domains it covers—GUIs, the Internet, databases, systems programming, and so on—and presents each from the ground up, in *tutorial* fashion. Along the way, it focuses on commonly used tools and libraries, rather than language fundamentals. The net result is a resource that provides readers with an in-depth understanding of Python’s roles in practical, real-world programming work.

As a subtheme, this book also explores Python’s relevance as a *software development* tool—a role that many would classify as well beyond those typically associated with “scripting.” In fact, many of this book’s examples are scaled specifically for this purpose; among these, we’ll incrementally develop email clients that top out at thousands of lines of code. Programming at this full scale will always be challenging work, but we’ll find that it’s also substantially quicker and easier when done with Python.

This Fourth Edition has been updated to present the language, libraries, and practice of Python 3.X. Specifically, its examples use Python 3.1—the most recent version of Python at the time of writing—and its major examples were tested successfully under the third alpha release of Python 3.2 just prior to publication, but they reflect the version of the language common to the entire 3.X line. This edition has also been reorganized in ways that both streamline some of its former material and allow for coverage of newly emerged tools and topics.

Because this edition’s readership will include both newcomers as well as prior edition veterans, I want to use this Preface to expand on this book’s purpose and scope before we jump into code.

About This Book

This book is a tutorial introduction to using Python in common application domains and tasks. It teaches how to apply Python for system administration, GUIs, and the Web, and explores its roles in networking, databases, frontend scripting layers, text processing, and more. Although the Python language is used along the way, this book's focus is on *application* to real-world tasks instead of language fundamentals.

This Book's Ecosystem

Because of its scope, this book is designed to work best as the second of a two-volume set, and to be supplemented by a third. Most importantly, this book is an applications programming follow-up to the core language book *Learning Python*, whose subjects are officially prerequisite material here. Here's how the three books are related:

- *Learning Python* covers the fundamentals of Python programming in depth. It focuses on the core Python language, and its topics are prerequisite to this book.
- *Programming Python*, this book, covers the application of Python to real-world programming tasks. It focuses on libraries and tools, and it assumes you already know Python fundamentals.
- *Python Pocket Reference* provides a quick reference to details not listed exhaustively here. It doesn't teach much, but it allows you to look up details fast.

In some sense, this book is to application programming what *Learning Python* is to the core language—a gradual tutorial, which makes almost no assumptions about your background and presents each topic from the ground up. By studying this book's coverage of Web basics, for example, you'll be equipped to build simple websites, and you will be able to make sense of more advanced frameworks and tools as your needs evolve. GUIs are similarly taught incrementally, from basic to advanced.

In addition, this book is designed to be supplemented by the quick-reference book *Python Pocket Reference*, which provides the small details finessed here and serves as a resource for looking up the fine points. That book is reference only, and is largely void of both examples and narrative, but it serves to augment and complement both *Learning Python*'s fundamentals and *Programming Python*'s applications. Because its current Fourth Edition gives both Python 2.X and 3.X versions of the tools it covers, that book also serves as a resource for readers transitioning between the two Python lines (more on this in a moment).*

* Disclosure: I am the author of all three books mentioned in this section, which affords me the luxury of tightly controlling their scopes in order to avoid overlap. It also means that as an author, I try to avoid commenting on the many other Python books available, some of which are very good and may cover topics not addressed in any of my own books. Please see the Web for other Python resources. All three of my books reflect my 13 years on the Python training trail and stem from the original *Programming Python* written back in 1995 <insert grizzled prospector photo here>.

What This Book Is Not

Because of the scopes carved out by the related books I just mentioned, this book's scope follows two explicit constraints:

- It does not cover Python language fundamentals
- It is not intended as a language reference

The former of these constraints reflects the fact that core language topics are the exclusive domain of *Learning Python*, and I encourage you to consult that book before tackling this one if you are completely new to the Python language, as its topics are assumed here. Some language techniques are shown by example in this book too, of course, and the larger examples here illustrate how core concepts come together into realistic programs. OOP, for example, is often best sampled in the context of the larger programs we'll write here. Officially, though, this book assumes you already know enough Python fundamentals to understand its example code. Our focus here is mostly on libraries and tools; please see other resources if the basic code we'll use in that role is unclear.

The latter of the two constraints listed above reflects what has been a common misconception about this book over the years (indeed, this book might have been better titled *Applying Python* had we been more clairvoyant in 1995). I want to make this as clear as I can: this is *not* a reference book. It is a *tutorial*. Although you can hunt for some details using the index and table of contents, this book is not designed for that purpose. Instead, *Python Pocket Reference* provides the sort of quick reference to details that you'll find useful once you start writing nontrivial code on your own. There are other reference-focused resources available, including other books and Python's own reference manuals set. Here, the goal is a gradual tutorial that teaches you how to apply Python to common tasks but does not document minute details exhaustively.

About This Fourth Edition

If this is the first edition of this book you've seen, you're probably less interested in recent changes, and you should feel free to skip ahead past this section. For readers of prior editions, though, this Fourth Edition of this book has changed in three important ways:

- It's been updated to cover Python 3.X (only).
- It's been slimmed down to sharpen its focus and make room for new topics.
- It's been updated for newly emerged topics and tools in the Python world.

The first of these is probably the most significant—this edition employs the Python 3.X language, its version of the standard library, and the common practice of its users. To better explain how this and the other two changes take shape in this edition, though, I need to fill in a few more details.

Specific Changes in This Edition

Because the prior versions of this book were widely read, here is a quick rundown of some of the most prominent specific changes in this edition:

Its existing material was shortened to allow for new topics

The prior edition of this book was also a 1600-page volume, which didn't allow much room for covering new Python topics (Python 3.X's Unicode orientation alone implies much new material). Luckily, recent changes in the Python world have allowed us to pare down some less critical existing material this time around, in order to free up room for new coverage.

Depth was not sacrificed in the process, of course, and this is still just as substantial a book as before. In general, though, avoiding new growth was a primary goal of this update; many of the other specific changes and removals I'll mention below were made, in part, to help accommodate new topics.

It covers 3.X (only)

This book's examples and narrative have been updated to reflect and use the 3.X version of Python. Python 2.X is no longer supported here, except where 3.X and 2.X Pythons overlap. Although the overlap is large enough to make this of use to 2.X readers too, this is now officially a 3.X-only text.

This turns out to be a major factor behind the lack of growth in this edition. By restricting our scope to Python 3.X—the incompatible successor to the Python 2.X line, and considered to be Python's future—we were able to avoid doubling the coverage size in places where the two Python lines differ. This version limit is especially important in a book like this that is largely about more advanced examples, which can be listed in only one version's style.

For readers who still straddle the 2.X and 3.X worlds, I'll say more about Python 3.X changes later in this Preface. Probably the most significant 3.X-related change described there is the new Internationalization support in PyEdit and PyMailGUI; though 2.X had Unicode too, its new prominence in 3.X almost forces such systems to rethink their former ASCII-only ways.

Inclusion of newly emerged libraries and tools

Since the prior edition, a variety of new libraries and tools have either come online or risen in popularity, and they get new mention here. This includes new standard library tools such as `subprocess` (in Chapters 2 and 3) and `multiprocessing` (in Chapter 5), as well as new third-party web frameworks and ORM database toolkits. Most of these are not covered extensively (many popular third-party extensions are complex systems in their own right and are best covered by dedicated books), but they are at the least introduced in summary form here.

For example, Python 3.1's new `tkinter.ttk` Tk themed widget set shows up in Chapter 7 now, but only briefly; as a rule, this edition prefers to mention such extensions in passing, rather than attempting to show you code without adequate explanation.

This Preface was tightened up

I've removed all the instructions for using and running program examples. Instead, please consult the `README` file in the examples distribution for example usage details. Moreover, most of the original acknowledgments are gone here because they are redundant with those in [Learning Python](#); since that book is now considered a prerequisite, duplication of material here is unwarranted. A description of book contents was also deleted; please see the table of contents for a preview of this book's structure.

The initial Python overview chapter is gone

I've removed the prior edition's "managerial summary" chapter which introduced Python's strong points, prominent users, philosophies, and so on. Proselytizing does play an important role in a field that sometimes asks the "why" questions less often than it should. Indeed, if advocacy had not been part of the Python experience, we'd probably all be using Perl or shell languages today!

However, this chapter has now grown completely redundant with a similar chapter in [Learning Python](#). Since that book is a precursor to this one, I opted to not devote space to restating "Pythonista" propaganda here (fun as it may be). Instead, this book assumes you already know why Python is worth using, and we jump right into applying it here.

The conclusion's postscripts are gone

This book's conclusion comes from the first edition, and it is now 15 years old. Naturally, some of it reflects the Python mindset from that period more than that of today. For example, its focus on Python's role in hybrid applications seemed more important in 1995 than in 2010; in today's much larger Python world, most Python users never deal with linked-in C code at all.

In prior editions, I added postscripts for each edition to elaborate on and update the ideas presented in the book's conclusion. These postscripts are gone now, replaced by a short note at the start of the conclusion. I opted to keep the conclusion itself, though, because it's still relevant to many readers and bears some historic value. Well, that, plus the jokes...

The forewords are gone

For reasons similar to those of the prior two points, the accumulated forewords from the prior three editions were also dropped this time around. You can read all about Python creator Guido van Rossum's historical rationale for Python's evolution in numerous places on the Web, if you are so inclined. If you are interested in how Python has changed technically over the years, see also the "What's New" documents that are part of the Python standard manuals set (available at <http://www.python.org/doc>, and installed alongside Python on Windows and other platforms).

The C integration part has been reduced to just one chapter

I've reduced the C extending and embedding part's material to one shorter chapter at the end of the tools part, which briefly introduces the core concepts in this

domain. Only a fraction of Python users must care about linking in C libraries today, and those who do already have the skills required to read the larger and more compete example of integration present in the source code of Python itself. There is still enough to hint at possibilities here, but vast amounts of C code have been cut, in deference to the better examples you'll find in Python's own code.

The systems programming part was condensed and reworked

The former two larger system examples chapters have been merged into one shorter one, with new or greatly rewritten examples. In fact, this part ([Part II](#)) was probably overhauled the most of any part in the book. It incorporates new tools such as `subprocess` and `multiprocessing`, introduces sockets earlier, and removes dated topics and examples still lingering from prior editions. Frankly, a few of the file-oriented examples here dated back to the 1990s, and were overdue for a general refresh. The initial chapter in this part was also split into two to make its material easier to read (shell context, including streams, gets its own chapter now), and a few large program listings here (including the auto-configuring launcher scripts) are now external suggested reading.

Some larger examples were removed (but are available in the examples distribution)

Along the same lines, two of the larger GUI examples in the prior edition, *PyTree* and *PyForm*, have been removed. Instead, their updated code is available in the book's examples distribution package, as suggested supplemental reading. You'll still find many larger examples covered and listed in this edition—including both GUI- and Web-based renderings of full-featured email clients, along with image viewers, calculators, clocks, Unicode-aware text editors, drawing programs, regression test scripts, and more. However, because the code of the examples removed doesn't add much to what is already covered, and because they were already largely self-study examples anyhow, I've made them optional and external to the printed text in this edition.

The advanced Internet topics chapter was replaced by brief summaries

I've cut the advanced Internet topics chapter completely, leaving only simple summaries at the start of the Internet part (intentionally mirroring the GUI option summaries at the start of the GUI part). This includes prior coverage for tools such as the ZOPE web framework, COM, Windows active scripting and ASP, HTMLgen, Python Server Pages (PSP), Jython, and the now very dated Grail system. Some of these systems still receive honorable mention in the summaries, but none are now presented in any sort of detail. Summaries of new tools (including many of those listed in the following paragraph) were added to this set, but again, in brief fashion with no example code.

Despite authors' best attempts to foresee the future, the Web domain evolves faster than books like this can. For instance, Web frameworks like Django, Google's App Engine, TurboGears, pylons, and web2py are now popular alternatives to ZOPE. Similarly, the .NET framework supersedes much of COM on Windows; IronPython now provides the same type of integration for .NET as Jython did first

for Java; and active scripting has been eclipsed by AJAX and JavaScript-oriented frameworks on the client such as Flex, Silverlight, and pyjamas (generally known today as rich Internet applications, RIAs). Culture shift aside, the examples formerly presented in this category were by themselves also insufficient to either teach or do justice to the subject tools.

Rather than including incomplete (and nearly useless) coverage of tools that are prone to both evolution and demise during this edition's expected lifespan, I now provide only brief overviews of the current hot topics in the Web domain, and I encourage readers to search the Web for more details. More to the point, the goal of the book you're reading is to impart the sort of in-depth knowledge of Internet and Web fundamentals that will allow you to use more advanced systems well, when you're ready to take the leap.

One exception here: the XML material of this prior chapter was spared and relocated in expanded form to the text processing chapter (where it probably belonged all along). In a related vein, the coverage of ZOPE's ZODB object-oriented database was retained, although it was shortened radically to allow new coverage of ORMs such as SQLAlchemy and SQLObject (again, in overview form).

Use of tools available for 3.X today

At this writing, Python 3.X is still in its adoption phase, and some of the third-party tools that this book formerly employed in its examples are still available in Python 2.X form only. To work around this temporary flux, I've changed some code to use alternatives that already support 3.X today.

The most notable of these is the SQL database section—this now uses the in-process SQLite library, which is a standard part of Python and already in 3.X form, rather than the enterprise-level MySQL interface which is still at 2.X today. Luckily, the Python portable database API allows scripts to work largely the same on both, so this is a minor pragmatic sacrifice.

Of special note, the PIL extension used to display JPEGs in the GUI part was ported to 3.1 just when it was needed for this update, thanks to Fredrik Lundh. It's still not officially released in 3.X form as I submit the final draft of this book in July 2010, but it should be soon, and 3.X patches are provided in the book examples package as a temporary measure.

Advanced core language topics are not covered here

More advanced Python language tools such as descriptors, properties, decorators, metaclasses, and Unicode text processing basics are all part of the core Python language. Because of that, they are covered in the Fourth Edition of [Learning Python](#), not here. For example, Unicode text and the changes it implies for files, filenames, sockets, and much more are discussed as encountered here, but the fundamentals of Unicode itself are not presented in complete depth. Some of the topics in this category are arguably application-level related too (or at least of interest to tool builders and API developers in general), but their coverage in [Learning](#)

[Python](#) allows us to avoid additional growth here. Please see that book for more on these subjects.

Other random bits

Naturally, there were additional smaller changes made along the way. For example, `tkinter.grid` method is used instead of `pack` for layout of most input forms, because it yields a more consistent layout on platforms where label font sizes don't match up with entry widget height (including on a Windows 7 netbook laptop, this edition's development machine). There's also new material scattered throughout, including a new exploration of redirecting streams to sockets in the Internet part; a new threaded and Unicode-aware "grep" dialog and process-wide change tests on exit in the `PyEdit` example; and other things you are probably better off uncovering along the way than reading further about in this Preface.

I also finally replaced some remaining "#" comment blocks at the top of source files with docstrings (even, for consistency, in scripts not meant to be imported, though some "#" lines are retained in larger examples to offset the text); changed a few lingering "while 1" to "while True"; use `+=` more often; and cleaned up a few other cases of now-dated coding patterns. Old habits may die hard, but such updates make the examples both more functional and more representative of common practice today.

Although new topics were added, all told, four chapters were cut outright (the non-technical introduction, one of the system example chapters, advanced Internet topics, and one integration chapter), some additional examples and material were trimmed (including `PyForm` and `PyTree`), and focus was deliberately restricted to Python 3.X and application fundamentals to conserve space.

What's Left, Then?

The combined effect of all the changes just outlined is that this edition more concisely and sharply reflects its core focus—that of a tutorial introduction to ways to apply Python in common programming domains. Nevertheless, as you can tell from this book's page count, it is still a substantial and *in-depth* book, designed to be a first step on your path to mastering realistic applications of Python.

Contrary to recent trends (and at some risk of being branded a heretic), I firmly believe that the job of books like this one is to elevate their readers, not pander to them. Lowering the intellectual bar does a disservice both to readers and to the fields in which they hope to work. While that means you won't find as many cartoons in this book as in some, this book also won't insult you by emphasizing entertainment at the expense of technical depth. Instead, the goal of my books is to impart sophisticated concepts in a satisfying and substantive way and to equip you with the tools you'll need in the real world of software development.

There are many types of learners, of course, and no one book can ever satisfy every possible audience. In fact, that's why the original version of this book later became two, with language basics delegated to [Learning Python](#). Moreover, one can make a case for a distinction between *programmers*, who must acquire deep software development skills, and *scripters*, who do not. For some, a rudimentary knowledge of programming may be enough to leverage a system or library that solves the problem at hand. That is, until their coding forays start encroaching on the realm of full-scale software engineering—a threshold that can inspire disappointment at worst, but a better appreciation of the challenging nature of this field at best.

No matter which camp you're from, it's important to understand this book's intent up-front. If you're looking for a shortcut to proficiency that's light on technical content, you probably won't be happy with this book (or the software field in general). If your goal is to master programming Python well, though, and have some fun along the way, you'll probably find this book to be an important piece of your learning experience.

At the end of the day, learning to program well is much more demanding than implied by some contemporary media. If you're willing to invest the focus and effort required, though, you'll find that it's also much more rewarding. This is especially true for those who equip themselves for the journey with a programmer-friendly tool like Python. While no book or class can turn you into a Python "Master of the Universe" by itself, this book's goal is to help you get there, by shortening your start-up time and providing a solid foundation in Python's most common application domains.

Python 3.X Impacts on This Book

As mentioned, this edition now covers Python 3.X only. Python 3.X is an incompatible version of the language. The 3.X core language itself is very similar to Python 2.X, but there are substantial changes in both the language and its many standard libraries. Although some readers with no prior background in 2.X may be able to bypass the differences, the changes had a big impact on the content of this edition. For the still very large existing Python 2.X user base, this section documents the most noteworthy changes in this category.

If you're interested in 2.X differences, I also suggest finding a copy of the Fourth Edition of the book [Python Pocket Reference](#) described earlier. That book gives both 2.X and 3.X versions of core language structures, built-in functions and exceptions, and many of the standard library modules and tools used in this book. Though not designed to be a reference or version translator per se, the Fourth Edition of [Learning Python](#) similarly covers both 2.X and 3.X, and as stated, is prerequisite material to this book. The goal of this 3.X-only [Programming Python](#) is not to abandon the current vast 2.X user base in favor of a still imaginary one for 3.X; it is to help readers with the migration, and avoid doubling the size of an already massive book.

Specific 3.X Changes

Luckily, many of the 2.X/3.X differences that impact this book’s presentation are trivial. For instance, the `tkinter` GUI toolkit, used extensively in this book, is shown under its 3.X `tkinter` name and package structure only; its 2.X `Tkinter` module incarnation is not described. This mostly boils down to different import statements, but only their Python 3 versions are given here. Similarly, to satisfy 3.X module naming conventions, 2.X’s `anydbm`, `Queue`, `thread`, `StringIO.StringIO`, and `urllib.open` become `dbm`, `queue`, `_thread`, `io.StringIO`, and `urllib.request.urlopen`, respectively, in both Python 3.X and this edition. Other tools are similarly renamed.

On the other hand, 3.X implies broader idiomatic changes which are, of course, more radical. For example, Python 3.X’s new Unicode awareness has inspired fully Internationalized versions of the PyEdit text editor and the PyMailGUI email client examples in this edition (more on this in a moment). Furthermore: the replacement of `os.popen2` with the `subprocess` module required new examples; the demise of `os.path.walk` in favor of `os.walk` allowed some examples to be trimmed; the new Unicode and binary dichotomy of files and strings impacted a host of additional existing examples and material; and new modules such as `multiprocessing` offer new options covered in this edition.

Beyond such library changes, core language changes in Python 3 are also reflected in this book’s example code. For instance, changes to 2.X’s `print`, `raw_input`, `keys`, `has_key`, `map`, and `apply` all required changes here. In addition, 3.X’s new *package-relative* import model impacted a few examples including `mailtoools` and expression parsers, and its different flavor of *division* forced some minor math updates in canvas-based GUI examples such as PyClock, PyDraw, and PyPhoto.

Of note here, I did not change all *% string formatting* expressions to use the new `str.format`, since both forms are supported in Python 3.1, and it now appears that they will be either indefinitely or forever. In fact, per a “grep” we’ll build and run in [Chapter 11](#)’s PyEdit example, it seems that this expression still appears over 3,000 times in Python 3.1’s own library code. Since I cannot predict Python evolution completely, see the first chapter for more on this if it ever requires updates in an unexpected future.

Also because of the 3.X scope, this edition is unable to use some third-party packages that are still in 2.X form only, as described earlier. This includes the leading MySQL interface, ZODB, PyCrypto, and others; as also mentioned, PIL was ported to 3.1 for use in this book, but this required a special patch and an official 3.X release is still presently pending. Many of these may be available in 3.X form by the time you read these words, assuming the Python world can either break some of the current cross dependencies in 2.X packages or adopt new 3.X-only tools.

Language Versus Library: Unicode

As a book focused on applications instead of core language fundamentals, language changes are not always obtrusive here. Indeed, in retrospect the book *Learning Python* may have been affected by 3.X core language changes more than this book. In most cases here, more example changes were probably made in the name of clarity or functionality than in support of 3.X itself.

On the other hand, Python 3.X does impact much code, and the impacts can be subtle at times. Readers with Python 2.X backgrounds will find that while 3.X core language changes are often simple to apply, updates required for changes in the 3.X standard library are sometimes more far reaching.

Chief among these, Python 3.X’s Unicode strings have had broad ramifications. Let’s be honest: to people who have spent their lives in an ASCII world, the impacts of the 3.X Unicode model can be downright aggravating at times! As we’ll see in this book, it affects file content; file names; pipe descriptors; sockets; text in GUIs; Internet protocols such as FTP and email; CGI scripts; and even some persistence tools. For better or worse, once we reach the world of applications programming as covered in this book, Unicode is no longer an optional topic for many or most Python 3.X programmers.

Of course, Unicode arguably never should have been entirely optional for many programmers in the first place. Indeed, we’ll find that things that may have appeared to work in 2.X never really did—treating text as raw byte strings can mask issues such as comparison results across encodings (see the grep utility of [Chapter 11](#)’s PyEdit for a prime example of code that should fail in the face of Unicode mismatches). Python 3.X elevates such issues to potentially every programmer’s panorama.

Still, porting nontrivial code to 3.X is not at all an insurmountable task. Moreover, many readers of this edition have the luxury of approaching Python 3.X as their first Python and need not deal with existing 2.X code. If this is your case, you’ll find Python 3.X to be a robust and widely applicable scripting and programming language, which addresses head-on many issues that once lurked in the shadows in 2.X.

Python 3.1 Limitations: Email, CGI

There’s one exception that I should call out here because of its impact on major book examples. In order to make its code relevant to the widest possible audience, this book’s major examples are related to Internet *email* and have much new support in this edition for Internationalization and Unicode in this domain. [Chapter 14](#)’s PyMailGUI and [Chapter 16](#)’s PyMailCGI, and all the prior examples they reuse, fall into this category. This includes the PyEdit text editor—now Unicode-aware for files, display, and greps.

On this front, there is both proverbial good news and bad. The good news is that in the end, we will be able to develop the feature-rich and fully Internationalized PyMailGUI email client in this book, using the `email` package as it currently exists. This will include support for arbitrary encodings in both text content and message headers, for

both viewing and composing messages. The less happy news is that this will come at some cost in workaround complexity in Python 3.1.

Unfortunately, as we'll learn in [Chapter 13](#), the `email` package in Python 3.1 has a number of issues related to `str/bytes` combinations in Python 3.X. For example, there's no simple way to guess the encoding needed to convert mail `bytes` returned by the `poplib` module to the `str` expected by the `email` parser. Moreover, the `email` package is currently broken altogether for some types of messages, and it has uneven or type-specific support for some others.

This situation appears to be temporary. Some of the issues encountered in this book are already scheduled to be repaired (in fact, one such fix in 3.2 required a last-minute patch to one of this book's 3.1 workarounds in [Chapter 13](#)). Furthermore, a new version of `email` is being developed to accommodate the 3.X Unicode/bytes dichotomy more accurately, but it won't materialize until long after this book is published, and it might be backward-incompatible with the current package's API, much like Python 3.X itself. Because of that, this book both codes workarounds and makes some assumption along the way, but please watch its website (described ahead) for required updates in future Pythons. One upside here is that the dilemmas posed neatly reflect those common in realistic programming—an underlying theme of this text.

These issues in the `email` package are also inherited by the `cgi` module for CGI file uploads, which are in large measure broken in 3.1. CGI scripts are a basic technique eclipsed by many web frameworks today, but they still serve as an entry-level way to learn Web fundamentals and are still at the heart of many larger toolkits. A future fix seems likely for this 3.1 flaw as well, but we have to make do with nonbinary CGI file uploads for this edition in Chapters [15](#) and [16](#), and limited email attachments in Py-MailCGI. This seems less than ideal nearly two years after 3.0's release, but such is life in the dynamic worlds of both software development at large and books that aim to lead the curve instead of following it.

Using Book Examples

Because this book's examples form much of its content, I want to say a few words about them up front.

Where to Look for Examples and Updates

As before, examples, updates, corrections, and supplements for this book will be maintained at the author's website, which lives officially at the following URL:

<http://www.rmi.net/~lutz/about-pp4e.html>

This page at my book support website will contain links to all supplemental information related to this version of the book. Because I don't own that domain name, though, if

that link ceases to be during this book’s shelf life, try the following alternative site as a fallback option:

<http://learning-python.com/books/about-pp4e.html> (alternative location)

If neither of those links work, try a general web search (which, of course, is what most readers will probably try first anyhow).

Wherever it may live, this website (as well as O’Reilly’s, described in the next section) is where you can fetch the book *examples distribution package*—an archive file containing all of the book’s examples, as well as some extras that are mentioned but not listed in the book itself. To work along without having to type the examples manually, download the package, unpack it, and consult its *README.txt* file for usage details. I’ll describe how example labels and system prompts in this book imply file locations in the package when we use our first script in the first chapter.

As for the first three editions, I will also be maintaining an informal “blog” on this website that describes Python changes over time and provides general book-related notes and updates that you should consider a supplemental appendix to this text.

O’Reilly’s website for this book, described later in this Preface, also has an errata report system, and you can report issues at either my site or O’Reilly’s. I tend to keep my book websites more up to date, but it’s not impossible that O’Reilly’s errata page may supersede mine for this edition. In any event, you should consider the union of these two lists to be the official word on book corrections and updates.

Example Portability

The examples in this book were all developed, tested, and run under Windows 7, and Python 3.1. The book’s major examples were all tested and ran successfully on the upcoming Python 3.2, too (its alpha 3 release), just before the book went to the printer, so most or all of this book applies to Python 3.2 as well. In addition, the C code of [Chapter 20](#) and a handful of parallel programming examples were run under Cygwin on Windows to emulate a Unix environment.

Although Python and its libraries are generally platform neutral, some of this book’s code may require minor changes to run on other platforms, such as Mac OS X, Linux, and other Unix variants. The tkinter GUI examples, as well as some systems programming scripts, may be especially susceptible to platform differences. Some portability issues are pointed out along the way, but others may not be explicitly noted.

Since I had neither time nor budget to test on and accommodate all possible machines that readers might use over the lifespan of this book, updates for platform-specific behaviors will have to fall into the suggested exercises category. If you find a platform dependency and wish to submit a patch for it, though, please see the updates site listed earlier; I’ll be happy to post any platform patches from readers there.

Demo Launchers

The book examples package described earlier also includes portable example demo launcher scripts named PyDemos and PyGadgets, which provide a quick look at some of this book’s major GUI- and Web-based examples. These scripts and their launchers, located at the top of the examples tree, can be run to self-configure program and module search paths, and so can generally be run immediately on compatible platforms, including Windows. See the package’s README files as well as the overviews near the end of Chapters 6 and 10 for more on these scripts.

Code Reuse Policies

We now interrupt this Preface for a word from the legal department. This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you’re reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O’Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product’s documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: “*Programming Python*, Fourth Edition, by Mark Lutz (O’Reilly). Copyright 2011 Mark Lutz, 978-0-596-15810-1.”

Contacting O’Reilly

I described my own examples and updates sites in the prior section. In addition to that advice, you can also address comments and questions about this book to the publisher:

O’Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States and Canada)
707-827-7000 (international/local)
707-829-0104 (fax)

As mentioned, O’Reilly maintains a web page for this book, which lists errata, examples, and any additional information. You can access this page at:

<http://oreilly.com/catalog/9780596158101>

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about books, conferences, software, Resource Centers, and the O'Reilly Network, see the O'Reilly website at:

<http://www.oreilly.com>

Conventions Used in This Book

The following font conventions are used in this book:

Italic

Used for file and directory names, to emphasize new terms when first introduced, and for some comments within code sections

Constant width

Used for code listings and to designate modules, methods, options, classes, functions, statements, programs, objects, and HTML tags

Constant width bold

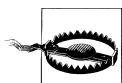
Used in code sections to show user input

Constant width italic

Used to mark replaceables



This icon designates a note related to the nearby text.



This icon designates a warning related to the nearby text.

Acknowledgments

I acknowledged numerous people in the preface of *Learning Python*, Fourth Edition, less than a year ago; because that book is a precursor to this one, and because the set is largely the same, I won't repeat the list in its entirety here. In short, though, I'm grateful to:

- O'Reilly, for promoting Python, and publishing "meaty" books in the Open Source domain
- The Python community, which has occupied sizeable portions of my world since 1992
- The thousands of students who attended the 250 Python classes I've taught since 1997

- The hundreds of thousands who read the 12 editions of the three Python books I've written since 1995
- Monty Python, Python's namesake, for so many great bits to draw from (more in the next chapter)

Although writing is ultimately a solitary task, the ideas that spring forth owe much to the input of many. I'm thankful for all the feedback I've been fortunate to receive over the last 18 years, both from classes and from readers. Students really are the best teachers of teachers.

On the (overly) personal front, I'd like to thank my brothers and sister for old days, as well as my children, Michael, Samantha, and Roxanne, for bragging rights.

And I'm especially thankful for my wife, Vera, who somehow managed to append very good things to this otherwise immutable object.

—Mark Lutz, July 2010

So, What's Python?

As discussed, this book won't devote much space to Python fundamentals, and we'll defer an abstract discussion of Python roles until the Conclusion, after you've had a chance to see it in action firsthand. If you are looking for a concise definition of this book's topic, though, try this:

Python is a general-purpose, open source computer programming language. It is optimized for software quality, developer productivity, program portability, and component integration. Python is used by at least hundreds of thousands of developers around the world in areas such as Internet scripting, systems programming, user interfaces, product customization, numeric programming, and more. It is generally considered to be among the top four or five most widely-used programming languages in the world today.

As a popular language focused on shrinking development time, Python is deployed in a wide variety of products and roles. Counted among its current user base are Google, YouTube, Industrial Light & Magic, ESRI, the BitTorrent file sharing system, NASA's Jet Propulsion Lab, the game Eve Online, and the National Weather Service. Python's application domains range from system administration, website development, cell phone scripting, and education to hardware testing, investment analysis, computer games, and spacecraft control.

Among other things, Python sports a remarkably simple, readable, and maintainable syntax; integration with external components coded in other languages; a multi-paradigm design, with OOP, functional, and modular structures; and a vast collection of precoded interfaces and utilities. Its tool set makes it a flexible and agile language, ideal for both quick tactical tasks as well as longer-range strategic application development efforts. Although it is a general-purpose language, Python is often called a *scripting language* because it makes it easy to utilize and direct other software components.

Perhaps Python's best asset, though, is simply that it makes software development more rapid and enjoyable. There is a class of people for whom programming is an end in itself. They enjoy the challenge. They write software for the pure pleasure of doing so and often view commercial or career reward as secondary consequence. This is the class that largely invented the Internet, open source, and Python. This is also the class that has historically been a primary audience for this book. As they've often relayed, with a tool like Python, programming can be just plain *fun*.

To truly understand how, read on; though something of a side effect, much of this book serves as a demonstration of Python's ideals in action in real-world code. As we'll see, especially when combined with toolkits for GUIs, websites, systems programming, and so on, Python serves as *enabling technology*.

PART I

The Beginning

This part of the book gets things started by taking us on a quick tour that reviews Python fundamental prerequisites and introduces some of the most common ways it is applied.

Chapter 1

This chapter kicks things off by using a simple example—recording information about people—to briefly introduce some of the major Python application domains we'll be studying in this book. We'll migrate the same example through multiple steps. Along the way, we'll meet databases, GUIs, websites, and more. This is something of a demo chapter, designed to pique your interest. We won't learn the full story here, but we'll have a chance to see Python in action before digging into the details. This chapter also serves as a review of some core language ideas you should be familiar with before starting this book, such as data representation and object-oriented programming (OOP).

The point of this part of the book is not to give you an in-depth look at Python, but just to let you sample its application and to provide you with a quick look at some of Python's broader goals and purposes.

A Sneak Preview

“Programming Python: The Short Story”

If you are like most people, when you pick up a book as large as this one, you’d like to know a little about what you’re going to be learning before you roll up your sleeves. That’s what this chapter is for—it provides a demonstration of some of the kinds of things you can do with Python, before getting into the details. You won’t learn the full story here, and if you’re looking for complete explanations of the tools and techniques applied in this chapter, you’ll have to read on to later parts of the book. The point here is just to whet your appetite, review a few Python basics, and preview some of the topics to come.

To do this, I’ll pick a fairly simple application task—constructing a database of records—and migrate it through multiple steps: interactive coding, command-line tools, console interfaces, GUIs, and simple web-based interfaces. Along the way, we’ll also peek at concepts such as data representation, object persistence, and object-oriented programming (OOP); explore some alternatives that we’ll revisit later in the book; and review some core Python ideas that you should be aware of before reading this book. Ultimately, we’ll wind up with a database of Python class instances, which can be browsed and changed from a variety of interfaces.

I’ll cover additional topics in this book, of course, but the techniques you will see here are representative of some of the domains we’ll explore later. And again, if you don’t completely understand the programs in this chapter, don’t worry because you shouldn’t—not yet anyway. This is just a Python demo. We’ll fill in the rest of the details soon enough. For now, let’s start off with a bit of fun.



Readers of the Fourth Edition of *Learning Python* might recognize some aspects of the running example used in this chapter—the characters here are similar in spirit to those in the OOP tutorial chapter in that book, and the later class-based examples here are essentially a variation on a theme. Despite some redundancy, I’m revisiting the example here for three reasons: it serves its purpose as a review of language fundamentals; some readers of this book haven’t read *Learning Python*; and the example receives expanded treatment here, with the addition of GUI and Web interfaces. That is, this chapter picks up where *Learning Python* left off, pushing this core language example into the realm of realistic applications—which, in a nutshell, reflects the purpose of this book.

The Task

Imagine, if you will, that you need to keep track of information about people for some reason. Maybe you want to store an address book on your computer, or perhaps you need to keep track of employees in a small business. For whatever reason, you want to write a program that keeps track of details about these people. In other words, you want to keep records in a database—to permanently store lists of people’s attributes on your computer.

Naturally, there are off-the-shelf programs for managing databases like these. By writing a program for this task yourself, however, you’ll have complete control over its operation. You can add code for special cases and behaviors that pre-coded software may not have anticipated. You won’t have to install and learn to use yet another database product. And you won’t be at the mercy of a software vendor to fix bugs or add new features. You decide to write a Python program to manage your people.

Step 1: Representing Records

If we’re going to store records in a database, the first step is probably deciding what those records will look like. There are a variety of ways to represent information about people in the Python language. Built-in object types such as lists and dictionaries are often sufficient, especially if we don’t initially care about processing the data we store.

Using Lists

Lists, for example, can collect attributes about people in a positionally ordered way. Start up your Python interactive interpreter and type the following two statements:

```
>>> bob = ['Bob Smith', 42, 30000, 'software']
>>> sue = ['Sue Jones', 45, 40000, 'hardware']
```

We've just made two records, albeit simple ones, to represent two people, Bob and Sue (my apologies if you really are Bob or Sue, generically or otherwise*). Each record is a list of four properties: name, age, pay, and job fields. To access these fields, we simply index by position; the result is in parentheses here because it is a tuple of two results:

```
>>> bob[0], sue[2]          # fetch name, pay  
('Bob Smith', 40000)
```

Processing records is easy with this representation; we just use list operations. For example, we can extract a last name by splitting the name field on blanks and grabbing the last part, and we can give someone a raise by changing their list in-place:

```
>>> bob[0].split()[-1]      # what's bob's last name?  
'Smith'  
>>> sue[2] *= 1.25        # give sue a 25% raise  
>>> sue  
['Sue Jones', 45, 50000.0, 'hardware']
```

The last-name expression here proceeds from left to right: we fetch Bob's name, split it into a list of substrings around spaces, and index his last name (run it one step at a time to see how).

Start-up pointers

Since this is the first code in this book, here are some quick pragmatic pointers for reference:

- This code may be typed in the IDLE GUI; after typing **python** at a shell prompt (or the full directory path to it if it's not on your system path); and so on.
- The **>>>** characters are Python's prompt (not code you type yourself).
- The informational lines that Python prints when this prompt starts up are usually omitted in this book to save space.
- I'm running all of this book's code under Python 3.1; results in any 3.X release should be similar (barring unforeseeable Python changes, of course).
- Apart from some system and C integration code, most of this book's examples are run under Windows 7, though thanks to Python portability, it generally doesn't matter unless stated otherwise.

If you've never run Python code this way before, see an introductory resource such as O'Reilly's *Learning Python* for help with getting started. I'll also have a few words to say about running code saved in script files later in this chapter.

* No, I'm serious. In the Python classes I teach, I had for many years regularly used the name "Bob Smith," age 40.5, and jobs "developer" and "manager" as a supposedly fictitious database record—until a class in Chicago, where I met a student named Bob Smith, who was 40.5 and was a developer and manager. The world is stranger than it seems.

A database list

Of course, what we've really coded so far is just two variables, not a database; to collect Bob and Sue into a unit, we might simply stuff them into another list:

```
>>> people = [bob, sue]                      # reference in list of lists
>>> for person in people:
    print(person)

['Bob Smith', 42, 30000, 'software']
['Sue Jones', 45, 50000.0, 'hardware']
```

Now the people list represents our database. We can fetch specific records by their relative positions and process them one at a time, in loops:

```
>>> people[1][0]
'Sue Jones'

>>> for person in people:
    print(person[0].split()[-1])           # print last names
    person[2] *= 1.20                     # give each a 20% raise

Smith
Jones

>>> for person in people: print(person[2])      # check new pay

36000.0
60000.0
```

Now that we have a list, we can also collect values from records using some of Python's more powerful iteration tools, such as list comprehensions, maps, and generator expressions:

```
>>> pays = [person[2] for person in people]    # collect all pay
>>> pays
[36000.0, 60000.0]

>>> pays = map((lambda x: x[2]), people)       # ditto (map is a generator in 3.X)
>>> list(pays)
[36000.0, 60000.0]

>>> sum(person[2] for person in people)        # generator expression, sum built-in
96000.0
```

To add a record to the database, the usual list operations, such as `append` and `extend`, will suffice:

```
>>> people.append(['Tom', 50, 0, None])
>>> len(people)
3
>>> people[-1][0]
'Tom'
```

Lists work for our people database, and they might be sufficient for some programs, but they suffer from a few major flaws. For one thing, Bob and Sue, at this point, are

just fleeting objects in memory that will disappear once we exit Python. For another, every time we want to extract a last name or give a raise, we'll have to repeat the kinds of code we just typed; that could become a problem if we ever change the way those operations work—we may have to update many places in our code. We'll address these issues in a few moments.

Field labels

Perhaps more fundamentally, accessing fields by position in a list requires us to memorize what each position means: if you see a bit of code indexing a record on magic position 2, how can you tell it is extracting a pay? In terms of understanding the code, it might be better to associate a field name with a field value.

We might try to associate names with relative positions by using the Python `range` built-in function, which generates successive integers when used in iteration contexts (such as the sequence assignment used initially here):

```
>>> NAME, AGE, PAY = range(3)          # 0, 1, and 2
>>> bob = ['Bob Smith', 42, 10000]
>>> bob[NAME]
'Bob Smith'
>>> PAY, bob[PAY]
(2, 10000)
```

This addresses readability: the three uppercase variables essentially become field names. This makes our code dependent on the field position assignments, though—we have to remember to update the range assignments whenever we change record structure. Because they are not directly associated, the names and records may become out of sync over time and require a maintenance step.

Moreover, because the field names are independent variables, there is no direct mapping from a record list back to its field's names. A raw record list, for instance, provides no way to label its values with field names in a formatted display. In the preceding record, without additional code, there is no path from value 42 to label AGE: `bob.index(42)` gives 1, the value of AGE, but not the name AGE itself.

We might also try this by using lists of tuples, where the tuples record both a field name and a value; better yet, a list of lists would allow for updates (tuples are immutable). Here's what that idea translates to, with slightly simpler records:

```
>>> bob = [['name', 'Bob Smith'], ['age', 42], ['pay', 10000]]
>>> sue = [['name', 'Sue Jones'], ['age', 45], ['pay', 20000]]
>>> people = [bob, sue]
```

This really doesn't fix the problem, though, because we still have to index by position in order to fetch fields:

```
>>> for person in people:
    print(person[0][1], person[2][1])      # name, pay
```

```

Bob Smith 10000
Sue Jones 20000

>>> [person[0][1] for person in people]      # collect names
['Bob Smith', 'Sue Jones']

>>> for person in people:
    print(person[0][1].split()[-1])          # get last names
    person[2][1] *= 1.10                      # give a 10% raise

Smith
Jones
>>> for person in people: print(person[2])

['pay', 11000.0]
['pay', 22000.0]

```

All we've really done here is add an extra level of positional indexing. To do better, we might inspect field names in loops to find the one we want (the loop uses tuple assignment here to unpack the name/value pairs):

```

>>> for person in people:
    for (name, value) in person:
        if name == 'name': print(value)    # find a specific field

Bob Smith
Sue Jones

```

Better yet, we can code a fetcher function to do the job for us:

```

>>> def field(record, label):
    for (fname, fvalue) in record:
        if fname == label:                 # find any field by name
            return fvalue

>>> field(bob, 'name')
'Bob Smith'
>>> field(sue, 'pay')
22000.0

>>> for rec in people:
    print(field(rec, 'age'))           # print all ages

42
45

```

If we proceed down this path, we'll eventually wind up with a set of record interface functions that generically map field names to field data. If you've done any Python coding in the past, though, you probably already know that there is an easier way to code this sort of association, and you can probably guess where we're headed in the next section.

Using Dictionaries

The list-based record representations in the prior section work, though not without some cost in terms of performance required to search for field names (assuming you need to care about milliseconds and such). But if you already know some Python, you also know that there are more efficient and convenient ways to associate property names and values. The built-in dictionary object is a natural:

```
>>> bob = {'name': 'Bob Smith', 'age': 42, 'pay': 30000, 'job': 'dev'}
>>> sue = {'name': 'Sue Jones', 'age': 45, 'pay': 40000, 'job': 'hdw'}
```

Now, Bob and Sue are objects that map field names to values automatically, and they make our code more understandable and meaningful. We don't have to remember what a numeric offset means, and we let Python search for the value associated with a field's name with its efficient dictionary indexing:

```
>>> bob['name'], sue['pay']           # not bob[0], sue[2]
('Bob Smith', 40000)

>>> bob['name'].split()[-1]
'Smith'

>>> sue['pay'] *= 1.10
>>> sue['pay']
44000.0
```

Because fields are accessed mnemonically now, they are more meaningful to those who read your code (including you).

Other ways to make dictionaries

Dictionaries turn out to be so useful in Python programming that there are even more convenient ways to code them than the traditional literal syntax shown earlier—e.g., with keyword arguments and the type constructor, as long as the keys are all strings:

```
>>> bob = dict(name='Bob Smith', age=42, pay=30000, job='dev')
>>> sue = dict(name='Sue Jones', age=45, pay=40000, job='hdw')
>>> bob
{'pay': 30000, 'job': 'dev', 'age': 42, 'name': 'Bob Smith'}
>>> sue
{'pay': 40000, 'job': 'hdw', 'age': 45, 'name': 'Sue Jones'}
```

by filling out a dictionary one field at a time (recall that dictionary keys are pseudo-randomly ordered):

```
>>> sue = {}
>>> sue['name'] = 'Sue Jones'
>>> sue['age'] = 45
>>> sue['pay'] = 40000
>>> sue['job'] = 'hdw'
>>> sue
{'job': 'hdw', 'pay': 40000, 'age': 45, 'name': 'Sue Jones'}
```

and by zipping together name/value lists:

```
>>> names = ['name', 'age', 'pay', 'job']
>>> values = ['Sue Jones', 45, 40000, 'hdw']
>>> list(zip(names, values))
[('name', 'Sue Jones'), ('age', 45), ('pay', 40000), ('job', 'hdw')]
>>> sue = dict(zip(names, values))
>>> sue
{'job': 'hdw', 'pay': 40000, 'age': 45, 'name': 'Sue Jones'}
```

We can even make dictionaries from a sequence of key values and an optional starting value for all the keys (handy to initialize an empty dictionary):

```
>>> fields = ('name', 'age', 'job', 'pay')
>>> record = dict.fromkeys(fields, '?')
>>> record
{'job': '?', 'pay': '?', 'age': '?', 'name': '?'}
```

Lists of dictionaries

Regardless of how we code them, we still need to collect our dictionary-based records into a database; a list does the trick again, as long as we don't require access by key at the top level:

```
>>> bob
{'pay': 30000, 'job': 'dev', 'age': 42, 'name': 'Bob Smith'}
>>> sue
{'job': 'hdw', 'pay': 40000, 'age': 45, 'name': 'Sue Jones'}

>>> people = [bob, sue]                                # reference in a list
>>> for person in people:
    print(person['name'], person['pay'], sep=', ')    # all name, pay

Bob Smith, 30000
Sue Jones, 40000

>>> for person in people:
    if person['name'] == 'Sue Jones':                  # fetch sue's pay
        print(person['pay'])

40000
```

Iteration tools work just as well here, but we use keys rather than obscure positions (in database terms, the list comprehension and map in the following code project the database on the "name" field column):

```
>>> names = [person['name'] for person in people]      # collect names
>>> names
['Bob Smith', 'Sue Jones']

>>> list(map((lambda x: x['name']), people))          # ditto, generate
['Bob Smith', 'Sue Jones']

>>> sum(person['pay'] for person in people)            # sum all pay
70000
```

Interestingly, tools such as list comprehensions and on-demand generator expressions can even approach the utility of SQL queries here, albeit operating on in-memory objects:

```
>>> [rec['name'] for rec in people if rec['age'] >= 45]    # SQL-ish query
['Sue Jones']

>>> [(rec['age'] ** 2 if rec['age'] >= 45 else rec['age']) for rec in people]
[42, 2025]

>>> G = (rec['name'] for rec in people if rec['age'] >= 45)
>>> next(G)
'Sue Jones'

>>> G = ((rec['age'] ** 2 if rec['age'] >= 45 else rec['age']) for rec in people)
>>> G.__next__()
42
```

And because dictionaries are normal Python objects, these records can also be accessed and updated with normal Python syntax:

```
>>> for person in people:
...     print(person['name'].split()[-1])                      # last name
...     person['pay'] *= 1.10                                    # a 10% raise

Smith
Jones

>>> for person in people: print(person['pay'])

33000.0
44000.0
```

Nested structures

Incidentally, we could avoid the last-name extraction code in the prior examples by further structuring our records. Because all of Python’s compound datatypes can be nested inside each other and as deeply as we like, we can build up fairly complex information structures easily—simply type the object’s syntax, and Python does all the work of building the components, linking memory structures, and later reclaiming their space. This is one of the great advantages of a scripting language such as Python.

The following, for instance, represents a more structured record by nesting a dictionary, list, and tuple inside another dictionary:

```
>>> bob2 = {'name': {'first': 'Bob', 'last': 'Smith'},
...           'age': 42,
...           'job': ['software', 'writing'],
...           'pay': (40000, 50000)}
```

Because this record contains nested structures, we simply index twice to go two levels deep:

```
>>> bob2['name']                                # bob's full name
{'last': 'Smith', 'first': 'Bob'}
>>> bob2['name']['last']                         # bob's last name
'Smith'
>>> bob2['pay'][1]                               # bob's upper pay
50000
```

The name field is another dictionary here, so instead of splitting up a string, we simply index to fetch the last name. Moreover, people can have many jobs, as well as minimum and maximum pay limits. In fact, Python becomes a sort of query language in such cases—we can fetch or change nested data with the usual object operations:

```
>>> for job in bob2['job']: print(job)          # all of bob's jobs
software
writing

>> bob2['job'][-1]                             # bob's last job
'writing'
>>> bob2['job'].append('janitor')              # bob gets a new job
>>> bob2
{'job': ['software', 'writing', 'janitor'], 'pay': (40000, 50000), 'age': 42, 'name':
{'last': 'Smith', 'first': 'Bob'}}
```

It's OK to grow the nested list with `append`, because it is really an independent object. Such nesting can come in handy for more sophisticated applications; to keep ours simple, we'll stick to the original flat record structure.

Dictionaries of dictionaries

One last twist on our people database: we can get a little more mileage out of dictionaries here by using one to represent the database itself. That is, we can use a dictionary of dictionaries—the outer dictionary is the database, and the nested dictionaries are the records within it. Rather than a simple list of records, a dictionary-based database allows us to store and retrieve records by symbolic key:

```
>>> bob = dict(name='Bob Smith', age=42, pay=30000, job='dev')
>>> sue = dict(name='Sue Jones', age=45, pay=40000, job='hdw')
>>> bob
{'pay': 30000, 'job': 'dev', 'age': 42, 'name': 'Bob Smith'}

>>> db = {}
>>> db['bob'] = bob                           # reference in a dict of dicts
>>> db['sue'] = sue
>>>
>>> db['bob']['name']                         # fetch bob's name
'Bob Smith'
>>> db['sue']['pay'] = 50000                  # change sue's pay
>>> db['sue']['pay']                          # fetch sue's pay
50000
```

Notice how this structure allows us to access a record directly instead of searching for it in a loop—we get to Bob’s name immediately by indexing on key `bob`. This really is a dictionary of dictionaries, though you won’t see all the gory details unless you display the database all at once (the Python `pprint` pretty-printer module can help with legibility here):

```
>>> db
{'bob': {'pay': 30000, 'job': 'dev', 'age': 42, 'name': 'Bob Smith'}, 'sue':
{'pay': 50000, 'job': 'hdw', 'age': 45, 'name': 'Sue Jones'}}

>>> import pprint
>>> pprint.pprint(db)
{'bob': {'age': 42, 'job': 'dev', 'name': 'Bob Smith', 'pay': 30000},
 'sue': {'age': 45, 'job': 'hdw', 'name': 'Sue Jones', 'pay': 50000}}
```

If we still need to step through the database one record at a time, we can now rely on dictionary iterators. In recent Python releases, a dictionary iterator produces one key in a `for` loop each time through (for compatibility with earlier releases, we can also call the `db.keys` method explicitly in the `for` loop rather than saying just `db`, but since Python 3’s `keys` result is a generator, the effect is roughly the same):

```
>>> for key in db:
    print(key, '=>', db[key]['name'])

bob => Bob Smith
sue => Sue Jones

>>> for key in db:
    print(key, '=>', db[key]['pay'])

bob => 30000
sue => 50000
```

To visit all records, either index by key as you go:

```
>>> for key in db:
    print(db[key]['name'].split()[-1])
    db[key]['pay'] *= 1.10
```

```
Smith
Jones
```

or step through the dictionary’s values to access records directly:

```
>>> for record in db.values(): print(record['pay'])

33000.0
55000.0

>>> x = [db[key]['name'] for key in db]
>>> x
['Bob Smith', 'Sue Jones']

>>> x = [rec['name'] for rec in db.values()]
```

```
>>> x
['Bob Smith', 'Sue Jones']
```

And to add a new record, simply assign it to a new key; this is just a dictionary, after all:

```
>>> db['tom'] = dict(name='Tom', age=50, job=None, pay=0)
>>>
>>> db['tom']
{'pay': 0, 'job': None, 'age': 50, 'name': 'Tom'}
>>> db['tom']['name']
'Tom'
>>> list(db.keys())
['bob', 'sue', 'tom']
>>> len(db)
3
>>> [rec['age'] for rec in db.values()]
[42, 45, 50]
>>> [rec['name'] for rec in db.values() if rec['age'] >= 45]      # SQL-ish query
['Sue Jones', 'Tom']
```

Although our database is still a transient object in memory, it turns out that this dictionary-of-dictionaries format corresponds exactly to a system that saves objects permanently—the shelf (yes, this should probably be *shelf*, grammatically speaking, but the Python module name and term is *shelve*). To learn how, let’s move on to the next section.

Step 2: Storing Records Persistently

So far, we’ve settled on a dictionary-based representation for our database of records, and we’ve reviewed some Python data structure concepts along the way. As mentioned, though, the objects we’ve seen so far are temporary—they live in memory and they go away as soon as we exit Python or the Python program that created them. To make our people persistent, they need to be stored in a file of some sort.

Using Formatted Files

One way to keep our data around between program runs is to write all the data out to a simple text file, in a formatted way. Provided the saving and loading tools agree on the format selected, we’re free to use any custom scheme we like.

Test data script

So that we don’t have to keep working interactively, let’s first write a script that initializes the data we are going to store (if you’ve done any Python work in the past, you know that the interactive prompt tends to become tedious once you leave the realm of simple one-liners). [Example 1-1](#) creates the sort of records and database dictionary we’ve been working with so far, but because it is a module, we can import it repeatedly without having to retype the code each time. In a sense, this module is a database itself, but its program code format doesn’t support automatic or end-user updates as is.

```

Example 1-1. PP4E\Preview\initdata.py

# initialize data to be stored in files, pickles, shelves

# records
bob = {'name': 'Bob Smith', 'age': 42, 'pay': 30000, 'job': 'dev'}
sue = {'name': 'Sue Jones', 'age': 45, 'pay': 40000, 'job': 'hdw'}
tom = {'name': 'Tom', 'age': 50, 'pay': 0, 'job': None}

# database
db = {}
db['bob'] = bob
db['sue'] = sue
db['tom'] = tom

if __name__ == '__main__':      # when run as a script
    for key in db:
        print(key, '=>\n  ', db[key])

```

As usual, the `__name__` test at the bottom of Example 1-1 is true only when this file is run, not when it is imported. When run as a top-level script (e.g., from a command line, via an icon click, or within the IDLE GUI), the file's self-test code under this test dumps the database's contents to the standard output stream (remember, that's what `print` function-call statements do by default).

Here is the script in action being run from a system command line on Windows. Type the following command in a Command Prompt window after a `cd` to the directory where the file is stored, and use a similar console window on other types of computers:

```

...\\PP4E\\Preview> python initdata.py
bob =>
    {'job': 'dev', 'pay': 30000, 'age': 42, 'name': 'Bob Smith'}
sue =>
    {'job': 'hdw', 'pay': 40000, 'age': 45, 'name': 'Sue Jones'}
tom =>
    {'job': None, 'pay': 0, 'age': 50, 'name': 'Tom'}

```

File name conventions

Since this is our first source file (a.k.a. “script”), here are three usage notes for this book's examples:

- The text `...\\PP4E\\Preview>` in the first line of the preceding example listing stands for your operating system's prompt, which can vary per platform; you type just the text that follows this prompt (`python initdata.py`).
- Like all examples in this book, the system prompt also gives the directory in the downloadable book examples package where this command should be run. When running this script using a command-line in a system shell, make sure the shell's current working directory is `PP4E\\Preview`. This can matter for examples that use files in the working directory.

- Similarly, the label that precedes every example file's code listing tells you where the source file resides in the examples package. Per the [Example 1-1](#) listing label shown earlier, this script's full filename is *PP4E\Preview\initdata.py* in the examples tree.

We'll use these conventions throughout the book; see the Preface for more on getting the examples if you wish to work along. I occasionally give more of the directory path in system prompts when it's useful to provide the extra execution context, especially in the system part of the book (e.g., a "C:\" prefix from Windows or more directory names).

Script start-up pointers

I gave pointers for using the interactive prompt earlier. Now that we've started running script files, here are also a few quick startup pointers for using Python scripts in general:

- On some platforms, you may need to type the full directory path to the Python program on your machine; if Python isn't on your system path setting on Windows, for example, replace `python` in the command with `C:\Python31\python` (this assumes you're using Python 3.1).
- On most Windows systems you also don't need to type `python` on the command line at all; just type the file's name to run it, since Python is registered to open ".py" script files.
- You can also run this file inside Python's standard IDLE GUI (open the file and use the Run menu in the text edit window), and in similar ways from any of the available third-party Python IDEs (e.g., Komodo, Eclipse, NetBeans, and the Wing IDE).
- If you click the program's file icon to launch it on Windows, be sure to add an `input()` call to the bottom of the script to keep the output window up. On other systems, icon clicks may require a `#!` line at the top and executable permission via a `chmod` command.

I'll assume here that you're able to run Python code one way or another. Again, if you're stuck, see other books such as [Learning Python](#) for the full story on launching Python programs.

Data format script

Now, all we have to do is store all of this in-memory data in a file. There are a variety of ways to accomplish this; one of the most basic is to write one piece of data at a time, with separators between each that we can use when reloading to break the data apart. [Example 1-2](#) shows one way to code this idea.

Example 1-2. PP4E\Preview\make_db_file.py

```
"""
Save in-memory database object to a file with custom formatting;
assume 'endrec.', 'enddb.', and '=>' are not used in the data;
assume db is dict of dict; warning: eval can be dangerous - it
runs strings as code; could also eval() record dict all at once;
could also dbfile.write(key + '\n') vs print(key, file=dbfile);
"""

dbfilename = 'people-file'
ENDDB = 'enddb.'
ENDREC = 'endrec.'
RECSEP = '=>'

def storeDbase(db, dbfilename=dbfilename):
    "formatted dump of database to flat file"
    dbfile = open(dbfilename, 'w')
    for key in db:
        print(key, file=dbfile)
        for (name, value) in db[key].items():
            print(name + RECSEP + repr(value), file=dbfile)
        print(ENDREC, file=dbfile)
    print(ENDDB, file=dbfile)
    dbfile.close()

def loadDbase(dbfilename=dbfilename):
    "parse data to reconstruct database"
    dbfile = open(dbfilename)
    import sys
    sys.stdin = dbfile
    db = {}
    key = input()
    while key != ENddb:
        rec = {}
        field = input()
        while field != ENDREC:
            name, value = field.split(RECSEP)
            rec[name] = eval(value)
            field = input()
        db[key] = rec
        key = input()
    return db

if __name__ == '__main__':
    from initdata import db
    storeDbase(db)
```

This is a somewhat complex program, partly because it has both saving and loading logic and partly because it does its job the hard way; as we'll see in a moment, there are better ways to get objects into files than by manually formatting and parsing them. For simple tasks, though, this does work; running [Example 1-2](#) as a script writes the database out to a flat file. It has no printed output, but we can inspect the database file interactively after this script is run, either within IDLE or from a console window where

you're running these examples (as is, the database file shows up in the current working directory):

```
...\\PP4E\\Preview> python make_db_file.py
...\\PP4E\\Preview> python
>>> for line in open('people-file'):
...     print(line, end='')
...
bob
job=>'dev'
pay=>30000
age=>42
name=>'Bob Smith'
endrec.
sue
job=>'hdw'
pay=>40000
age=>45
name=>'Sue Jones'
endrec.
tom
job=>None
pay=>0
age=>50
name=>'Tom'
endrec.
enddb.
```

This file is simply our database's content with added formatting. Its data originates from the test data initialization module we wrote in [Example 1-1](#) because that is the module from which [Example 1-2](#)'s self-test code imports its data. In practice, [Example 1-2](#) itself could be imported and used to store a variety of databases and files.

Notice how data to be written is formatted with the as-code `repr` call and is re-created with the `eval` call, which treats strings as Python code. That allows us to store and re-create things like the `None` object, but it is potentially unsafe; you shouldn't use `eval` if you can't be sure that the database won't contain malicious code. For our purposes, however, there's probably no cause for alarm.

Utility scripts

To test further, [Example 1-3](#) reloads the database from a file each time it is run.

Example 1-3. PP4E\\Preview\\dump_db_file.py

```
from make_db_file import loadDbase
db = loadDbase()
for key in db:
    print(key, '=>\n  ', db[key])
print(db['sue']['name'])
```

And [Example 1-4](#) makes changes by loading, updating, and storing again.

Example 1-4. PP4E\Preview\update_db_file.py

```
from make_db_file import loadDbase, storeDbase
db = loadDbase()
db['sue']['pay'] *= 1.10
db['tom']['name'] = 'Tom Tom'
storeDbase(db)
```

Here are the dump script and the update script in action at a system command line; both Sue's pay and Tom's name change between script runs. The main point to notice is that the data stays around after each script exits—our objects have become persistent simply because they are mapped to and from text files:

```
...\\PP4E\\Preview> python dump_db_file.py
bob =>
    {'pay': 30000, 'job': 'dev', 'age': 42, 'name': 'Bob Smith'}
sue =>
    {'pay': 40000, 'job': 'hdw', 'age': 45, 'name': 'Sue Jones'}
tom =>
    {'pay': 0, 'job': None, 'age': 50, 'name': 'Tom'}
Sue Jones

...\\PP4E\\Preview> python update_db_file.py
...\\PP4E\\Preview> python dump_db_file.py
bob =>
    {'pay': 30000, 'job': 'dev', 'age': 42, 'name': 'Bob Smith'}
sue =>
    {'pay': 44000.0, 'job': 'hdw', 'age': 45, 'name': 'Sue Jones'}
tom =>
    {'pay': 0, 'job': None, 'age': 50, 'name': 'Tom Tom'}
Sue Jones
```

As is, we'll have to write Python code in scripts or at the interactive command line for each specific database update we need to perform (later in this chapter, we'll do better by providing generalized console, GUI, and web-based interfaces instead). But at a basic level, our text file is a database of records. As we'll learn in the next section, though, it turns out that we've just done a lot of pointless work.

Using Pickle Files

The formatted text file scheme of the prior section works, but it has some major limitations. For one thing, it has to read the entire database from the file just to fetch one record, and it must write the entire database back to the file after each set of updates. Although storing one record's text per file would work around this limitation, it would also complicate the program further.

For another thing, the text file approach assumes that the data separators it writes out to the file will not appear in the data to be stored: if the characters => happen to appear in the data, for example, the scheme will fail. We might work around this by generating XML text to represent records in the text file, using Python's XML parsing tools, which we'll meet later in this text, to reload; XML tags would avoid collisions with actual

data's text, but creating and parsing XML would complicate the program substantially too.

Perhaps worst of all, the formatted text file scheme is already complex without being general: it is tied to the dictionary-of-dictionaries structure, and it can't handle anything else without being greatly expanded. It would be nice if a general tool existed that could translate any sort of Python data to a format that could be saved in a file in a single step.

That is exactly what the Python `pickle` module is designed to do. The `pickle` module translates an in-memory Python object into a *serialized* byte stream—a string of bytes that can be written to any file-like object. The `pickle` module also knows how to reconstruct the original object in memory, given the serialized byte stream: we get back the exact same object. In a sense, the `pickle` module replaces proprietary data formats—its serialized format is general and efficient enough for any program. With `pickle`, there is no need to manually translate objects to data when storing them persistently, and no need to manually parse a complex format to get them back. Pickling is similar in spirit to XML representations, but it's both more Python-specific, and much simpler to code.

The net effect is that pickling allows us to store and fetch native Python objects as they are and in a single step—we use normal Python syntax to process pickled records. Despite what it does, the `pickle` module is remarkably easy to use. [Example 1-5](#) shows how to store our records in a flat file, using `pickle`.

Example 1-5. PP4E\Preview\make_db_pickle.py

```
from initdata import db
import pickle
dbfile = open('people-pickle', 'wb')           # use binary mode files in 3.X
pickle.dump(db, dbfile)                         # data is bytes, not str
dbfile.close()
```

When run, this script stores the entire database (the dictionary of dictionaries defined in [Example 1-1](#)) to a flat file named *people-pickle* in the current working directory. The `pickle` module handles the work of converting the object to a string. [Example 1-6](#) shows how to access the pickled database after it has been created; we simply open the file and pass its content back to `pickle` to remake the object from its serialized string.

Example 1-6. PP4E\Preview\dump_db_pickle.py

```
import pickle
dbfile = open('people-pickle', 'rb')           # use binary mode files in 3.X
db = pickle.load(dbfile)
for key in db:
    print(key, '=>\n  ', db[key])
print(db['sue']['name'])
```

Here are these two scripts at work, at the system command line again; naturally, they can also be run in IDLE, and you can open and inspect the pickle file by running the same sort of code interactively as well:

```
...\\PP4E\\Preview> python make_db_pickle.py
...\\PP4E\\Preview> python dump_db_pickle.py
bob =>
    {'pay': 30000, 'job': 'dev', 'age': 42, 'name': 'Bob Smith'}
sue =>
    {'pay': 40000, 'job': 'hdw', 'age': 45, 'name': 'Sue Jones'}
tom =>
    {'pay': 0, 'job': None, 'age': 50, 'name': 'Tom'}
Sue Jones
```

Updating with a pickle file is similar to a manually formatted file, except that Python is doing all of the formatting work for us. [Example 1-7](#) shows how.

Example 1-7. PP4E\\Preview\\update-db-pickle.py

```
import pickle
dbfile = open('people-pickle', 'rb')
db = pickle.load(dbfile)
dbfile.close()

db['sue']['pay'] *= 1.10
db['tom']['name'] = 'Tom Tom'

dbfile = open('people-pickle', 'wb')
pickle.dump(db, dbfile)
dbfile.close()
```

Notice how the entire database is written back to the file after the records are changed in memory, just as for the manually formatted approach; this might become slow for very large databases, but we'll ignore this for the moment. Here are our update and dump scripts in action—as in the prior section, Sue's pay and Tom's name change between scripts because they are written back to a file (this time, a pickle file):

```
...\\PP4E\\Preview> python update_db_pickle.py
...\\PP4E\\Preview> python dump_db_pickle.py
bob =>
    {'pay': 30000, 'job': 'dev', 'age': 42, 'name': 'Bob Smith'}
sue =>
    {'pay': 44000.0, 'job': 'hdw', 'age': 45, 'name': 'Sue Jones'}
tom =>
    {'pay': 0, 'job': None, 'age': 50, 'name': 'Tom Tom'}
Sue Jones
```

As we'll learn in [Chapter 17](#), the Python pickling system supports nearly arbitrary object types—lists, dictionaries, class instances, nested structures, and more. There, we'll also learn about the pickler's text and binary storage protocols; as of Python 3, all protocols use bytes objects to represent pickled data, which in turn requires pickle files to be opened in binary mode for all protocols. As we'll see later in this chapter, the pickler and its data format also underlie shelves and ZODB databases, and pickled class instances provide both data and behavior for objects stored.

In fact, pickling is more general than these examples may imply. Because they accept any object that provides an interface compatible with files, pickling and unpickling may

be used to transfer native Python objects to a variety of media. Using a network socket, for instance, allows us to ship pickled Python objects across a network and provides an alternative to larger protocols such as SOAP and XML-RPC.

Using Per-Record Pickle Files

As mentioned earlier, one potential disadvantage of this section's examples so far is that they may become slow for very large databases: because the entire database must be loaded and rewritten to update a single record, this approach can waste time. We could improve on this by storing each record in the database in a separate flat file. The next three examples show one way to do so; [Example 1-8](#) stores each record in its own flat file, using each record's original key as its filename with a *.pkl* appended (it creates the files *bob.pkl*, *sue.pkl*, and *tom.pkl* in the current working directory).

Example 1-8. PP4E\Preview\make_db_pickle_recs.py

```
from initdata import bob, sue, tom
import pickle
for (key, record) in [('bob', bob), ('tom', tom), ('sue', sue)]:
    recfile = open(key + '.pkl', 'wb')
    pickle.dump(record, recfile)
    recfile.close()
```

Next, [Example 1-9](#) dumps the entire database by using the standard library's `glob` module to do filename expansion and thus collect all the files in this directory with a *.pkl* extension. To load a single record, we open its file and deserialize with `pickle`; we must load only one record file, though, not the entire database, to fetch one record.

Example 1-9. PP4E\Preview\dump_db_pickle_recs.py

```
import pickle, glob
for filename in glob.glob('*.pkl'):           # for 'bob','sue','tom'
    recfile = open(filename, 'rb')
    record = pickle.load(recfile)
    print(filename, '=>\n ', record)

suefile = open('sue.pkl', 'rb')
print(pickle.load(suefile)['name'])          # fetch sue's name
```

Finally, [Example 1-10](#) updates the database by fetching a record from its file, changing it in memory, and then writing it back to its pickle file. This time, we have to fetch and rewrite only a single record file, not the full database, to update.

Example 1-10. PP4E\Preview\update_db_pickle_recs.py

```
import pickle
suefile = open('sue.pkl', 'rb')
sue = pickle.load(suefile)
suefile.close()
```

```
sue['pay'] *= 1.10
suefile = open('sue.pkl', 'wb')
pickle.dump(sue, suefile)
suefile.close()
```

Here are our file-per-record scripts in action; the results are about the same as in the prior section, but database keys become real filenames now. In a sense, the filesystem becomes our top-level dictionary—filenames provide direct access to each record.

```
...\\PP4E\\Preview> python make_db_pickle_recs.py
...\\PP4E\\Preview> python dump_db_pickle_recs.py
bob.pkl =>
    {'pay': 30000, 'job': 'dev', 'age': 42, 'name': 'Bob Smith'}
sue.pkl =>
    {'pay': 40000, 'job': 'hdw', 'age': 45, 'name': 'Sue Jones'}
tom.pkl =>
    {'pay': 0, 'job': None, 'age': 50, 'name': 'Tom'}
Sue Jones

...\\PP4E\\Preview> python update_db_pickle_recs.py
...\\PP4E\\Preview> python dump_db_pickle_recs.py
bob.pkl =>
    {'pay': 30000, 'job': 'dev', 'age': 42, 'name': 'Bob Smith'}
sue.pkl =>
    {'pay': 44000.0, 'job': 'hdw', 'age': 45, 'name': 'Sue Jones'}
tom.pkl =>
    {'pay': 0, 'job': None, 'age': 50, 'name': 'Tom'}
Sue Jones
```

Using Shelves

Pickling objects to files, as shown in the preceding section, is an optimal scheme in many applications. In fact, some applications use pickling of Python objects across network sockets as a simpler alternative to network protocols such as the SOAP and XML-RPC web services architectures (also supported by Python, but much heavier than `pickle`).

Moreover, assuming your filesystem can handle as many files as you'll need, pickling one record per file also obviates the need to load and store the entire database for each update. If we really want keyed access to records, though, the Python standard library offers an even higher-level tool: shelves.

Shelves automatically pickle objects to and from a keyed-access filesystem. They behave much like dictionaries that must be opened, and they persist after each program exits. Because they give us key-based access to stored records, there is no need to manually manage one flat file per record—the shelf system automatically splits up stored records and fetches and updates only those records that are accessed and changed. In this way, shelves provide utility similar to per-record pickle files, but they are usually easier to code.

The `shelve` interface is just as simple as `pickle`: it is identical to dictionaries, with extra open and close calls. In fact, to your code, a shelve really does appear to be a persistent dictionary of persistent objects; Python does all the work of mapping its content to and from a file. For instance, [Example 1-11](#) shows how to store our in-memory dictionary objects in a shelve for permanent keeping.

Example 1-11. PP4E\Preview\make_db_shelve.py

```
from initdata import bob, sue
import shelve
db = shelve.open('people-shelve')
db['bob'] = bob
db['sue'] = sue
db.close()
```

This script creates one or more files in the current directory with the name *people-shelve* as a prefix (in Python 3.1 on Windows, *people-shelve.bak*, *people-shelve.dat*, and *people-shelve.dir*). You shouldn't delete these files (they are your database!), and you should be sure to use the same base name in other scripts that access the shelve. [Example 1-12](#), for instance, reopens the shelve and indexes it by key to fetch its stored records.

Example 1-12. PP4E\Preview\dump_db_shelve.py

```
import shelve
db = shelve.open('people-shelve')
for key in db:
    print(key, '=>\n  ', db[key])
print(db['sue']['name'])
db.close()
```

We still have a dictionary of dictionaries here, but the top-level dictionary is really a shelve mapped onto a file. Much happens when you access a shelve's keys—it uses `pickle` internally to serialize and deserialize objects stored, and it interfaces with a keyed-access filesystem. From your perspective, though, it's just a persistent dictionary. [Example 1-13](#) shows how to code shelve updates.

Example 1-13. PP4E\Preview\update_db_shelve.py

```
from initdata import tom
import shelve
db = shelve.open('people-shelve')
sue = db['sue']                      # fetch sue
sue['pay'] *= 1.50
db['sue'] = sue                       # update sue
db['tom'] = tom                        # add a new record
db.close()
```

Notice how this code fetches `sue` by key, updates in memory, and then reassigned to the key to update the shelve; this is a requirement of shelves by default, but not always of more advanced shelve-like systems such as ZODB, covered in [Chapter 17](#). As we'll see

later, `shelve.open` also has a newer `writeback` keyword argument, which, if passed `True`, causes all records loaded from the shelve to be cached in memory, and automatically written back to the shelve when it is closed; this avoids manual write backs on changes, but can consume memory and make closing slow.

Also note how shelve files are explicitly closed. Although we don't need to pass mode flags to `shelve.open` (by default it creates the shelve if needed, and opens it for reads and writes otherwise), some underlying keyed-access filesystems may require a `close` call in order to flush output buffers after changes.

Finally, here are the shelve-based scripts on the job, creating, changing, and fetching records. The records are still dictionaries, but the database is now a dictionary-like shelve which automatically retains its state in a file between program runs:

```
...\\PP4E\\Preview> python make_db_shelve.py
...\\PP4E\\Preview> python dump_db_shelve.py
bob =>
    {'pay': 30000, 'job': 'dev', 'age': 42, 'name': 'Bob Smith'}
sue =>
    {'pay': 40000, 'job': 'hdw', 'age': 45, 'name': 'Sue Jones'}
Sue Jones

...\\PP4E\\Preview> python update_db_shelve.py
...\\PP4E\\Preview> python dump_db_shelve.py
bob =>
    {'pay': 30000, 'job': 'dev', 'age': 42, 'name': 'Bob Smith'}
sue =>
    {'pay': 60000.0, 'job': 'hdw', 'age': 45, 'name': 'Sue Jones'}
tom =>
    {'pay': 0, 'job': None, 'age': 50, 'name': 'Tom'}
Sue Jones
```

When we ran the update and dump scripts here, we added a new record for key `tom` and increased Sue's pay field by 50 percent. These changes are permanent because the record dictionaries are mapped to an external file by `shelve`. (In fact, this is a particularly good script for Sue—something she might consider scheduling to run often, using a cron job on Unix, or a Startup folder or msconfig entry on Windows...)

What's in a Name?

Though it's a surprisingly well-kept secret, Python gets its name from the 1970s British TV comedy series *Monty Python's Flying Circus*. According to Python folklore, Guido van Rossum, Python's creator, was watching reruns of the show at about the same time he needed a name for a new language he was developing. And as they say in show business, "the rest is history."

Because of this heritage, references to the comedy group's work often show up in examples and discussion. For instance, the name *Brian* appears often in scripts; the words *spam*, *lumberjack*, and *shrubbery* have a special connotation to Python users; and presentations are sometimes referred to as *The Spanish Inquisition*. As a rule, if a Python user starts using phrases that have no relation to reality, they're probably borrowed

from the Monty Python series or movies. Some of these phrases might even pop up in this book. You don't have to run out and rent *The Meaning of Life* or *The Holy Grail* to do useful work in Python, of course, but it can't hurt.

While "Python" turned out to be a distinctive name, it has also had some interesting side effects. For instance, when the Python newsgroup, comp.lang.python, came online in 1994, its first few weeks of activity were almost entirely taken up by people wanting to discuss topics from the TV show. More recently, a special Python supplement in the *Linux Journal* magazine featured photos of Guido garbed in an obligatory "nice red uniform."

Python's news list still receives an occasional post from fans of the show. For instance, one early poster innocently offered to swap Monty Python scripts with other fans. Had he known the nature of the forum, he might have at least mentioned whether they were portable or not.

Step 3: Stepping Up to OOP

Let's step back for a moment and consider how far we've come. At this point, we've created a database of records: the shelve, as well as per-record pickle file approaches of the prior section suffice for basic data storage tasks. As is, our records are represented as simple dictionaries, which provide easier-to-understand access to fields than do lists (by key, rather than by position). Dictionaries, however, still have some limitations that may become more critical as our program grows over time.

For one thing, there is no central place for us to collect record processing logic. Extracting last names and giving raises, for instance, can be accomplished with code like the following:

```
>>> import shelve
>>> db = shelve.open('people-shelve')
>>> bob = db['bob']
>>> bob['name'].split()[-1]           # get bob's last name
'Smith'
>>> sue = db['sue']
>>> sue['pay'] *= 1.25             # give sue a raise
>>> sue['pay']
75000.0
>>> db['sue'] = sue
>>> db.close()
```

This works, and it might suffice for some short programs. But if we ever need to change the way last names and raises are implemented, we might have to update this kind of code in many places in our program. In fact, even finding all such magical code snippets could be a challenge; hardcoding or cutting and pasting bits of logic redundantly like this in more than one place will almost always come back to haunt you eventually.

It would be better to somehow hide—that is, *encapsulate*—such bits of code. Functions in a module would allow us to implement such operations in a single place and thus

avoid code redundancy, but still wouldn't naturally associate them with the records themselves. What we'd like is a way to bind processing logic with the data stored in the database in order to make it easier to understand, debug, and reuse.

Another downside to using dictionaries for records is that they are difficult to expand over time. For example, suppose that the set of data fields or the procedure for giving raises is different for different kinds of people (perhaps some people get a bonus each year and some do not). If we ever need to extend our program, there is no natural way to customize simple dictionaries. For future growth, we'd also like our software to support extension and customization in a natural way.

If you've already studied Python in any sort of depth, you probably already know that this is where its OOP support begins to become attractive:

Structure

With OOP, we can naturally associate processing logic with record data—classes provide both a program unit that combines logic and data in a single package and a hierarchy that allows code to be easily factored to avoid redundancy.

Encapsulation

With OOP, we can also wrap up details such as name processing and pay increases behind method functions—i.e., we are free to change method implementations without breaking their users.

Customization

And with OOP, we have a natural growth path. Classes can be extended and customized by coding new subclasses, without changing or breaking already working code.

That is, under OOP, we program by customizing and reusing, not by rewriting. OOP is an option in Python and, frankly, is sometimes better suited for strategic than for tactical tasks. It tends to work best when you have time for upfront planning—something that might be a luxury if your users have already begun storming the gates.

But especially for larger systems that change over time, its code reuse and structuring advantages far outweigh its learning curve, and it can substantially cut development time. Even in our simple case, the customizability and reduced redundancy we gain from classes can be a decided advantage.

Using Classes

OOP is easy to use in Python, thanks largely to Python's dynamic typing model. In fact, it's so easy that we'll jump right into an example: [Example 1-14](#) implements our database records as class instances rather than as dictionaries.

Example 1-14. PP4E\Preview\person_start.py

```
class Person:  
    def __init__(self, name, age, pay=0, job=None):
```

```

self.name = name
self.age = age
self.pay = pay
self.job = job

if __name__ == '__main__':
    bob = Person('Bob Smith', 42, 30000, 'software')
    sue = Person('Sue Jones', 45, 40000, 'hardware')
    print(bob.name, sue.pay)

    print(bob.name.split()[-1])
    sue.pay *= 1.10
    print(sue.pay)

```

There is not much to this class—just a constructor method that fills out the instance with data passed in as arguments to the class name. It's sufficient to represent a database record, though, and it can already provide tools such as defaults for pay and job fields that dictionaries cannot. The self-test code at the bottom of this file creates two instances (records) and accesses their attributes (fields); here is this file's output when run under IDLE (a system command-line works just as well):

```

Bob Smith 40000
Smith
44000.0

```

This isn't a database yet, but we could stuff these objects into a list or dictionary as before in order to collect them as a unit:

```

>>> from person_start import Person
>>> bob = Person('Bob Smith', 42)
>>> sue = Person('Sue Jones', 45, 40000)

>>> people = [bob, sue]                                # a "database" list
>>> for person in people:
...     print(person.name, person.pay)

Bob Smith 0
Sue Jones 40000

>>> x = [(person.name, person.pay) for person in people]
>>> x
[('Bob Smith', 0), ('Sue Jones', 40000)]

>>> [rec.name for rec in people if rec.age >= 45]      # SQL-ish query
['Sue Jones']

>>> [(rec.age ** 2 if rec.age >= 45 else rec.age) for rec in people]
[42, 2025]

```

Notice that Bob's pay defaulted to zero this time because we didn't pass in a value for that argument (maybe Sue is supporting him now?). We might also implement a class that represents the database, perhaps as a subclass of the built-in list or dictionary types, with insert and delete methods that encapsulate the way the database is implemented. We'll abandon this path for now, though, because it will be more useful to store these

records persistently in a shelve, which already encapsulates stores and fetches behind an interface for us. Before we do, though, let's add some logic.

Adding Behavior

So far, our class is just data: it replaces dictionary keys with object attributes, but it doesn't add much to what we had before. To really leverage the power of classes, we need to add some behavior. By wrapping up bits of behavior in class method functions, we can insulate clients from changes. And by packaging methods in classes along with data, we provide a natural place for readers to look for code. In a sense, classes combine records and the programs that process those records; methods provide logic that interprets and updates the data (we say they are *object-oriented*, because they always process an object's data).

For instance, [Example 1-15](#) adds the last-name and raise logic as class methods; methods use the `self` argument to access or update the instance (record) being processed.

Example 1-15. PP4E\Preview\person.py

```
class Person:
    def __init__(self, name, age, pay=0, job=None):
        self.name = name
        self.age = age
        self.pay = pay
        self.job = job
    def lastName(self):
        return self.name.split()[-1]
    def giveRaise(self, percent):
        self.pay *= (1.0 + percent)

if __name__ == '__main__':
    bob = Person('Bob Smith', 42, 30000, 'software')
    sue = Person('Sue Jones', 45, 40000, 'hardware')
    print(bob.name, sue.pay)

    print(bob.lastName())
    sue.giveRaise(.10)
    print(sue.pay)
```

The output of this script is the same as the last, but the results are being computed by methods now, not by hardcoded logic that appears redundantly wherever it is required:

```
Bob Smith 40000
Smith
44000.0
```

Adding Inheritance

One last enhancement to our records before they become permanent: because they are implemented as classes now, they naturally support customization through the inheritance search mechanism in Python. [Example 1-16](#), for instance, customizes the last

section's `Person` class in order to give a 10 percent bonus by default to managers whenever they receive a raise (any relation to practice in the real world is purely coincidental).

Example 1-16. PP4E\Preview\manager.py

```
from person import Person

class Manager(Person):
    def giveRaise(self, percent, bonus=0.1):
        self.pay *= (1.0 + percent + bonus)

if __name__ == '__main__':
    tom = Manager(name='Tom Doe', age=50, pay=50000)
    print(tom.lastName())
    tom.giveRaise(.20)
    print(tom.pay)
```

When run, this script's self-test prints the following:

```
Doe
65000.0
```

Here, the `Manager` class appears in a module of its own, but it could have been added to the `person` module instead (Python doesn't require just one class per file). It inherits the constructor and last-name methods from its superclass, but it customizes just the `giveRaise` method (there are a variety of ways to code this extension, as we'll see later). Because this change is being added as a new subclass, the original `Person` class, and any objects generated from it, will continue working unchanged. Bob and Sue, for example, inherit the original raise logic, but Tom gets the custom version because of the class from which he is created. In OOP, we program by *customizing*, not by changing.

In fact, code that uses our objects doesn't need to be at all aware of what the raise method does—it's up to the object to do the right thing based on the class from which it is created. As long as the object supports the expected interface (here, a method called `giveRaise`), it will be compatible with the calling code, regardless of its specific type, and even if its method works differently than others.

If you've already studied Python, you may know this behavior as *polymorphism*; it's a core property of the language, and it accounts for much of your code's flexibility. When the following code calls the `giveRaise` method, for example, what happens depends on the `obj` object being processed; Tom gets a 20 percent raise instead of 10 percent because of the `Manager` class's customization:

```
>>> from person import Person
>>> from manager import Manager

>>> bob = Person(name='Bob Smith', age=42, pay=10000)
>>> sue = Person(name='Sue Jones', age=45, pay=20000)
>>> tom = Manager(name='Tom Doe', age=55, pay=30000)
>>> db = [bob, sue, tom]
```

```
>>> for obj in db:  
    obj.giveRaise(.10)          # default or custom  
  
>>> for obj in db:  
    print(obj.lastName(), '=>', obj.pay)  
  
Smith => 11000.0  
Jones => 22000.0  
Doe => 36000.0
```

Refactoring Code

Before we move on, there are a few coding alternatives worth noting here. Most of these underscore the Python OOP model, and they serve as a quick review.

Augmenting methods

As a first alternative, notice that we have introduced some redundancy in [Example 1-16](#): the raise calculation is now repeated in two places (in the two classes). We could also have implemented the customized `Manager` class by *augmenting* the inherited `raise` method instead of replacing it completely:

```
class Manager(Person):  
    def giveRaise(self, percent, bonus=0.1):  
        Person.giveRaise(self, percent + bonus)
```

The trick here is to call back the superclass's version of the method directly, passing in the `self` argument explicitly. We still redefine the method, but we simply run the general version after adding 10 percent (by default) to the passed-in percentage. This coding pattern can help reduce code redundancy (the original raise method's logic appears in only one place and so is easier to change) and is especially handy for kicking off superclass constructor methods in practice.

If you've already studied Python OOP, you know that this coding scheme works because we can always call methods through either an instance or the class name. In general, the following are equivalent, and both forms may be used explicitly:

```
instance.method(arg1, arg2)  
class.method(instance, arg1, arg2)
```

In fact, the first form is mapped to the second—when calling through the instance, Python determines the class by searching the inheritance tree for the method name and passes in the instance automatically. Either way, within `giveRaise`, `self` refers to the instance that is the subject of the call.

Display format

For more object-oriented fun, we could also add a few operator overloading methods to our people classes. For example, a `__str__` method, shown here, could return a string

to give the display format for our objects when they are printed as a whole—much better than the default display we get for an instance:

```
class Person:  
    def __str__(self):  
        return '<%s => %s>' % (self.__class__.__name__, self.name)  
  
tom = Manager('Tom Jones', 50)  
print(tom)                                # prints: <Manager => Tom Jones
```

Here `__class__` gives the lowest class from which `self` was made, even though `__str__` may be inherited. The net effect is that `__str__` allows us to print instances directly instead of having to print specific attributes. We could extend this `__str__` to loop through the instance's `__dict__` attribute dictionary to display all attributes generically; for this preview we'll leave this as a suggested exercise.

We might even code an `__add__` method to make `+ expressions` automatically call the `giveRaise` method. Whether we should is another question; the fact that a `+` expression gives a person a raise might seem more magical to the next person reading our code than it should.

Constructor customization

Finally, notice that we didn't pass the `job` argument when making a manager in [Example 1-16](#); if we had, it would look like this with keyword arguments:

```
tom = Manager(name='Tom Doe', age=50, pay=50000, job='manager')
```

The reason we didn't include a `job` in the example is that it's redundant with the class of the object: if someone is a manager, their class should imply their job title. Instead of leaving this field blank, though, it may make more sense to provide an explicit constructor for managers, which fills in this field automatically:

```
class Manager(Person):  
    def __init__(self, name, age, pay):  
        Person.__init__(self, name, age, pay, 'manager')
```

Now when a manager is created, its `job` is filled in automatically. The trick here is to call to the superclass's version of the method explicitly, just as we did for the `giveRaise` method earlier in this section; the only difference here is the unusual name for the constructor method.

Alternative classes

We won't use any of this section's three extensions in later examples, but to demonstrate how they work, [Example 1-17](#) collects these ideas in an alternative implementation of our `Person` classes.

Example 1-17. PP4E\Preview\person_alternative.py

```
"""
Alternative implementation of person classes, with data, behavior,
and operator overloading (not used for objects stored persistently)
"""

class Person:
    """
    a general person: data+logic
    """

    def __init__(self, name, age, pay=0, job=None):
        self.name = name
        self.age = age
        self.pay = pay
        self.job = job

    def lastName(self):
        return self.name.split()[-1]

    def giveRaise(self, percent):
        self.pay *= (1.0 + percent)

    def __str__(self):
        return ('<%s => %s: %s, %s>' %
               (self.__class__.__name__, self.name, self.job, self.pay))

class Manager(Person):
    """
    a person with custom raise
    inherits general lastname, str
    """

    def __init__(self, name, age, pay):
        Person.__init__(self, name, age, pay, 'manager')

    def giveRaise(self, percent, bonus=0.1):
        Person.giveRaise(self, percent + bonus)

if __name__ == '__main__':
    bob = Person('Bob Smith', 44)
    sue = Person('Sue Jones', 47, 40000, 'hardware')
    tom = Manager(name='Tom Doe', age=50, pay=50000)
    print(sue, sue.pay, sue.lastName())
    for obj in (bob, sue, tom):
        obj.giveRaise(.10)                      # run this obj's giveRaise
        print(obj)                            # run common __str__ method
```

Notice the polymorphism in this module's self-test loop: all three objects share the constructor, last-name, and printing methods, but the raise method called is dependent upon the class from which an instance is created. When run, [Example 1-17](#) prints the following to standard output—the manager's job is filled in at construction, we get the new custom display format for our objects, and the new version of the manager's raise method works as before:

```
<Person => Sue Jones: hardware, 40000> 40000 Jones
<Person => Bob Smith: None, 0.0>
<Person => Sue Jones: hardware, 44000.0>
<Manager => Tom Doe: manager, 60000.0>
```

Such *refactoring* (restructuring) of code is common as class hierarchies grow and evolve. In fact, as is, we still can't give someone a raise if his pay is zero (Bob is out of luck); we probably need a way to set pay, too, but we'll leave such extensions for the next release. The good news is that Python's flexibility and readability make refactoring easy—it's simple and quick to restructure your code. If you haven't used the language yet, you'll find that Python development is largely an exercise in rapid, incremental, and interactive programming, which is well suited to the shifting needs of real-world projects.

Adding Persistence

It's time for a status update. We now have encapsulated in the form of classes customizable implementations of our records and their processing logic. Making our class-based records persistent is a minor last step. We could store them in per-record pickle files again; a shelve-based storage medium will do just as well for our goals and is often easier to code. [Example 1-18](#) shows how.

Example 1-18. PP4E\Preview\make_db_classes.py

```
import shelve
from person import Person
from manager import Manager

bob = Person('Bob Smith', 42, 30000, 'software')
sue = Person('Sue Jones', 45, 40000, 'hardware')
tom = Manager('Tom Doe', 50, 50000)

db = shelve.open('class-shelve')
db['bob'] = bob
db['sue'] = sue
db['tom'] = tom
db.close()
```

This file creates three class instances (two from the original class and one from its customization) and assigns them to keys in a newly created shelve file to store them permanently. In other words, it creates a shelve of class instances; to our code, the database looks just like a dictionary of class instances, but the top-level dictionary is mapped to a shelve file again. To check our work, [Example 1-19](#) reads the shelve and prints fields of its records.

Example 1-19. PP4E\Preview\dump_db_classes.py

```
import shelve
db = shelve.open('class-shelve')
for key in db:
    print(key, '=>\n  ', db[key].name, db[key].pay)

bob = db['bob']
print(bob.lastName())
print(db['tom'].lastName())
```

Note that we don't need to reimport the `Person` class here in order to fetch its instances from the shelve or run their methods. When instances are shelved or pickled, the underlying pickling system records both instance attributes and enough information to locate their classes automatically when they are later fetched (the class's module simply has to be on the module search path when an instance is loaded). This is on purpose; because the class and its instances in the shelve are stored separately, you can change the class to modify the way stored instances are interpreted when loaded (more on this later in the book). Here is the shelve dump script's output just after creating the shelve with the maker script:

```
bob =>
    Bob Smith 30000
sue =>
    Sue Jones 40000
tom =>
    Tom Doe 50000
Smith
Doe
```

As shown in [Example 1-20](#), database updates are as simple as before (compare this to [Example 1-13](#)), but dictionary keys become attributes of instance objects, and updates are implemented by class method calls instead of hardcoded logic. Notice how we still fetch, update, and reassign to keys to update the shelve.

Example 1-20. PP4E\Preview\update_db_classes.py

```
import shelve
db = shelve.open('class-shelve')

sue = db['sue']
sue.giveRaise(.25)
db['sue'] = sue

tom = db['tom']
tom.giveRaise(.20)
db['tom'] = tom
db.close()
```

And last but not least, here is the dump script again after running the update script; Tom and Sue have new pay values, because these objects are now persistent in the shelve. We could also open and inspect the shelve by typing code at Python's interactive command line; despite its longevity, the shelve is just a Python object containing Python objects.

```
bob =>
    Bob Smith 30000
sue =>
    Sue Jones 50000.0
tom =>
    Tom Doe 65000.0
Smith
Doe
```

Tom and Sue both get a raise this time around, because they are persistent objects in the shelve database. Although shelves can also store simpler object types such as lists and dictionaries, class instances allow us to combine both data and behavior for our stored items. In a sense, instance attributes and class methods take the place of records and processing programs in more traditional schemes.

Other Database Options

At this point, we have a full-fledged database system: our classes simultaneously implement record data and record processing, and they encapsulate the implementation of the behavior. And the Python `pickle` and `shelve` modules provide simple ways to store our database persistently between program executions. This is not a relational database (we store objects, not tables, and queries take the form of Python object processing code), but it is sufficient for many kinds of programs.

If we need more functionality, we could migrate this application to even more powerful tools. For example, should we ever need full-blown SQL query support, there are interfaces that allow Python scripts to communicate with relational databases such as MySQL, PostgreSQL, and Oracle in portable ways.

ORMs (object relational mappers) such as SQLAlchemy offer another approach which retains the Python class view, but translates it to and from relational database tables—in a sense providing the best of both worlds, with Python class syntax on top, and enterprise-level databases underneath.

Moreover, the open source ZODB system provides a more comprehensive object database for Python, with support for features missing in shelves, including concurrent updates, transaction commits and rollbacks, automatic updates on in-memory component changes, and more. We'll explore these more advanced third-party tools in [Chapter 17](#). For now, let's move on to putting a good face on our system.

“Buses Considered Harmful”

Over the years, Python has been remarkably well supported by the volunteer efforts of both countless individuals and formal organizations. Today, the nonprofit Python Software Foundation (PSF) oversees Python conferences and other noncommercial activities. The PSF was preceded by the PSA, a group that was originally formed in response to an early thread on the Python newsgroup that posed the semiserious question: “What would happen if Guido was hit by a bus?”

These days, Python creator Guido van Rossum is still the ultimate arbiter of proposed Python changes. He was officially anointed the BDFL—Benevolent Dictator for Life—of Python at the first Python conference and still makes final yes and no decisions on language changes (and apart from 3.0’s deliberate incompatibilities, has usually said no: a good thing in the programming languages domain, because Python tends to change slowly and in backward-compatible ways).

But Python's user base helps support the language, work on extensions, fix bugs, and so on. It is a true community project. In fact, Python development is now a completely open process—anyone can inspect the latest source code files or submit patches by visiting a website (see <http://www.python.org> for details).

As an open source package, Python development is really in the hands of a very large cast of developers working in concert around the world—so much so that if the BDFL ever does pass the torch, Python will almost certainly continue to enjoy the kind of support its users have come to expect. Though not without pitfalls of their own, open source projects by nature tend to reflect the needs of their user communities more than either individuals or shareholders.

Given Python's popularity, bus attacks seem less threatening now than they once did. Of course, I can't speak for Guido.

Step 4: Adding Console Interaction

So far, our database program consists of class instances stored in a shelve file, as coded in the preceding section. It's sufficient as a storage medium, but it requires us to run scripts from the command line or type code interactively in order to view or process its content. Improving on this is straightforward: simply code more general programs that interact with users, either from a console window or from a full-blown graphical interface.

A Console Shelve Interface

Let's start with something simple. The most basic kind of interface we can code would allow users to type keys and values in a console window in order to process the database (instead of writing Python program code). [Example 1-21](#), for instance, implements a simple interactive loop that allows a user to query multiple record objects in the shelve by key.

Example 1-21. PP4E\Preview\peopleinteract_query.py

```
# interactive queries
import shelve
fieldnames = ('name', 'age', 'job', 'pay')
maxfield  = max(len(f) for f in fieldnames)
db = shelve.open('class-shelve')

while True:
    key = input('\nKey? => ')           # key or empty line, exc at eof
    if not key: break
    try:
        record = db[key]                # fetch by key, show in console
```

```

except:
    print('No such key "%s"!' % key)
else:
    for field in fieldnames:
        print(field.ljust(maxfield), '=>', getattr(record, field))

```

This script uses the `getattr` built-in function to fetch an object's attribute when given its name string, and the `ljust` left-justify method of strings to align outputs (`max_field`, derived from a generator expression, is the length of the longest field name). When run, this script goes into a loop, inputting keys from the interactive user (technically, from the standard input stream, which is usually a console window) and displaying the fetched records field by field. An empty line ends the session. If our shelf of class instances is still in the state we left it near the end of the last section:

```

... \PP4E\Preview> dump_db_classes.py
bob =>
    Bob Smith 30000
sue =>
    Sue Jones 50000.0
tom =>
    Tom Doe 65000.0
Smith
Doe

```

We can then use our new script to query the object database interactively, by key:

```

... \PP4E\Preview> peopleinteract_query.py

Key? => sue
name => Sue Jones
age  => 45
job   => hardware
pay   => 50000.0

Key? => nobody
No such key "nobody"!

Key? =>

```

[Example 1-22](#) goes further and allows interactive updates. For an input key, it inputs values for each field and either updates an existing record or creates a new object and stores it under the key.

Example 1-22. PP4E\Preview\peopleinteract_update.py

```

# interactive updates
import shelve
from person import Person
fieldnames = ('name', 'age', 'job', 'pay')

db = shelve.open('class-shelve')
while True:
    key = input('\nKey? => ')
    if not key: break

```

```

if key in db:
    record = db[key]                      # update existing record
else:
    record = Person(name='?', age='?')      # eval: quote strings
for field in fieldnames:
    currval = getattr(record, field)
    newtext = input('\t[%s]=%s\n\t\tnew?=>' % (field, currval))
    if newtext:
        setattr(record, field, eval(newtext))
db[key] = record
db.close()

```

Notice the use of `eval` in this script to convert inputs (as usual, that allows any Python object type, but it means you must quote string inputs explicitly) and the use of `setattr` call to assign an attribute given its name string. When run, this script allows any number of records to be added and changed; to keep the current value of a record's field, press the Enter key when prompted for a new value:

```

Key? => tom
[name]=Tom Doe
new?=>
[age]=50
new?=>56
[job]=None
new?=>'mgr'
[pay]=65000.0
new?=>90000

Key? => nobody
[name]=?
new?=>'John Doh'
[age]=?
new?=>55
[job]=None
new?=>
[pay]=0
new?=>None

Key? =>

```

This script is still fairly simplistic (e.g., errors aren't handled), but using it is much easier than manually opening and modifying the shelve at the Python interactive prompt, especially for nonprogrammers. Run the query script to check your work after an update (we could combine query and update into a single script if this becomes too cumbersome, albeit at some cost in code and user-experience complexity):

```

Key? => tom
name => Tom Doe
age  => 56
job   => mgr
pay   => 90000

Key? => nobody
name => John Doh

```

```
age  => 55
job  => None
pay  => None
```

```
Key? =>
```

Step 5: Adding a GUI

The console-based interface approach of the preceding section works, and it may be sufficient for some users assuming that they are comfortable with typing commands in a console window. With just a little extra work, though, we can add a GUI that is more modern, easier to use, less error prone, and arguably sexier.

GUI Basics

As we'll see later in this book, a variety of GUI toolkits and builders are available for Python programmers: tkinter, wxPython, PyQt, PythonCard, Dabo, and more. Of these, tkinter ships with Python, and it is something of a de facto standard.

tkinter is a lightweight toolkit and so meshes well with a scripting language such as Python; it's easy to do basic things with tkinter, and it's straightforward to do more advanced things with extensions and OOP-based code. As an added bonus, tkinter GUIs are portable across Windows, Linux/Unix, and Macintosh; simply copy the source code to the machine on which you wish to use your GUI. tkinter doesn't come with all the bells and whistles of larger toolkits such as wxPython or PyQt, but that's a major factor behind its relative simplicity, and it makes it ideal for getting started in the GUI domain.

Because tkinter is designed for scripting, coding GUIs with it is straightforward. We'll study all of its concepts and tools later in this book. But as a first example, the first program in tkinter is just a few lines of code, as shown in [Example 1-23](#).

Example 1-23. PP4E\Preview\tkinter001.py

```
from tkinter import *
Label(text='Spam').pack()
mainloop()
```

From the tkinter module (really, a module package in Python 3), we get screen device (a.k.a. "widget") construction calls such as `Label`; geometry manager methods such as `pack`; widget configuration presets such as the `TOP` and `RIGHT` attachment side hints we'll use later for `pack`; and the `mainloop` call, which starts event processing.

This isn't the most useful GUI ever coded, but it demonstrates tkinter basics and it builds the fully functional window shown in [Figure 1-1](#) in just three simple lines of code. Its window is shown here, like all GUIs in this book, running on Windows 7; it works the same on other platforms (e.g., Mac OS X, Linux, and older versions of Windows), but renders in with native look and feel on each.

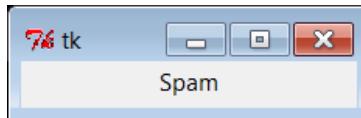


Figure 1-1. `tkinter001.py` window

You can launch this example in IDLE, from a console command line, or by clicking its icon—the same way you can run other Python scripts. tkinter itself is a standard part of Python and works out-of-the-box on Windows and others, though you may need extra configuration or install steps on some computers (more details later in this book).

It's not much more work to code a GUI that actually responds to a user: [Example 1-24](#) implements a GUI with a button that runs the `reply` function each time it is pressed.

Example 1-24. `PP4E\Preview\tkinter101.py`

```
from tkinter import *
from tkinter.messagebox import showinfo

def reply():
    showinfo(title='popup', message='Button pressed!')

window = Tk()
button = Button(window, text='press', command=reply)
button.pack()
window.mainloop()
```

This example still isn't very sophisticated—it creates an explicit `Tk` main window for the application to serve as the parent container of the button, and it builds the simple window shown in [Figure 1-2](#) (in tkinter, containers are passed in as the first argument when making a new widget; they default to the main window). But this time, each time you click the “press” button, the program responds by running Python code that pops up the dialog window in [Figure 1-3](#).

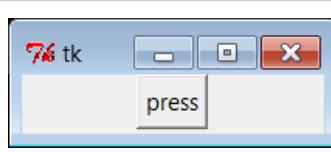


Figure 1-2. `tkinter101.py` main window

Notice that the pop-up dialog looks like it should for Windows 7, the platform on which this screenshot was taken; again, tkinter gives us a native look and feel that is appropriate for the machine on which it is running. We can customize this GUI in many ways (e.g., by changing colors and fonts, setting window titles and icons, using photos

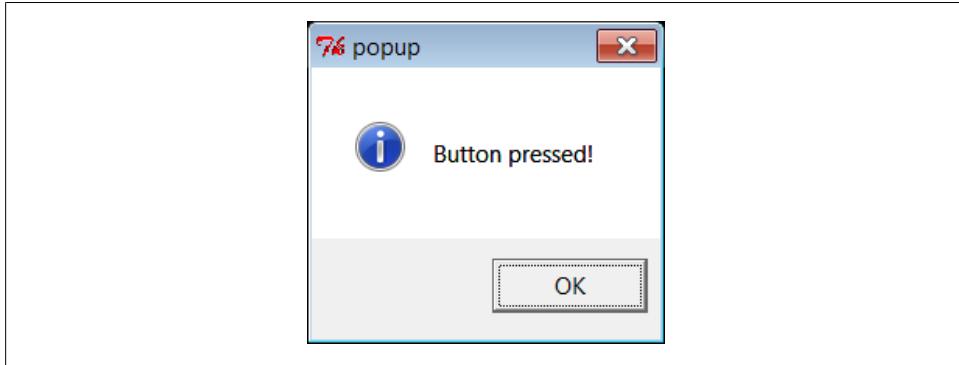


Figure 1-3. *tkinter101.py* common dialog pop up

on buttons instead of text), but part of the power of tkinter is that we need to set only the options we are interested in tailoring.

Using OOP for GUIs

All of our GUI examples so far have been top-level script code with a function for handling events. In larger programs, it is often more useful to code a GUI as a subclass of the tkinter `Frame` widget—a container for other widgets. [Example 1-25](#) shows our single-button GUI recoded in this way as a class.

Example 1-25. PP4E\Preview\tkinter102.py

```
from tkinter import *
from tkinter.messagebox import showinfo

class MyGui(Frame):
    def __init__(self, parent=None):
        Frame.__init__(self, parent)
        button = Button(self, text='press', command=self.reply)
        button.pack()
    def reply(self):
        showinfo(title='popup', message='Button pressed!')

if __name__ == '__main__':
    window = MyGui()
    window.pack()
    window.mainloop()
```

The button’s event handler is a *bound method*—`self.reply`, an object that remembers both `self` and `reply` when later called. This example generates the same window and pop up as [Example 1-24](#) (Figures 1-2 and 1-3); but because it is now a subclass of `Frame`, it automatically becomes an attachable *component*—i.e., we can add all of the widgets this class creates, as a package, to any other GUI, just by attaching this `Frame` to the GUI. [Example 1-26](#) shows how.

Example 1-26. PP4E\Preview\attachgui.py

```
from tkinter import *
from tkinter102 import MyGui

# main app window
mainwin = Tk()
Label(mainwin, text=__name__).pack()

# popup window
popup = Toplevel()
Label(popup, text='Attach').pack(side=LEFT)
MyGui(popup).pack(side=RIGHT)           # attach my frame
mainwin.mainloop()
```

This example attaches our one-button GUI to a larger window, here a `Toplevel` popup window created by the importing application and passed into the construction call as the explicit parent (you will also get a `Tk` main window; as we'll learn later, you always do, whether it is made explicit in your code or not). Our one-button widget package is attached to the right side of its container this time. If you run this live, you'll get the scene captured in [Figure 1-4](#); the “press” button is our attached custom `Frame`.

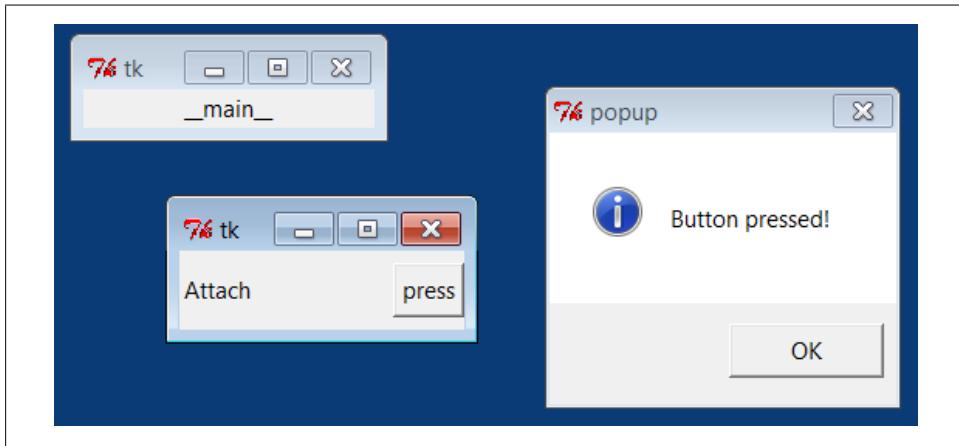


Figure 1-4. Attaching GUIs

Moreover, because `MyGui` is coded as a class, the GUI can be customized by the usual inheritance mechanism; simply define a subclass that replaces the parts that differ. The `reply` method, for example, can be customized this way to do something unique, as demonstrated in [Example 1-27](#).

Example 1-27. PP4E\Preview\customizegui.py

```
from tkinter import mainloop
from tkinter.messagebox import showinfo
from tkinter102 import MyGui
```

```

class CustomGui(MyGui):
    def reply(self):
        showinfo(title='popup', message='Ouch!')

if __name__ == '__main__':
    CustomGui().pack()
    mainloop()

```

When run, this script creates the same main window and button as the original `MyGui` class. But pressing its button generates a different reply, as shown in [Figure 1-5](#), because the custom version of the `reply` method runs.

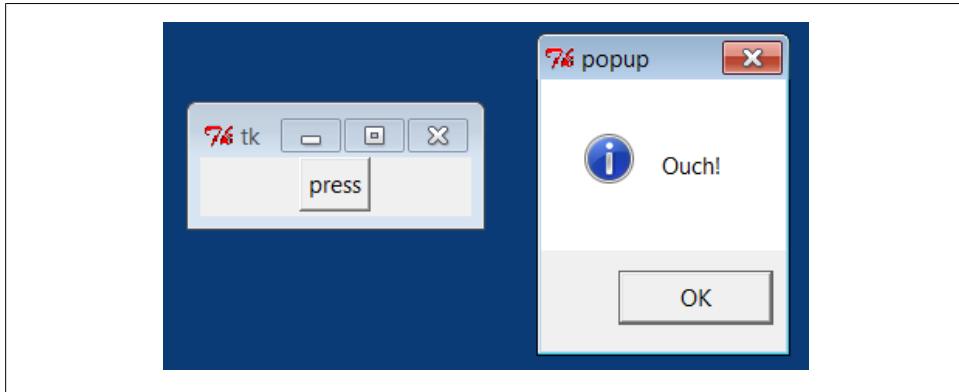


Figure 1-5. Customizing GUIs

Although these are still small GUIs, they illustrate some fairly large ideas. As we'll see later in the book, using OOP like this for inheritance and attachment allows us to reuse packages of widgets in other programs—calculators, text editors, and the like can be customized and added as components to other GUIs easily if they are classes. As we'll also find, subclasses of widget class can provide a common appearance or standardized behavior for all their instances—similar in spirit to what some observers might call GUI styles or themes. It's a normal byproduct of Python and OOP.

Getting Input from a User

As a final introductory script, [Example 1-28](#) shows how to input data from the user in an `Entry` widget and display it in a pop-up dialog. The `lambda` it uses defers the call to the `reply` function so that inputs can be passed in—a common tkinter coding pattern; without the `lambda`, `reply` would be called when the button is made, instead of when it is later pressed (we could also use `ent` as a global variable within `reply`, but that makes it less general). This example also demonstrates how to change the icon and title of a top-level window; here, the window icon file is located in the same directory as the script (if the icon call in this script fails on your platform, try commenting-out the call; icons are notoriously platform specific).

Example 1-28. PP4E\Preview\tkinter103.py

```
from tkinter import *
from tkinter.messagebox import showinfo

def reply(name):
    showinfo(title='Reply', message='Hello %s!' % name)

top = Tk()
top.title('Echo')
top.iconbitmap('py-blue-trans-out.ico')

Label(top, text="Enter your name:").pack(side=TOP)
ent = Entry(top)
ent.pack(side=TOP)
btn = Button(top, text="Submit", command=(lambda: reply(ent.get())))
btn.pack(side=LEFT)

top.mainloop()
```

As is, this example is just three widgets attached to the Tk main top-level window; later we'll learn how to use nested Frame container widgets in a window like this to achieve a variety of layouts for its three widgets. [Figure 1-6](#) gives the resulting main and pop-up windows after the Submit button is pressed. We'll see something very similar later in this chapter, but rendered in a web browser with HTML.

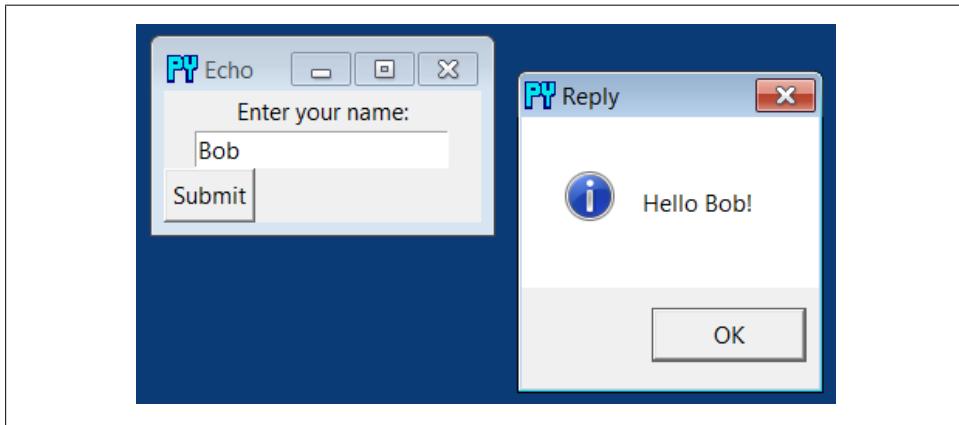


Figure 1-6. Fetching input from a user

The code we've seen so far demonstrates many of the core concepts in GUI programming, but tkinter is much more powerful than these examples imply. There are more than 20 widgets in tkinter and many more ways to input data from a user, including multiple-line text, drawing canvases, pull-down menus, radio and check buttons, and scroll bars, as well as other layout and event handling mechanisms. Beyond tkinter itself, both open source extensions such as PMW, as well as the Tix and ttk toolkits now part of Python's standard library, can add additional widgets we can use in our

Python tkinter GUIs and provide an even more professional look and feel. To hint at what is to come, let's put tkinter to work on our database of people.

A GUI Shelve Interface

For our database application, the first thing we probably want is a GUI for viewing the stored data—a form with field names and values—and a way to fetch records by key. It would also be useful to be able to update a record with new field values given its key and to add new records from scratch by filling out the form. To keep this simple, we'll use a single GUI for all of these tasks. [Figure 1-7](#) shows the window we are going to code as it looks in Windows 7; the record for the key sue has been fetched and displayed (our shelve is as we last left it again). This record is really an instance of our class in our shelve file, but the user doesn't need to care.

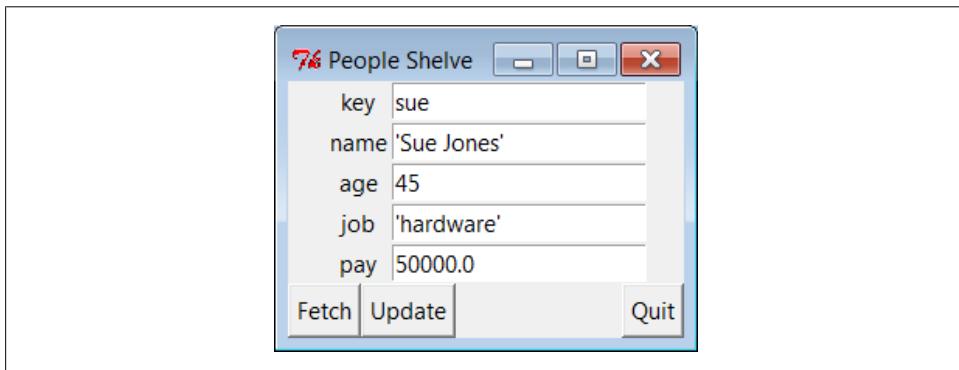


Figure 1-7. *peoplegui.py* main display/input window

Coding the GUI

Also, to keep this simple, we'll assume that all records in the database have the same sets of fields. It would be a minor extension to generalize this for any set of fields (and come up with a general form GUI constructor tool in the process), but we'll defer such evolutions to later in this book. [Example 1-29](#) implements the GUI shown in [Figure 1-7](#).

Example 1-29. PP4E\Preview\peoplegui.py

```
"""
Implement a GUI for viewing and updating class instances stored in a shelve;
the shelve lives on the machine this script runs on, as 1 or more local files;
"""

from tkinter import *
from tkinter.messagebox import showerror
import shelve
shelvename = 'class-shelve'
fieldnames = ('name', 'age', 'job', 'pay')
```

```

def makeWidgets():
    global entries
    window = Tk()
    window.title('People Shelve')
    form = Frame(window)
    form.pack()
    entries = {}
    for (ix, label) in enumerate(('key',) + fieldnames):
        lab = Label(form, text=label)
        ent = Entry(form)
        lab.grid(row=ix, column=0)
        ent.grid(row=ix, column=1)
        entries[label] = ent
    Button(window, text="Fetch", command=fetchRecord).pack(side=LEFT)
    Button(window, text="Update", command=updateRecord).pack(side=LEFT)
    Button(window, text="Quit", command=window.quit).pack(side=RIGHT)
    return window

def fetchRecord():
    key = entries['key'].get()
    try:
        record = db[key]                      # fetch by key, show in GUI
    except:
        showerror(title='Error', message='No such key!')
    else:
        for field in fieldnames:
            entries[field].delete(0, END)
            entries[field].insert(0, repr(getattr(record, field)))

def updateRecord():
    key = entries['key'].get()
    if key in db:
        record = db[key]                      # update existing record
    else:
        from person import Person              # make/store new one for key
        record = Person(name='?', age='?')     # eval: strings must be quoted
    for field in fieldnames:
        setattr(record, field, eval(entries[field].get()))
    db[key] = record

db = shelve.open(shelvename)
window = makeWidgets()
window.mainloop()
db.close() # back here after quit or window close

```

This script uses the widget `grid` method to arrange labels and entries, instead of `pack`; as we'll see later, gridding arranges by rows and columns, and so it is a natural for forms that horizontally align labels with entries well. We'll also see later that forms can usually be laid out just as nicely using `pack` with nested row frames and fixed-width labels. Although the GUI doesn't handle window resizes well yet (that requires configuration options we'll explore later), adding this makes the `grid` and `pack` alternatives roughly the same in code size.

Notice how the end of this script opens the shelfe as a global variable and starts the GUI; the shelfe remains open for the lifespan of the GUI (`mainloop` returns only after the main window is closed). As we'll see in the next section, this state retention is very different from the web model, where each interaction is normally a standalone program. Also notice that the use of global variables makes this code simple but unusable outside the context of our database; more on this later.

Using the GUI

The GUI we're building is fairly basic, but it provides a view on the shelfe file and allows us to browse and update the file without typing any code. To fetch a record from the shelfe and display it on the GUI, type its key into the GUI's "key" field and click Fetch. To change a record, type into its input fields after fetching it and click Update; the values in the GUI will be written to the record in the database. And to add a new record, fill out all of the GUI's fields with new values and click Update—the new record will be added to the shelfe file using the key and field inputs you provide.

In other words, the GUI's fields are used for both display and input. [Figure 1-8](#) shows the scene after adding a new record (via Update), and [Figure 1-9](#) shows an error dialog pop up issued when users try to fetch a key that isn't present in the shelfe.

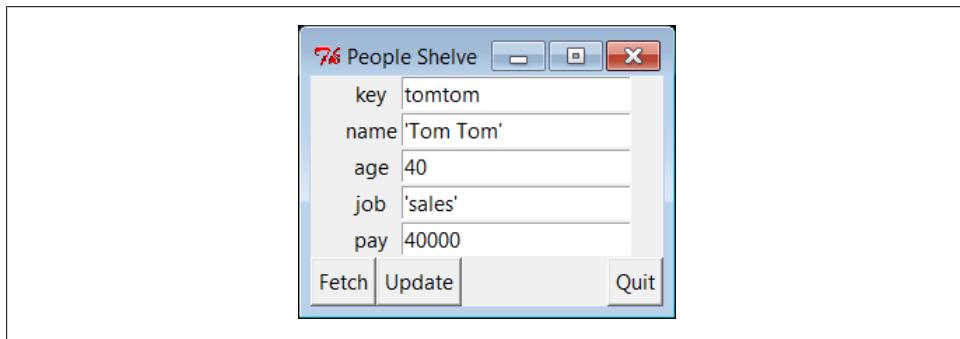


Figure 1-8. `peoplegui.py` after adding a new persistent object



Figure 1-9. `peoplegui.py` common error dialog pop up

Notice how we're using `repr` again to display field values fetched from the shelve and `eval` to convert field values to Python objects before they are stored in the shelve. As mentioned previously, this is potentially dangerous if someone sneaks some malicious code into our shelve, but we'll finesse such concerns for now.

Keep in mind, though, that this scheme means that strings must be quoted in input fields other than the key—they are assumed to be Python code. In fact, you could type an arbitrary Python expression in an input field to specify a value for an update. Typing "`Tom`"*3 in the name field, for instance, would set the name to `TomTomTom` after an update (for better or worse!); fetch to see the result.

Even though we now have a GUI for browsing and changing records, we can still check our work by interactively opening and inspecting the shelve file or by running scripts such as the `dump` utility in [Example 1-19](#). Remember, despite the fact that we're now viewing records in a GUI's windows, the database is a Python shelve file containing native Python class instance objects, so any Python code can access it. Here is the `dump` script at work after adding and changing a few persistent objects in the GUI:

```
... \PP4E\Preview> python dump_db_classes.py
sue =>
    Sue Jones 50000.0
bill =>
    bill 9999
nobody =>
    John Doh None
tomtom =>
    Tom Tom 40000
tom =>
    Tom Doe 90000
bob =>
    Bob Smith 30000
peg =>
    1 4
Smith
Doe
```

Future directions

Although this GUI does the job, there is plenty of room for improvement:

- As coded, this GUI is a simple set of functions that share the global list of input fields (`entries`) and a global shelve (`db`). We might instead pass `db` in to `makeWidgets`, and pass along both these two objects as function arguments to the callback handlers using the `Lambda` trick of the prior section. Though not crucial in a script this small, as a rule of thumb, making your external dependencies explicit like this makes your code both easier to understand and reusable in other contexts.
- We could also structure this GUI as a class to support attachment and customization (`globals` would become instance attributes), though it's unlikely that we'll need to reuse such a specific GUI.

- More usefully, we could pass in the `fieldnames` tuple as an input parameter to the functions here to allow them to be used for other record types in the future. Code at the bottom of the file would similarly become a function with a passed-in shelf filename, and we would also need to pass in a new record construction call to the update function because `Person` could not be hardcoded. Such generalization is beyond the scope of this preview, but it makes for a nice exercise if you are so inclined. Later, I'll also point you to a suggested reading example in the book examples package, `PyForm`, which takes a different approach to generalized form construction.
- To make this GUI more user friendly, it might also be nice to add an index window that displays all the keys in the database in order to make browsing easier. Some sort of verification before updates might be useful as well, and Delete and Clear buttons would be simple to code. Furthermore, assuming that inputs are Python code may be more bother than it is worth; a simpler input scheme might be easier and safer. (I won't officially say these are suggested exercises too, but it sounds like they could be.)
- We could also support window resizing (as we'll learn, widgets can grow and shrink with the window) and provide an interface for calling methods available on stored instances' classes too (as is, the `pay` field can be updated, but there is no way to invoke the `giveRaise` method).
- If we plan to distribute this GUI widely, we might package it up as a standalone executable program—a *frozen binary* in Python terminology—using third-party tools such as Py2Exe, PyInstaller, and others (search the Web for pointers). Such a program can be run directly without installing Python on the receiving end, because the Python bytecode interpreter is included in the executable itself.

I'll leave all such extensions as points to ponder, and revisit some of them later in this book.

Before we move on, two notes. First, I should mention that even more graphical packages are available to Python programmers. For instance, if you need to do graphics beyond basic windows, the `tkinter` `Canvas` widget supports freeform graphics. Third-party extensions such as Blender, OpenGL, VPython, PIL, VTK, Maya, and PyGame provide even more advanced graphics, visualization, and animation tools for use with Python scripts. Moreover, the PMW, Tix, and `ttk` widget kits mentioned earlier extend `tkinter` itself. See Python's library manual for Tix and `ttk`, and try the PyPI site or a web search for third-party graphics extensions.

And in deference to fans of other GUI toolkits such as `wxPython` and `PyQt`, I should also note that there are other GUI options to choose from and that choice is sometimes very subjective. `tkinter` is shown here because it is mature, robust, fully open source, well documented, well supported, lightweight, and a standard part of Python. By most accounts, it remains the standard for building portable GUIs in Python.

Other GUI toolkits for Python have pros and cons of their own, discussed later in this book. For example, some exchange code simplicity for richer widget sets. wxPython, for example, is much more feature-rich, but it's also much more complicated to use. By and large, though, other toolkits are variations on a theme—once you've learned one GUI toolkit, others are easy to pick up. Because of that, we'll focus on learning one toolkit in its entirety in this book instead of sampling many partially.

Although they are free to employ network access at will, programs written with traditional GUIs like tkinter generally run on a single, self-contained machine. Some consider web pages to be a kind of GUI as well, but you'll have to read the next and final section of this chapter to judge that for yourself.

For a Good Time...

There's much more to the tkinter toolkit than we've touched on in this preview, of course, and we'll study it in depth in this book. As another quick example to hint at what's possible, though, the following script, *fungui.py*, uses the Python `random` module to pick from a list, makes new independent windows with `Toplevel`, and uses the tkinter `after` callback to loop by scheduling methods to run again after a number of milliseconds:

```
from tkinter import *
import random
fontsize = 30
colors = ['red', 'green', 'blue', 'yellow', 'orange', 'cyan', 'purple']

def onSpam():
    popup = Toplevel()
    color = random.choice(colors)
    Label(popup, text='Popup', bg='black', fg=color).pack(fill=BOTH)
    mainLabel.config(fg=color)

def onFlip():
    mainLabel.config(fg=random.choice(colors))
    main.after(250, onFlip)

def onGrow():
    global fontsize
    fontsize += 5
    mainLabel.config(font=('arial', fontsize, 'italic'))
    main.after(100, onGrow)

main = Tk()
mainLabel = Label(main, text='Fun Gui!', relief=RAISED)
mainLabel.config(font=('arial', fontsize, 'italic'), fg='cyan', bg='navy')
mainLabel.pack(side=TOP, expand=YES, fill=BOTH)
Button(main, text='spam', command=onSpam).pack(fill=X)
Button(main, text='flip', command=onFlip).pack(fill=X)
Button(main, text='grow', command=onGrow).pack(fill=X)
main.mainloop()
```

Run this on your own to see how it works. It creates a main window with a custom label and three buttons—one button pops up a new window with a randomly colored label, and the other two kick off potentially independent timer loops, one of which

keeps changing the color used in the main window, and another that keeps expanding the main window label's font. Be careful if you do run this, though; the colors flash, and the label font gets bigger 10 times per second, so be sure you are able to kill the main window before it gets away from you. Hey—I warned you!

Step 6: Adding a Web Interface

GUI interfaces are easier to use than command lines and are often all we need to simplify access to data. By making our database available on the Web, though, we can open it up to even wider use. Anyone with Internet access and a web browser can access the data, regardless of where they are located and which machine they are using. Anything from workstations to cell phones will suffice. Moreover, web-based interfaces require only a web browser; there is no need to install Python to access the data except on the single-server machine. Although traditional web-based approaches may sacrifice some of the utility and speed of in-process GUI toolkits, their portability gain can be compelling.

As we'll also see later in this book, there are a variety of ways to go about scripting interactive web pages of the sort we'll need in order to access our data. Basic server-side CGI scripting is more than adequate for simple tasks like ours. Because it's perhaps the simplest approach, and embodies the foundations of more advanced techniques, CGI scripting is also well-suited to getting started on the Web.

For more advanced applications, a wealth of toolkits and frameworks for Python—including Django, TurboGears, Google's App Engine, pylons, web2py, Zope, Plone, Twisted, CherryPy, Webware, mod_python, PSP, and Quixote—can simplify common tasks and provide tools that we might otherwise need to code from scratch in the CGI world. Though they pose a new set of tradeoffs, emerging technologies such as Flex, Silverlight, and pyjamas (an AJAX-based port of the Google Web Toolkit to Python, and Python-to-JavaScript compiler) offer additional paths to achieving interactive or dynamic user-interfaces in web pages on clients, and open the door to using Python in Rich Internet Applications (RIAs).

I'll say more about these tools later. For now, let's keep things simple and code a CGI script.

CGI Basics

CGI scripting in Python is easy as long as you already have a handle on things like HTML forms, URLs, and the client/server model of the Web (all topics we'll address in detail later in this book). Whether you're aware of all the underlying details or not, the basic interaction model is probably familiar.

In a nutshell, a user visits a website and receives a form, coded in HTML, to be filled out in his or her browser. After submitting the form, a script, identified within either

the form or the address used to contact the server, is run on the server and produces another HTML page as a reply. Along the way, data typically passes through three programs: from the client browser, to the web server, to the CGI script, and back again to the browser. This is a natural model for the database access interaction we're after—users can submit a database key to the server and receive the corresponding record as a reply page.

We'll go into CGI basics in depth later in this book, but as a first example, let's start out with a simple interactive example that requests and then echoes back a user's name in a web browser. The first page in this interaction is just an input form produced by the HTML file shown in [Example 1-30](#). This HTML file is stored on the web server machine, and it is transferred to the web browser running on the client machine upon request.

Example 1-30. PP4E\Preview\cgi101.html

```
<html>
<title>Interactive Page</title>
<body>
<form method=POST action="cgi-bin/cgi101.py">
    <P><B>Enter your name:</B>
    <P><input type=text name=user>
    <P><input type=submit>
</form>
</body></html>
```

Notice how this HTML form names the script that will process its input on the server in its `action` attribute. This page is requested by submitting its URL (web address). When received by the web browser on the client, the input form that this code produces is shown in [Figure 1-10](#) (in Internet Explorer here).

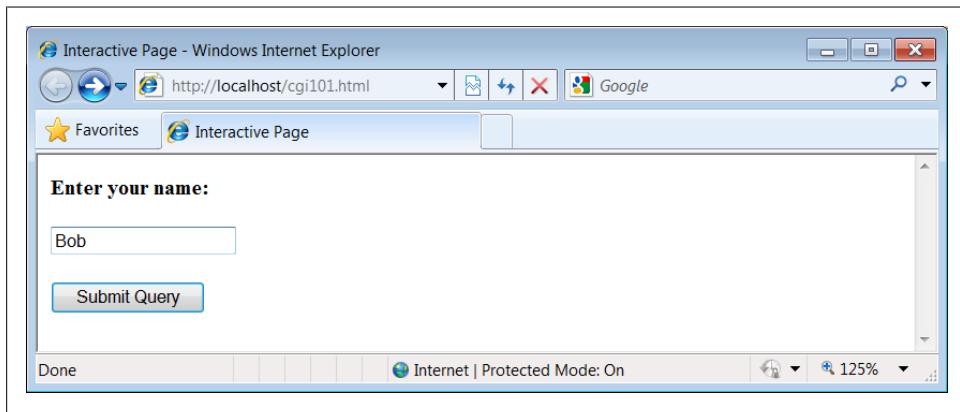


Figure 1-10. cgi101.html input form page

When this input form is submitted, a web server intercepts the request (more on the web server in a moment) and runs the Python CGI script in [Example 1-31](#). Like the

HTML file, this Python script resides on the same machine as the web server; it's run on the server machine to handle the inputs and generate a reply to the browser on the client. It uses the `cgi` module to parse the form's input and insert it into the HTML reply stream, properly escaped. The `cgi` module gives us a dictionary-like interface to form inputs sent by the browser, and the HTML code that this script prints winds up rendering the next page on the client's browser. In the CGI world, the standard output stream is connected to the client through a socket.

Example 1-31. PP4E\Preview\cgi-bin\cgi101.py

```
#!/usr/bin/python
import cgi
form = cgi.FieldStorage()           # parse form data
print('Content-type: text/html\n')    # hdr plus blank line
print('<title>Reply Page</title>')      # html reply page
if not 'user' in form:
    print('<h1>Who are you?</h1>')
else:
    print('<h1>Hello <i>%s</i>!</h1>' % cgi.escape(form['user'].value))
```

And if all goes well, we receive the reply page shown in [Figure 1-11](#)—essentially, just an echo of the data we entered in the input page. The page in this figure is produced by the HTML printed by the Python CGI script running on the server. Along the way, the user's name was transferred from a client to a server and back again—potentially across networks and miles. This isn't much of a website, of course, but the basic principles here apply, whether you're just echoing inputs or doing full-blown e-whatever.

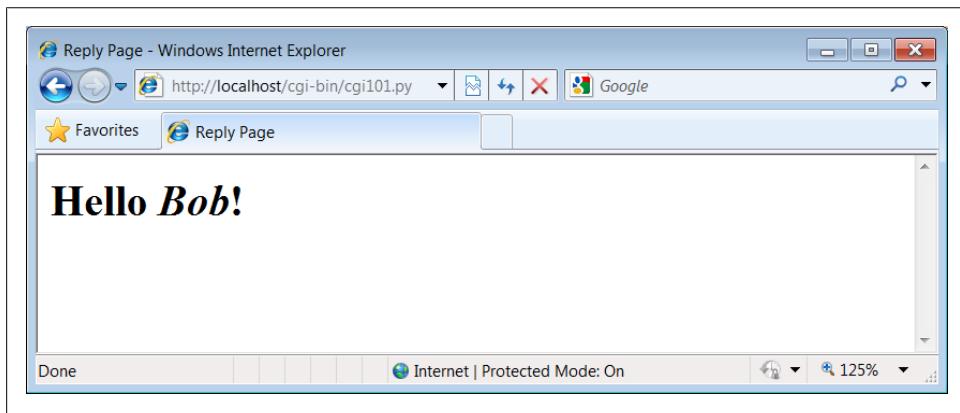


Figure 1-11. cgi101.py script reply page for input form

If you have trouble getting this interaction to run on Unix-like systems, you may need to modify the path to your Python in the `#!` line at the top of the script file and make it executable with a `chmod` command, but this is dependent on your web server (again, more on the missing server piece in a moment).

Also note that the CGI script in [Example 1-31](#) isn't printing complete HTML: the `<html>` and `<body>` tags of the static HTML file in [Example 1-30](#) are missing. Strictly speaking, such tags should be printed, but web browsers don't mind the omissions, and this book's goal is not to teach legalistic HTML; see other resources for more on HTML.

GUIs versus the Web

Before moving on, it's worth taking a moment to compare this basic CGI example with the simple GUI of [Example 1-28](#) and [Figure 1-6](#). Here, we're running scripts on a server to generate HTML that is rendered in a web browser. In the GUI, we make calls to build the display and respond to events within a single process and on a single machine. The GUI runs multiple layers of software, but not multiple programs. By contrast, the CGI approach is much more distributed—the server, the browser, and possibly the CGI script itself run as separate programs that usually communicate over a network.

Because of such differences, the standalone GUI model may be simpler and more direct: there is no intermediate server, replies do not require invoking a new program, no HTML needs to be generated, and the full power of a GUI toolkit is at our disposal. On the other hand, a web-based interface can be viewed in any browser on any computer and only requires Python on the server machine.

And just to muddle the waters further, a GUI can also employ Python's standard library networking tools to fetch and display data from a remote server (that's how web browsers do their work internally), and some newer frameworks such as Flex, Silverlight, and pyjamas provide toolkits that support more full-featured user interfaces within web pages on the client (the RIAs I mentioned earlier), albeit at some added cost in code complexity and software stack depth. We'll revisit the trade-offs of the GUI and CGI schemes later in this book, because it's a major design choice today. First, let's preview a handful of pragmatic issues related to CGI work before we apply it to our people database.

Running a Web Server

Of course, to run CGI scripts at all, we need a web server that will serve up our HTML and launch our Python scripts on request. The server is a required mediator between the browser and the CGI script. If you don't have an account on a machine that has such a server available, you'll want to run one of your own. We could configure and run a full production-level web server such as the open source Apache system (which, by the way, can be tailored with Python-specific support by the `mod_python` extension). For this chapter, however, I instead wrote a simple web server in Python using the code in [Example 1-32](#).

We'll revisit the tools used in this example later in this book. In short, because Python provides precoded support for various types of network servers, we can build a

CGI-capable and portable HTTP web server in just 8 lines of code (and a whopping 16 if we include comments and blank lines).

As we'll see later in this book, it's also easy to build proprietary network servers with low-level socket calls in Python, but the standard library provides canned implementations for many common server types, web based or otherwise. The `socketserver` module, for instance, supports threaded and forking versions of TCP and UDP servers. Third-party systems such as Twisted provide even more implementations. For serving up web content, the standard library modules used in [Example 1-32](#) provide what we need.

Example 1-32. PP4E\Preview\webserver.py

```
"""
Implement an HTTP web server in Python that knows how to run server-side
CGI scripts coded in Python;  serves files and scripts from current working
dir;  Python scripts must be stored in webdir\cgi-bin or webdir\htbin;
"""

import os, sys
from http.server import HTTPServer, CGIHTTPRequestHandler

webdir = '.'  # where your html files and cgi-bin script directory live
port   = 80   # default http://localhost/, else use http://localhost:xxxx/

os.chdir(webdir)                      # run in HTML root dir
srvraddr = ("", port)                  # my hostname, portnumber
srvrobj = HTTPServer(srvraddr, CGIHTTPRequestHandler)
srvrobj.serve_forever()                # run as perpetual daemon
```

The classes this script uses assume that the HTML files to be served up reside in the current working directory and that the CGI scripts to be run live in a *cgi-bin* or *htbin* subdirectory there. We're using a *cgi-bin* subdirectory for scripts, as suggested by the filename of [Example 1-31](#). Some web servers look at filename extensions to detect CGI scripts; our script uses this subdirectory-based scheme instead.

To launch the server, simply run this script (in a console window, by an icon click, or otherwise); it runs perpetually, waiting for requests to be submitted from browsers and other clients. The server listens for requests on the machine on which it runs and on the standard HTTP port number 80. To use this script to serve up other websites, either launch it from the directory that contains your HTML files and a *cgi-bin* subdirectory that contains your CGI scripts, or change its `webdir` variable to reflect the site's root directory (it will automatically change to that directory and serve files located there).

But where in cyberspace do you actually run the server script? If you look closely enough, you'll notice that the server name in the addresses of the prior section's examples (near the top right of the browser after the `http://`) is always *localhost*. To keep this simple, I am running the web server on the same machine as the web browser; that's what the server name "localhost" (and the equivalent IP address "127.0.0.1") means. That is, the client and server machines are the same: the client (web browser)

and server (web server) are just different processes running at the same time on the same computer.

Though not meant for enterprise-level work, this turns out to be a great way to test CGI scripts—you can develop them on the same machine without having to transfer code back to a remote server machine after each change. Simply run this script from the directory that contains both your HTML files and a *cgi-bin* subdirectory for scripts and then use *http://localhost/...* in your browser to access your HTML and script files. Here is the trace output the web server script produces in a Windows console window that is running on the same machine as the web browser and launched from the directory where the HTML files reside:

```
...\\PP4E\\Preview> python webserver.py
mark-VAIO - - [28/Jan/2010 18:34:01] "GET /cgi101.html HTTP/1.1" 200 -
mark-VAIO - - [28/Jan/2010 18:34:12] "POST /cgi-bin/cgi101.py HTTP/1.1" 200 -
mark-VAIO - - [28/Jan/2010 18:34:12] command: C:\\Python31\\python.exe -u C:\\Users
\\mark\\Stuff\\Books\\4E\\PP4E\\dev\\Examples\\PP4E\\Preview\\cgi-bin\\cgi101.py ""
mark-VAIO - - [28/Jan/2010 18:34:13] CGI script exited OK
mark-VAIO - - [28/Jan/2010 18:35:25] "GET /cgi-bin/cgi101.py?user=Sue+Smith HTTP
/1.1" 200 -
mark-VAIO - - [28/Jan/2010 18:35:25] command: C:\\Python31\\python.exe -u C:\\Users
\\mark\\Stuff\\Books\\4E\\PP4E\\dev\\Examples\\PP4E\\Preview\\cgi-bin\\cgi101.py
mark-VAIO - - [28/Jan/2010 18:35:26] CGI script exited OK
```

One pragmatic note here: you may need administrator privileges in order to run a server on the script’s default port 80 on some platforms: either find out how to run this way or try running on a different port. To run this server on a different port, change the port number in the script and name it explicitly in the URL (e.g., *http://localhost:8888/*). We’ll learn more about this convention later in this book.

And to run this server on a remote computer, upload the HTML files and CGI scripts subdirectory to the remote computer, launch the server script on that machine, and replace “localhost” in the URLs with the domain name or IP address of your server machine (e.g., *http://www.myserver.com/*). When running the server remotely, all the interaction will be as shown here, but inputs and replies will be automatically shipped across network connections, not routed between programs running on the same computer.

To delve further into the server classes our web server script employs, see their implementation in Python’s standard library (*C:\\Python31\\Lib* for Python 3.1); one of the major advantages of open source system like Python is that we can always look under the hood this way. In [Chapter 15](#), we’ll expand [Example 1-32](#) to allow the directory name and port numbers to be passed in on the command line.

Using Query Strings and `urllib`

In the basic CGI example shown earlier, we ran the Python script by filling out and submitting a form that contained the name of the script. Really, server-side CGI scripts can be invoked in a variety of ways—either by submitting an input form as shown so

far or by sending the server an explicit URL (Internet address) string that contains inputs at the end. Such an explicit URL can be sent to a server either inside or outside of a browser; in a sense, it bypasses the traditional input form page.

For instance, [Figure 1-12](#) shows the reply generated by the server after typing a URL of the following form in the address field at the top of the web browser (+ means a space here):

```
http://localhost/cgi-bin/cgi101.py?user=Sue+Smith
```

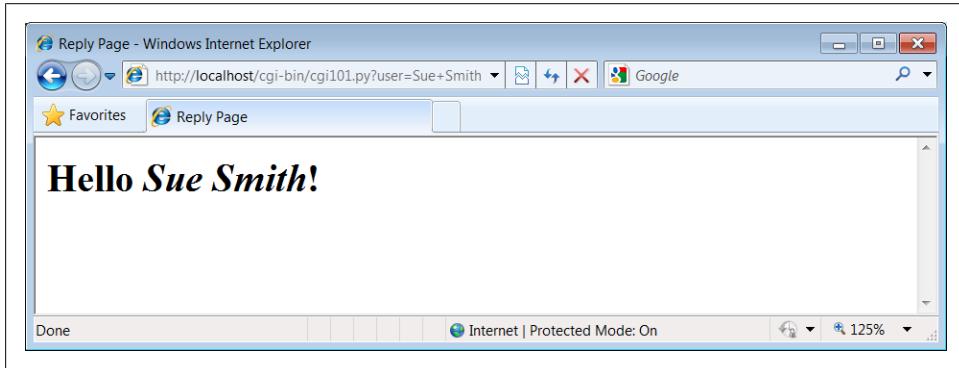


Figure 1-12. cgi101.py reply to GET-style query parameters

The inputs here, known as *query parameters*, show up at the end of the URL after the ?; they are not entered into a form's input fields. Adding inputs to URLs is sometimes called a GET request. Our original input form uses the POST method, which instead ships inputs in a separate step. Luckily, Python CGI scripts don't have to distinguish between the two; the `cgi` module's input parser handles any data submission method differences for us.

It's even possible, and often useful, to submit URLs with inputs appended as query parameters completely outside any web browser. The Python `urllib` module package, for instance, allows us to read the reply generated by a server for any valid URL. In effect, it allows us to visit a web page or invoke a CGI script from within another script; your Python code, instead of a browser, acts as the web client. Here is this module in action, run from the interactive command line:

```
>>> from urllib.request import urlopen
>>> conn = urlopen('http://localhost/cgi-bin/cgi101.py?user=Sue+Smith')
>>> reply = conn.read()
>>> reply
b'<title>Reply Page</title>\n<h1>Hello <i>Sue Smith</i>!</h1>\n'
>>> urlopen('http://localhost/cgi-bin/cgi101.py').read()
b'<title>Reply Page</title>\n<h1>Who are you?</h1>\n'
>>> urlopen('http://localhost/cgi-bin/cgi101.py?user=Bob').read()
b'<title>Reply Page</title>\n<h1>Hello <i>Bob</i>!</h1>\n'
```

The `urllib` module package gives us a file-like interface to the server's reply for a URL. Notice that the output we read from the server is raw HTML code (normally rendered by a browser). We can process this text with any of Python's text-processing tools, including:

- String methods to search and split
- The `re` regular expression pattern-matching module
- Full-blown HTML and XML parsing support in the standard library, including `html.parser`, as well as SAX-, DOM-, and ElementTree-style XML parsing tools.

When combined with such tools, the `urllib` package is a natural for a variety of techniques—ad-hoc interactive testing of websites, custom client-side GUIs, “screen scraping” of web page content, and automated regression testing systems for remote server-side CGI scripts.

Formatting Reply Text

One last fine point: because CGI scripts use text to communicate with clients, they need to format their replies according to a set of rules. For instance, notice how [Example 1-31](#) adds a blank line between the reply's header and its HTML by printing an explicit newline (`\n`) in addition to the one `print` adds automatically; this is a required separator.

Also note how the text inserted into the HTML reply is run through the `cgi.escape` (a.k.a. `html.escape` in Python 3.2; see the note under [“Python HTML and URL Escape Tools” on page 1203](#)) call, just in case the input includes a character that is special in HTML. For example, [Figure 1-13](#) shows the reply we receive for form input `Bob </i> Smith`—the `</i>` in the middle becomes `</i>` in the reply, and so doesn't interfere with real HTML code (use your browser's view source option to see this for yourself); if not escaped, the rest of the name would not be italicized.

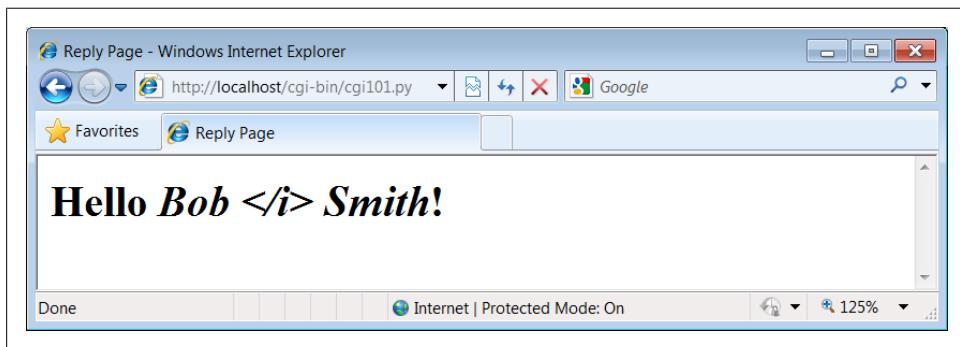


Figure 1-13. Escaping HTML characters

Escaping text like this isn't always required, but it is a good rule of thumb when its content isn't known; scripts that generate HTML have to respect its rules. As we'll see later in this book, a related call, `urllib.parse.quote`, applies URL escaping rules to text. As we'll also see, larger frameworks often handle text formatting tasks for us.

A Web-Based Shelf Interface

Now, to use the CGI techniques of the prior sections for our database application, we basically just need a bigger input and reply form. [Figure 1-14](#) shows the form we'll implement for accessing our database in a web browser.

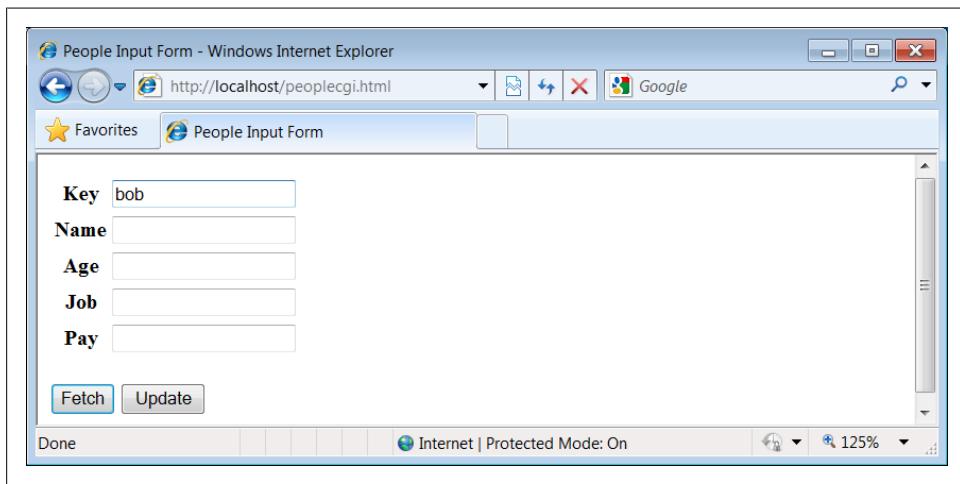


Figure 1-14. *peoplec.cgi.html* input page

Coding the website

To implement the interaction, we'll code an initial HTML input form, as well as a Python CGI script for displaying fetch results and processing update requests. [Example 1-33](#) shows the input form's HTML code that builds the page in [Figure 1-14](#).

Example 1-33. PP4E\Preview\peoplec.cgi.html

```
<html>
<title>People Input Form</title>
<body>
<form method=POST action="cgi-bin/peoplec.cgi.py">
  <table>
    <tr><th>Key <td><input type=text name=key>
    <tr><th>Name <td><input type=text name=name>
    <tr><th>Age <td><input type=text name=age>
    <tr><th>Job <td><input type=text name=job>
    <tr><th>Pay <td><input type=text name=pay>
  </table>
```

```

<p>
<input type=submit value="Fetch", name=action>
<input type=submit value="Update", name=action>
</form>
</body></html>

```

To handle form (and other) requests, [Example 1-34](#) implements a Python CGI script that fetches and updates our shelve's records. It echoes back a page similar to that produced by [Example 1-33](#), but with the form fields filled in from the attributes of actual class objects in the shelve database.

As in the GUI, the same web page is used for both displaying results and inputting updates. Unlike the GUI, this script is run anew for each step of user interaction, and it reopens the database each time (the reply page's `action` field provides a link back to the script for the next request). The basic CGI model provides no automatic memory from page to page, so we have to start from scratch each time.

Example 1-34. PP4E\Preview\cgi-bin\peoplecgi.py

```

"""
Implement a web-based interface for viewing and updating class instances
stored in a shelve; the shelve lives on server (same machine if localhost)
"""

import cgi, shelve, sys, os                      # cgi.test() dumps inputs
shelvename = 'class-shelve'                        # shelve files are in cwd
fieldnames = ('name', 'age', 'job', 'pay')

form = cgi.FieldStorage()                          # parse form data
print('Content-type: text/html')                  # hdr, blank line is in replyhtml
sys.path.insert(0, os.getcwd())                   # so this and pickler find person

# main html template
replyhtml = """
<html>
<title>People Input Form</title>
<body>
<form method=POST action="peoplecgi.py">
<table>
<tr><th>key<td><input type=text name=key value="%(key)s">
$ROWS$
</table>
<p>
<input type=submit value="Fetch", name=action>
<input type=submit value="Update", name=action>
</form>
</body></html>
"""

# insert html for data rows at $ROWS$
rowhtml  = '<tr><th>%s<td><input type=text name=%s value="%%(%s)s">\n'
rowshtml = ''
for fieldname in fieldnames:
    rowshtml += (rowhtml % ((fieldname,) * 3))


```

```

replyhtml = replyhtml.replace('$ROWS$', rowshtml)

def htmlize(adict):
    new = adict.copy()
    for field in fieldnames:                      # values may have &, >, etc.
        value = new[field]                         # display as code: quoted
        new[field] = cgi.escape(repr(value))       # html-escape special chars
    return new

def fetchRecord(db, form):
    try:
        key = form['key'].value
        record = db[key]
        fields = record.__dict__                  # use attribute dict
        fields['key'] = key                      # to fill reply string
    except:
        fields = dict.fromkeys(fieldnames, '?')
        fields['key'] = 'Missing or invalid key!'
    return fields

def updateRecord(db, form):
    if not 'key' in form:
        fields = dict.fromkeys(fieldnames, '?')
        fields['key'] = 'Missing key input!'
    else:
        key = form['key'].value
        if key in db:
            record = db[key]                    # update existing record
        else:
            from person import Person          # make/store new one for key
            record = Person(name='?', age='?')  # eval: strings must be quoted
        for field in fieldnames:
            setattr(record, field, eval(form[field].value))
        db[key] = record
        fields = record.__dict__
        fields['key'] = key
    return fields

db = shelve.open(shelvename)
action = form['action'].value if 'action' in form else None
if action == 'Fetch':
    fields = fetchRecord(db, form)
elif action == 'Update':
    fields = updateRecord(db, form)
else:
    fields = dict.fromkeys(fieldnames, '?')      # bad submit button value
    fields['key'] = 'Missing or invalid action!'
db.close()
print(replyhtml % htmlize(fields))           # fill reply from dict

```

This is a fairly large script, because it has to handle user inputs, interface with the database, and generate HTML for the reply page. Its behavior is fairly straightforward, though, and similar to the GUI of the prior section.

Directories, string formatting, and security

A few fine points before we move on. First of all, make sure the web server script we wrote earlier in [Example 1-32](#) is running before you proceed; it's going to catch our requests and route them to our script.

Also notice how this script adds the current working directory (`os.getcwd`) to the `sys.path` module search path when it first starts. Barring a `PYTHONPATH` change, this is required to allow both the pickler and this script itself to import the `person` module one level up from the script. Because of the new way the web server runs CGI scripts in Python 3, the current working directory isn't added to `sys.path`, even though the `shelve`'s files are located there correctly when opened. Such details can vary per server.

The only other feat of semi-magic the CGI script relies on is using a record's attribute dictionary (`__dict__`) as the source of values when applying HTML escapes to field values and string formatting to the HTML reply template string in the last line of the script. Recall that a `%(key)s` replacement target fetches a value by key from a dictionary:

```
>>> D = {'say': 5, 'get': 'shrubbery'}
>>> D['say']
5
>>> S = '%(say)s => %(get)s' % D
>>> S
'5 => shrubbery'
```

By using an object's attribute dictionary, we can refer to attributes by name in the format string. In fact, part of the reply template is generated by code. If its structure is confusing, simply insert statements to print `replyhtml` and to call `sys.exit`, and run from a simple command line. This is how the table's HTML in the middle of the reply is generated (slightly formatted here for readability):

```
<table>
<tr><th>key</td><input type=text name=key value="%(key)s">
<tr><th>name</td><input type=text name=name value="%(name)s">
<tr><th>age</td><input type=text name=age value="%(age)s">
<tr><th>job</td><input type=text name=job value="%(job)s">
<tr><th>pay</td><input type=text name=pay value="%(pay)s">
</table>
```

This text is then filled in with key values from the record's attribute dictionary by string formatting at the end of the script. This is done after running the dictionary through a utility to convert its values to code text with `repr` and escape that text per HTML conventions with `cgi.escape` (again, the last step isn't always required, but it's generally a good practice).

These HTML reply lines could have been hardcoded in the script, but generating them from a tuple of field names is a more general approach—we can add new fields in the future without having to update the HTML template each time. Python's string processing tools make this a snap.

In the interest of fairness, I should point out that Python's newer `str.format` method could achieve much the same effect as the traditional % format expression used by this script, and it provides specific syntax for referencing object attributes which to some might seem more explicit than using `__dict__` keys:

```
>>> D = {'say': 5, 'get': 'shrubbery'}
```

```
>>> '%(say)s => %(get)s' % D          # expression: key reference
'5 => shrubbery'
>>> '{say} => {get}'.format(**D)       # method: key reference
'5 => shrubbery'
```

```
>>> from person import Person
>>> bob = Person('Bob', 35)
```

```
>>> '%(name)s, %(age)s' % bob.__dict__    # expression: __dict__ keys
'Bob, 35'
>>> '{0.name} => {0.age}'.format(bob)     # method: attribute syntax
'Bob => 35'
```

Because we need to escape attribute values first, though, the `format` method call's attribute syntax can't be used directly this way; the choice is really between both technique's key reference syntax above. (At this writing, it's not clear which formatting technique may come to dominate, so we take liberties with using either in this book; if one replaces the other altogether someday, you'll want to go with the winner.)

In the interest of security, I also need to remind you one last time that the `eval` call used in this script to convert inputs to Python objects is powerful, but not secure—it happily runs any Python code, which can perform any system modifications that the script's process has permission to make. If you care, you'll need to trust the input source, run in a restricted environment, or use more focused input converters like `int` and `float`. This is generally a larger concern in the Web world, where request strings might arrive from arbitrary sources. Since we're all friends here, though, we'll ignore the threat.

Using the website

Despite the extra complexities of servers, directories, and strings, using the web interface is as simple as using the GUI, and it has the added advantage of running on any machine with a browser and Web connection. To fetch a record, fill in the Key field and click Fetch; the script populates the page with field data grabbed from the corresponding class instance in the shelve, as illustrated in [Figure 1-15](#) for key `bob`.

[Figure 1-15](#) shows what happens when the key comes from the posted form. As usual, you can also invoke the CGI script by instead passing inputs on a query string at the end of the URL; [Figure 1-16](#) shows the reply we get when accessing a URL of the following form:

```
http://localhost/cgi-bin/peoplecgi.py?action=Fetch&key=sue
```

key bob
name 'Bob Smith'
age 42
job 'software'
pay 30000

Fetch Update

Figure 1-15. *peoplecgi.py* reply page

key sue
name 'Sue Jones'
age 45
job 'hardware'
pay 50000.0

Fetch Update

Figure 1-16. *peoplecgi.py* reply for query parameters

As we've seen, such a URL can be submitted either within your browser or by scripts that use tools such as the `urllib` package. Again, replace "localhost" with your server's domain name if you are running the script on a remote machine.

To update a record, fetch it by key, enter new values in the field inputs, and click Update; the script will take the input fields and store them in the attributes of the class instance in the shelfe. [Figure 1-17](#) shows the reply we get after updating sue.

Finally, adding a record works the same as in the GUI: fill in a new key and field values and click Update; the CGI script creates a new class instance, fills out its attributes, and stores it in the shelf under the new key. There really is a class object behind the web page here, but we don't have to deal with the logic used to generate it. [Figure 1-18](#) shows a record added to the database in this way.

The screenshot shows a Windows Internet Explorer window with the title "People Input Form - Windows Internet Explorer". The address bar displays "http://localhost/cgi-bin/peoplecsgi.py". The main content area contains a form with the following fields:

key	sue
name	'Sue Smith'
age	45
job	'hardware'
pay	60000

Below the form are two buttons: "Fetch" and "Update". At the bottom of the window, the status bar shows "Done" and "Internet | Protected Mode: On".

Figure 1-17. *peoplecsgi.py update reply*

The screenshot shows a Windows Internet Explorer window with the title "People Input Form - Windows Internet Explorer". The address bar displays "http://localhost/cgi-bin/peoplecsgi.py". The main content area contains a form with the following fields:

key	guido
name	'GvR'
age	None
job	'BDFL'
pay	'<shrubbery>'

Below the form are two buttons: "Fetch" and "Update". At the bottom of the window, the status bar shows "Done" and "Internet | Protected Mode: On".

Figure 1-18. *peoplecsgi.py after adding a new record*

In principle, we could also update and add records by submitting a URL—either from a browser or from a script—such as:

```
http://localhost/cgi-bin/  
peoplecsgi.py?action=Update&key=sue&pay=50000&name=Sue+Smith& ...more...
```

Except for automated tools, though, typing such a long URL will be noticeably more difficult than filling out the input page. Here is part of the reply page generated for the “guido” record’s display of [Figure 1-18](#) (use your browser’s “view page source” option to see this for yourself). Note how the < and > characters are translated to HTML escapes with `cgi.escape` before being inserted into the reply:

```
<tr><th>key<td><input type=text name=key value="guido">
<tr><th>name<td><input type=text name=name value="'GVR'">
<tr><th>age<td><input type=text name=age value="None">
<tr><th>job<td><input type=text name=job value="'BDFL'">
<tr><th>pay<td><input type=text name=pay value="'&lt;shrubbery&gt;'">
```

As usual, the standard library `urllib` module package comes in handy for testing our CGI script; the output we get back is raw HTML, but we can parse it with other standard library tools and use it as the basis of a server-side script regression testing system run on any Internet-capable machine. We might even parse the server’s reply fetched this way and display its data in a client-side GUI coded with `tkinter`; GUIs and web pages are not mutually exclusive techniques. The last test in the following interaction shows a portion of the error message page’s HTML that is produced when the action is missing or invalid in the inputs, with line breaks added for readability:

```
>>> from urllib.request import urlopen
>>> url = 'http://localhost/cgi-bin/peoplecgi.py?action=Fetch&key=sue'
>>> urlopen(url).read()
b'<html>\n<title>People Input Form</title>\n<body>\n<form method=POST action="peoplecgi.py">\n      <table>\n<tr><th>key<td><input type=text name=key value="sue">\n<tr><th>name<td><input type=text name=name value="\'Sue Smith\'">\n<tr><t ...more deleted...\n\n>>> urlopen('http://localhost/cgi-bin/peoplecgi.py').read()
b'<html>\n<title>People Input Form</title>\n<body>\n<form method=POST action="peoplecgi.py">\n      <table>\n<tr><th>key<td><input type=text name=key value="Missing or invalid action!">\n      <tr><th>name<td><input type=text name=name value="\'?\'">\n<tr><th>age<td><input type=text name=age value="\'?\'">\n<tr> ...more deleted...'
```

In fact, if you’re running this CGI script on “localhost,” you can use both the last section’s GUI and this section’s web interface to view the same physical shelve file—these are just alternative interfaces to the same persistent Python objects. For comparison, [Figure 1-19](#) shows what the record we saw in [Figure 1-18](#) looks like in the GUI; it’s the same object, but we are not contacting an intermediate server, starting other scripts, or generating HTML to view it.

And as before, we can always check our work on the server machine either interactively or by running scripts. We may be viewing a database through web browsers and GUIs, but, ultimately, it is just Python objects in a Python shelve file:

```
>>> import shelve
>>> db = shelve.open('class-shelve')
>>> db['sue'].name
'Sue Smith'
```

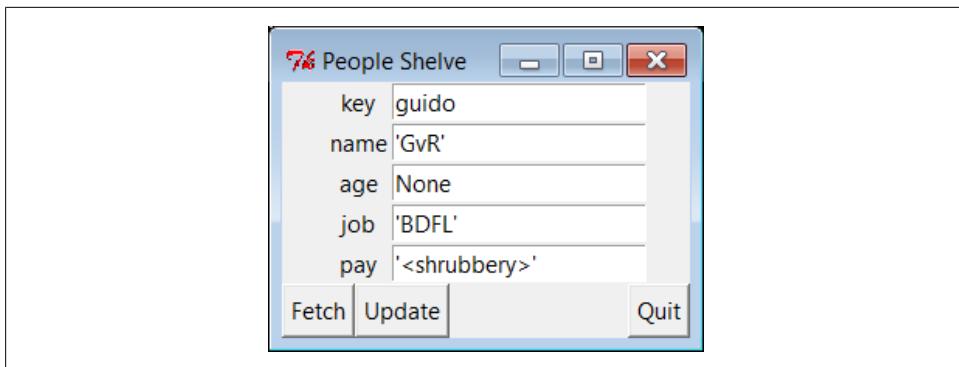


Figure 1-19. Same object displayed in the GUI

```
>>> db['guido'].job  
'BDFL'  
>>> list(db['guido'].name)  
['G', 'v', 'R']  
>>> list(db.keys())  
['sue', 'bill', 'nobody', 'tomtom', 'tom', 'bob', 'peg', 'guido']
```

Here in action again is the original database script we wrote in [Example 1-19](#) before we moved on to GUIs and the web; there are many ways to view Python data:

```
...\\PP4E\\Preview> dump_db_classes.py  
sue =>  
    Sue Smith 60000  
bill =>  
    bill 9999  
nobody =>  
    John Doh None  
tomtom =>  
    Tom Tom 40000  
tom =>  
    Tom Doe 90000  
bob =>  
    Bob Smith 30000  
peg =>  
    1 4  
guido =>  
    GvR <shrubbery>  
Smith  
Doe
```

Future directions

Naturally, there are plenty of improvements we could make here, too:

- The HTML code of the initial input page in [Example 1-33](#), for instance, is somewhat redundant with the script in [Example 1-34](#), and it could be automatically generated by another script that shares common information.

- In fact, we might avoid hardcoding HTML in our script completely if we use one of the HTML generator tools we'll meet later, including HTMLgen (a system for creating HTML from document object trees) and PSP (Python Server Pages, a server-side HTML templating system for Python similar to PHP and ASP).
- For ease of maintenance, it might also be better to split the CGI script's HTML code off to a separate file in order to better divide display from logic (different parties with possibly different skill sets could work on the different files).
- Moreover, if this website might be accessed by many people simultaneously, we would have to add file locking or move to a *database* such as ZODB or MySQL to support concurrent updates. ZODB and other full-blown database systems would also provide transaction rollbacks in the event of failures. For basic file locking, the `os.open` call and its flags provide the tools we need.
- ORMs (object relational mappers) for Python such as SQLAlchemy mentioned earlier might also allow us to gain concurrent update support of an underlying relational database system, but retain our Python class view of the data.
- In the end, if our site grows much beyond a few interactive pages, we might also migrate from basic CGI scripting to a more complete *web framework* such as one of those mentioned at the start of this section— Django, TurboGears, pyjamas, and others. If we must retain information across pages, tools such as cookies, hidden inputs, mod_python session data, and FastCGI may help too.
- If our site eventually includes content produced by its own users, we might transition to Plone, a popular open source Python- and Zope-based site builder that, using a workflow model, delegates control of site content to its producers.
- And if *wireless* or *cloud* interfaces are on our agenda, we might eventually migrate our system to cell phones using a Python port such as those available for scripting Nokia platforms and Google's Android, or to a cloud-computing platform such as Google's Python-friendly App Engine. Python tends to go wherever technology trends lead.

For now, though, both the GUI and web-based interfaces we've coded get the job done.

The End of the Demo

And that concludes our sneak preview demo of Python in action. We've explored data representation, OOP, object persistence, GUIs, and website basics. We haven't studied any of these topics in any great depth. Hopefully, though, this chapter has piqued your curiosity about Python applications programming.

In the rest of this book, we'll delve into these and other application programming tools and topics, in order to help you put Python to work in your own programs. In the next chapter, we begin our tour with the systems programming and administration tools available to Python programmers.

The Python “Secret Handshake”

I've been involved with Python for some 18 years now as of this writing in 2010, and I have seen it grow from an obscure language into one that is used in some fashion in almost every development organization and a solid member of the top four or five most widely-used programming languages in the world. It has been a fun ride.

But looking back over the years, it seems to me that if Python truly has a single legacy, it is simply that Python has made quality a more central focus in the development world. It was almost inevitable. A language that requires its users to line up code for readability can't help but make people raise questions about good software practice in general.

Probably nothing summarizes this aspect of Python life better than the standard library `this` module—a sort of Easter egg in Python written by Python core developer Tim Peters, which captures much of the design philosophy behind the language. To see `this` for yourself, go to any Python interactive prompt and import the module (naturally, it's available on all platforms):

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
>>>
```

Worth special mention, the “Explicit is better than implicit” rule has become known as “EIBTI” in the Python world—one of Python’s defining ideas, and one of its sharpest contrasts with other languages. As anyone who has worked in this field for more than a few years can attest, magic and engineering do not mix. Python has not always followed all of these guidelines, of course, but it comes very close. And if Python’s main contribution to the software world is getting people to think about such things, it seems like a win. Besides, it looked great on the T-shirt.

System Programming

This first in-depth part of the book presents Python’s system programming tools—interfaces to services in the underlying operating system as well as the context of an executing program. It consists of the following chapters:

Chapter 2

This chapter provides a comprehensive first look at commonly used system interface tools. It starts slowly and is meant in part as a reference for tools and techniques we’ll be using later in the book.

Chapter 3

This chapter continues the tour begun in [Chapter 2](#), by showing how Python’s system interfaces are applied to process standard streams, command-line arguments, shell variables, and more.

Chapter 4

This chapter continues our survey of system interfaces by focusing on tools and techniques used to process files and directories in Python. We’ll learn about binary files, tree walkers, and so on.

Chapter 5

This chapter is an introduction to Python’s library support for running programs in parallel. Here, you’ll find coverage of threads, process forks, pipes, sockets, signals, queues, and the like.

Chapter 6

This last chapter is a collection of typical system programming examples that draw upon the material of the prior four chapters. Python scripts here perform real tasks; among other things, they split and join files, compare and copy directory trees, test other programs, and search and launch files.

Although this part of the book emphasizes systems programming tasks, the tools introduced are general-purpose and are used often in later chapters.

System Tools

“The `os.path` to Knowledge”

This chapter begins our in-depth look at ways to apply Python to real programming tasks. In this and the following chapters, you’ll see how to use Python to write system tools, GUIs, database applications, Internet scripts, websites, and more. Along the way, we’ll also study larger Python programming concepts in action: code reuse, maintainability, object-oriented programming (OOP), and so on.

In this first part of the book, we begin our Python programming tour by exploring the *systems application domain*—scripts that deal with files, programs, and the general environment surrounding a program. Although the examples in this domain focus on particular kinds of tasks, the techniques they employ will prove to be useful in later parts of the book as well. In other words, you should begin your journey here, unless you are already a Python systems programming wizard.

Why Python Here?

Python’s system interfaces span application domains, but for the next five chapters, most of our examples fall into the category of *system tools*—programs sometimes called command-line utilities, shell scripts, system administration, systems programming, and other permutations of such words. Regardless of their title, you are probably already familiar with this sort of script; these scripts accomplish such tasks as processing files in a directory, launching test programs, and so on. Such programs historically have been written in nonportable and syntactically obscure shell languages such as DOS batch files, csh, and awk.

Even in this relatively simple domain, though, some of Python’s better attributes shine brightly. For instance, Python’s ease of use and extensive built-in library make it simple (and even fun) to use advanced system tools such as threads, signals, forks, sockets, and their kin; such tools are much less accessible under the obscure syntax of shell languages and the slow development cycles of compiled languages. Python’s support for concepts like code clarity and OOP also help us write shell tools that can be read,

maintained, and reused. When using Python, there is no need to start every new script from scratch.

Moreover, we'll find that Python not only includes all the interfaces we need in order to write system tools, but it also fosters script *portability*. By employing Python's standard library, most system scripts written in Python are automatically portable to all major platforms. For instance, you can usually run in Linux a Python directory-processing script written in Windows without changing its source code at all—simply copy over the source code. Though writing scripts that achieve such portability utopia requires some extra effort and practice, if used well, Python could be the only system scripting tool you need to use.

The Next Five Chapters

To make this part of the book easier to study, I have broken it down into five chapters:

- In this chapter, I'll introduce the main system-related modules in overview fashion. We'll meet some of the most commonly used system tools here for the first time.
- In [Chapter 3](#), we continue exploring the basic system interfaces by studying their role in core system programming concepts: streams, command-line arguments, environment variables, and so on.
- [Chapter 4](#) focuses on the tools Python provides for processing files, directories, and directory trees.
- In [Chapter 5](#), we'll move on to cover Python's standard tools for parallel processing—processes, threads, queues, pipes, signals, and more.
- [Chapter 6](#) wraps up by presenting a collection of complete system-oriented programs. The examples here are larger and more realistic, and they use the tools introduced in the prior four chapters to perform real, practical tasks. This collection includes both general system scripts, as well as scripts for processing directories of files.

Especially in the examples chapter at the end of this part, we will be concerned as much with system interfaces as with general Python development concepts. We'll see non-object-oriented and object-oriented versions of some examples along the way, for instance, to help illustrate the benefits of thinking in more strategic ways.

“Batteries Included”

This chapter, and those that follow, deal with both the Python language and its *standard library*—a collection of precoded modules written in Python and C that are automatically installed with the Python interpreter. Although Python itself provides an easy-to-use scripting language, much of the real action in Python development involves this vast library of programming tools (a few hundred modules at last count) that ship with the Python package.

In fact, the standard library is so powerful that it is not uncommon to hear Python described as *batteries included*—a phrase generally credited to Frank Stajano meaning that most of what you need for real day-to-day work is already there for importing. Python’s standard library, while not part of the core language per se, is a standard part of the Python system and you can expect it to be available wherever your scripts run. Indeed, this is a noteworthy difference between Python and some other scripting languages—because Python comes with so many library tools “out of the box,” supplemental sites like Perl’s CPAN are not as important.

As we’ll see, the standard library forms much of the challenge in Python programming. Once you’ve mastered the core language, you’ll find that you’ll spend most of your time applying the built-in functions and modules that come with the system. On the other hand, libraries are where most of the fun happens. In practice, programs become most interesting when they start using services external to the language interpreter: networks, files, GUIs, XML, databases, and so on. All of these are supported in the Python standard library.

Beyond the standard library, there is an additional collection of *third-party packages* for Python that must be fetched and installed separately. As of this writing, you can find most of these third-party extensions via general web searches, and using the links at <http://www.python.org> and at the PyPI website (accessible from <http://www.python.org>). Some third-party extensions are large systems in their own right; NumPy, Django, and VPython, for instance, add vector processing, website construction, and visualization, respectively.

If you have to do something special with Python, chances are good that either its support is part of the standard Python install package or you can find a free and open source module that will help. Most of the tools we’ll employ in this text are a standard part of Python, but I’ll be careful to point out things that must be installed separately. Of course, Python’s extreme code reuse idiom also makes your programs dependent on the code you reuse; in practice, though, and as we’ll see repeatedly in this book, powerful libraries coupled with open source access speed development without locking you into an existing set of features or limitations.

System Scripting Overview

To begin our exploration of the systems domain, we will take a quick tour through the standard library `sys` and `os` modules in this chapter, before moving on to larger system programming concepts. As you can tell from the length of their attribute lists, both of these are large modules—the following reflects Python 3.1 running on Windows 7 outside IDLE:

```
C:\...\PP4E\System> python
Python 3.1.1 (r311:74483, Aug 17 2009, 17:02:12) [MSC v.1500 32 bit (...)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys, os
>>> len(dir(sys))          # 65 attributes
65
```

```
>>> len(dir(os))          # 122 on Windows, more on Unix  
122  
>>> len(dir(os.path))    # a nested module within os  
52
```

The content of these two modules may vary per Python version and platform. For example, `os` is much larger under Cygwin after building Python 3.1 from its source code there (Cygwin is a system that provides Unix-like functionality on Windows; it is discussed further in “[More on Cygwin Python for Windows](#)” on page 185):

```
$ ./python.exe  
Python 3.1.1 (r311:74480, Feb 20 2010, 10:16:52)  
[GCC 3.4.4 (cygming special, gcd 0.12, using dmd 0.125)] on cygwin  
Type "help", "copyright", "credits" or "license" for more information.  
>>> import sys, os  
>>> len(dir(sys))  
64  
>>> len(dir(os))  
217  
>>> len(dir(os.path))  
51
```

As I’m not going to demonstrate every item in every built-in module, the first thing I want to do is show you how to get more details on your own. Officially, this task also serves as an excuse for introducing a few core system scripting concepts; along the way, we’ll code a first script to format documentation.

Python System Modules

Most system-level interfaces in Python are shipped in just two modules: `sys` and `os`. That’s somewhat oversimplified; other standard modules belong to this domain too. Among them are the following:

- `glob`
For filename expansion
- `socket`
For network connections and Inter-Process Communication (IPC)
- `threading, _thread, queue`
For running and synchronizing concurrent threads
- `time, timeit`
For accessing system time details
- `subprocess, multiprocessing`
For launching and controlling parallel processes
- `signal, select, shutil, tempfile, and others`
For various other system-related tasks

Third-party extensions such as `pySerial` (a serial port interface), `Pexpect` (an Expect work-alike for controlling cross-program dialogs), and even `Twisted` (a networking

framework) can be arguably lumped into the systems domain as well. In addition, some built-in functions are actually system interfaces as well—the `open` function, for example, interfaces with the file system. But by and large, `sys` and `os` together form the core of Python’s built-in system tools arsenal.

In principle at least, `sys` exports components related to the Python *interpreter* itself (e.g., the module search path), and `os` contains variables and functions that map to the operating system on which Python is run. In practice, this distinction may not always seem clear-cut (e.g., the standard input and output streams show up in `sys`, but they are arguably tied to operating system paradigms). The good news is that you’ll soon use the tools in these modules so often that their locations will be permanently stamped on your memory.*

The `os` module also attempts to provide a *portable* programming interface to the underlying operating system; its functions may be implemented differently on different platforms, but to Python scripts, they look the same everywhere. And if that’s still not enough, the `os` module also exports a nested submodule, `os.path`, which provides a portable interface to file and directory processing tools.

Module Documentation Sources

As you can probably deduce from the preceding paragraphs, learning to write system scripts in Python is mostly a matter of learning about Python’s system modules. Luckily, there are a variety of information sources to make this task easier—from module attributes to published references and books.

For instance, if you want to know everything that a built-in module exports, you can read its library manual entry; study its source code (Python is open source software, after all); or fetch its attribute list and documentation string interactively. Let’s import `sys` in Python 3.1 and see what it has to offer:

```
C:\...\PP4E\System> python
>>> import sys
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__package__',
 '__stderr__', '__stdin__', '__stdout__', '__clear_type_cache__', '__current_frames__',
 '__getframe__', 'api_version', 'argv', 'builtin_module_names', 'byteorder',
 'call_tracing', 'callstats', 'copyright', 'displayhook', 'dllhandle',
 'dont_write_bytecode', 'exc_info', 'excepthook', 'exec_prefix', 'executable',
 'exit', 'flags', 'float_info', 'float_repr_style', 'getcheckinterval',
 'getdefaultencoding', 'getfilesystemencoding', 'getprofile', 'getrecursionlimit',
 'getrefcount', 'getsizeof', 'gettrace', 'getwindowsversion', 'hexversion',
 'int_info', 'intern', 'maxsize', 'maxunicode', 'meta_path', 'modules', 'path',
 'path_hooks', 'path_importer_cache', 'platform', 'prefix', 'ps1', 'ps2',
 'setcheckinterval', 'setfilesystemencoding', 'setprofile', 'setrecursionlimit',
```

* They may also work their way into your subconscious. Python newcomers sometimes describe a phenomenon in which they “dream in Python” (insert overly simplistic Freudian analysis here...).

```
'settrace', 'stderr', 'stdin', 'stdout', 'subversion', 'version', 'version_info',
'warnoptions', 'winver']
```

The `dir` function simply returns a list containing the string names of all the attributes in any object with attributes; it's a handy memory jogger for modules at the interactive prompt. For example, we know there is something called `sys.version`, because the name `version` came back in the `dir` result. If that's not enough, we can always consult the `__doc__` string of built-in modules:

```
>>> sys.__doc__
"This module provides access to some objects used or maintained by the\ncurrent interpreter and to functions that interact strongly with the interpreter.\n\nDynamic obj
ects:\n\nargv -- command line arguments; argv[0] is the script pathname if known
\npath -- module search path; path[0] is the script directory, else ''\nmodules
-- dictionary of loaded modules\nndisplayhook -- called to show results in an i
...lots of text deleted here..."
```

Paging Documentation Strings

The `__doc__` built-in attribute just shown usually contains a string of documentation, but it may look a bit weird when displayed this way—it's one long string with embedded end-line characters that print as `\n`, not as a nice list of lines. To format these strings for a more humane display, you can simply use a `print` function-call statement:

```
>>> print(sys.__doc__)
This module provides access to some objects used or maintained by the
interpreter and to functions that interact strongly with the interpreter.
```

Dynamic objects:

```
argv -- command line arguments; argv[0] is the script pathname if known
path -- module search path; path[0] is the script directory, else ''
modules -- dictionary of loaded modules
```

...lots of lines deleted here...

The `print` built-in function, unlike interactive displays, interprets end-line characters correctly. Unfortunately, `print` doesn't, by itself, do anything about scrolling or paging and so can still be unwieldy on some platforms. Tools such as the built-in `help` function can do better:

```
>>> help(sys)
Help on built-in module sys:

NAME
    sys

FILE
    (built-in)

MODULE DOCS
    http://docs.python.org/library/sys
```

DESCRIPTION

This module provides access to some objects used or maintained by the interpreter and to functions that interact strongly with the interpreter.

Dynamic objects:

```
argv -- command line arguments; argv[0] is the script pathname if known  
path -- module search path; path[0] is the script directory, else ''  
modules -- dictionary of loaded modules
```

...lots of lines deleted here...

The `help` function is one interface provided by the PyDoc system—standard library code that ships with Python and renders documentation (documentation strings, as well as structural details) related to an object in a formatted way. The format is either like a Unix manpage, which we get for `help`, or an HTML page, which is more grandiose. It's a handy way to get basic information when working interactively, and it's a last resort before falling back on manuals and books.

A Custom Paging Script

The `help` function we just met is also fairly fixed in the way it displays information; although it attempts to page the display in some contexts, its page size isn't quite right on some of the machines I use. Moreover, it doesn't page at all in the IDLE GUI, instead relying on manual use of the scrollbar—potentially painful for large displays. When I want more control over the way help text is printed, I usually use a utility script of my own, like the one in [Example 2-1](#).

Example 2-1. PP4E\System\more.py

```
"""  
split and interactively page a string or file of text  
"""  
  
def more(text, numlines=15):  
    lines = text.splitlines()                      # like split('\n') but no '' at end  
    while lines:  
        chunk = lines[:numlines]  
        lines = lines[numlines:]  
        for line in chunk: print(line)  
        if lines and input('More?') not in ['y', 'Y']: break  
  
if __name__ == '__main__':  
    import sys                                     # when run, not imported  
    more(open(sys.argv[1]).read(), 10)             # page contents of file on cmdline
```

The meat of this file is its `more` function, and if you know enough Python to be qualified to read this book, it should be fairly straightforward. It simply splits up a string around end-line characters, and then slices off and displays a few lines at a time (15 by default) to avoid scrolling off the screen. A slice expression, `lines[:15]`, gets the first 15 items in a list, and `lines[15:]` gets the rest; to show a different number of lines each time,

pass a number to the `numlines` argument (e.g., the last line in [Example 2-1](#) passes 10 to the `numlines` argument of the `more` function).

The `splitlines` string object method call that this script employs returns a list of substrings split at line ends (e.g., `["line", "line", ...]`). An alternative `splitlines` method does similar work, but retains an empty line at the end of the result if the last line is `\n` terminated:

```
>>> line = 'aaa\nbbb\nccc\n'
>>> line.split('\n')
['aaa', 'bbb', 'ccc', '']
>>> line.splitlines()
['aaa', 'bbb', 'ccc']
```

As we'll see more formally in [Chapter 4](#), the end-of-line character is normally always `\n` (which stands for a byte usually having a binary value of 10) within a Python script, no matter what platform it is run upon. (If you don't already know why this matters, DOS `\r` characters in text are dropped by default when read.)

String Method Basics

Now, [Example 2-1](#) is a simple Python program, but it already brings up three important topics that merit quick detours here: it uses string methods, reads from a file, and is set up to be run or imported. Python string methods are not a system-related tool per se, but they see action in most Python programs. In fact, they are going to show up throughout this chapter as well as those that follow, so here is a quick review of some of the more useful tools in this set. String methods include calls for searching and replacing:

```
>>> mystr = 'xxxSPAMxxx'
>>> mystr.find('SPAM')                                # return first offset
3
>>> mystr = 'xxaaxxaa'
>>> mystr.replace('aa', 'SPAM')                      # global replacement
'xxSPAMxxSPAM'
```

The `find` call returns the offset of the first occurrence of a substring, and `replace` does global search and replacement. Like all string operations, `replace` returns a new string instead of changing its subject in-place (recall that strings are immutable). With these methods, substrings are just strings; in [Chapter 19](#), we'll also meet a module called `re` that allows regular expression *patterns* to show up in searches and replacements.

In more recent Pythons, the `in` membership operator can often be used as an alternative to `find` if all we need is a yes/no answer (it tests for a substring's presence). There are also a handful of methods for removing whitespace on the ends of strings—especially useful for lines of text read from a file:

```
>>> mystr = 'xxxSPAMxxx'
>>> 'SPAM' in mystr                                  # substring search/test
```

```

True
>>> 'Ni' in mystr                                # when not found
False
>>> mystr.find('Ni')
-1

>>> mystr = '\t Ni\n'
>>> mystr.strip()                                 # remove whitespace
'Ni'
>>> mystr.rstrip()                               # same, but just on right side
'\t Ni'

```

String methods also provide functions that are useful for things such as case conversions, and a standard library module named `string` defines some useful preset variables, among other things:

```

>>> mystr = 'SHRUBBERY'
>>> mystr.lower()                                # case converters
'shrubbery'

>>> mystr.isalpha()                             # content tests
True
>>> mystr.isdigit()
False

>>> import string                                # case presets: for 'in', etc.
>>> string.ascii_lowercase
'abcdefghijklmnopqrstuvwxyz'

>>> string.whitespace                           # whitespace characters
' \t\n\r\x0b\x0c'

```

There are also methods for splitting up strings around a substring delimiter and putting them back together with a substring in between. We'll explore these tools later in this book, but as an introduction, here they are at work:

```

>>> mystr = 'aaa,bbb,ccc'
>>> mystr.split(',')                            # split into substrings list
['aaa', 'bbb', 'ccc']

>>> mystr = 'a b\nc\nd'
>>> mystr.split()                                # default delimiter: whitespace
['a', 'b', 'c', 'd']

>>> delim = 'NI'
>>> delim.join(['aaa', 'bbb', 'ccc'])        # join substrings list
'aaaNIbbbNIccc'

>>> ' '.join(['A', 'dead', 'parrot'])       # add a space between
'A dead parrot'

```

```

>>> chars = list('Lorreta')                      # convert to characters list
>>> chars
['L', 'o', 'r', 'r', 'e', 't', 'a']
>>> chars.append('!')
>>> ''.join(chars)                            # to string: empty delimiter
'Lorreta!'

```

These calls turn out to be surprisingly powerful. For example, a line of data columns separated by tabs can be parsed into its columns with a single `split` call; the `more.py` script uses the `splitlines` variant shown earlier to split a string into a list of line strings. In fact, we can emulate the `replace` call we saw earlier in this section with a `split/join` combination:

```

>>> mystr = 'xxaaxxxaa'
>>> 'SPAM'.join(mystr.split('aa'))           # str.replace, the hard way!
'xxSPAMxxSPAM'

```

For future reference, also keep in mind that Python doesn’t automatically convert strings to numbers, or vice versa; if you want to use one as you would use the other, you must say so with manual conversions:

```

>>> int("42"), eval("42")                    # string to int conversions
(42, 42)

>>> str(42), repr(42)                      # int to string conversions
('42', '42')

>>> ("%d" % 42), '{:d}'.format(42)        # via formatting expression, method
('42', '42')

>>> "42" + str(1), int("42") + 1          # concatenation, addition
('421', 43)

```

In the last command here, the first expression triggers string concatenation (since both sides are strings), and the second invokes integer addition (because both objects are numbers). Python doesn’t assume you meant one or the other and convert automatically; as a rule of thumb, Python tries to avoid magic—and the temptation to guess—whenever possible. String tools will be covered in more detail later in this book (in fact, they get a full chapter in [Part V](#)), but be sure to also see the library manual for additional string method tools.

Other String Concepts in Python 3.X: Unicode and bytes

Technically speaking, the Python 3.X string story is a bit richer than I’ve implied here. What I’ve shown so far is the `str` object type—a sequence of characters (technically, Unicode “code points” represented as Unicode “code units”) which represents both ASCII and wider Unicode text, and handles encoding and decoding both manually on request and automatically on file transfers. Strings are coded in quotes (e.g., `'abc'`), along with various syntax for coding non-ASCII text (e.g., `'\xc4\xe8'`, `'\u00c4\u00e8'`).

Really, though, 3.X has two additional string types that support most `str` string operations: `bytes`—a sequence of short integers for representing 8-bit binary data, and `bytearray`—a mutable variant of bytes. You generally know you are dealing with `bytes` if strings display or are coded with a leading “b” character before the opening quote (e.g., `b'abc'`, `b'\xc4\xe8'`). As we’ll see in [Chapter 4](#), files in 3.X follow a similar dichotomy, using `str` in text mode (which also handles Unicode encodings and line-end conversions) and `bytes` in binary mode (which transfers bytes to and from files unchanged). And in [Chapter 5](#), we’ll see the same distinction for tools like sockets, which deal in byte strings today.

Unicode text is used in Internationalized applications, and many of Python’s binary-oriented tools deal in byte strings today. This includes some file tools we’ll meet along the way, such as the `open` call, and the `os.listdir` and `os.walk` tools we’ll study in upcoming chapters. As we’ll see, even simple directory tools sometimes have to be aware of Unicode in file content and names. Moreover, tools such as object pickling and binary data parsing are byte-oriented today.

Later in the book, we’ll also find that Unicode also pops up today in the text displayed in GUIs; the bytes shipped other networks; Internet standard such as email; and even some persistence topics such as DBM files and shelves. Any interface that deals in text necessarily deals in Unicode today, because `str` is Unicode, whether ASCII or wider. Once we reach the realm of the applications programming presented in this book, Unicode is no longer an optional topic for most Python 3.X programmers.

In this book, we’ll defer further coverage of Unicode until we can see it in the context of application topics and practical programs. For more fundamental details on how 3.X’s Unicode text and binary data support impact both string and file usage in some roles, please see [Learning Python](#), Fourth Edition; since this is officially a core language topic, it enjoys in-depth coverage and a full 45-page dedicated chapter in that book.

File Operation Basics

Besides processing strings, the `more.py` script also uses files—it opens the external file whose name is listed on the command line using the built-in `open` function, and it reads that file’s text into memory all at once with the file object `read` method. Since file objects returned by `open` are part of the core Python language itself, I assume that you have at least a passing familiarity with them at this point in the text. But just in case you’ve flipped to this chapter early on in your Pythonhood, the following calls load a file’s contents into a string, load a fixed-size set of bytes into a string, load a file’s contents into a list of line strings, and load the next line in the file into a string, respectively:

```
open('file').read()          # read entire file into string
open('file').read(N)         # read next N bytes into string
open('file').readlines()     # read entire file into line strings list
open('file').readline()      # read next line, through '\n'
```

As we'll see in a moment, these calls can also be applied to shell commands in Python to read their output. File objects also have `write` methods for sending strings to the associated file. File-related topics are covered in depth in [Chapter 4](#), but making an output file and reading it back is easy in Python:

```
>>> file = open('spam.txt', 'w')      # create file spam.txt
>>> file.write('spam' * 5) + '\n'    # write text: returns #characters written
21
>>> file.close()

>>> file = open('spam.txt')          # or open('spam.txt').read()
>>> text = file.read()              # read into a string
>>> text
'spamspamspamspamspam\n'
```

Using Programs in Two Ways

Also by way of review, the last few lines in the `more.py` file in [Example 2-1](#) introduce one of the first big concepts in shell tool programming. They instrument the file to be used in either of two ways—as a *script* or as a *library*.

Recall that every Python module has a built-in `_name_` variable that Python sets to the `_main_` string only when the file is run as a program, not when it's imported as a library. Because of that, the `more` function in this file is executed automatically by the last line in the file when this script is run as a top-level program, but not when it is imported elsewhere. This simple trick turns out to be one key to writing reusable script code: by coding program logic as *functions* rather than as top-level code, you can also import and reuse it in other scripts.

The upshot is that we can run `more.py` by itself or import and call its `more` function elsewhere. When running the file as a top-level program, we list on the command line the name of a file to be read and paged: as I'll describe in more depth in the next chapter, words typed in the command that is used to start a program show up in the built-in `sys.argv` list in Python. For example, here is the script file in action, paging itself (be sure to type this command line in your `PP4E\System` directory, or it won't find the input file; more on command lines later):

```
C:\...\PP4E\System> python more.py more.py
"""
split and interactively page a string or file of text
"""

def more(text, numlines=15):
    lines = text.splitlines()                      # like split('\n') but no '' at end
    while lines:
        chunk = lines[:numlines]
        lines = lines[numlines:]
        for line in chunk: print(line)
More?y
    if lines and input('More?') not in ['y', 'Y']: break
```

```
if __name__ == '__main__':
    import sys                      # when run, not imported
    more(open(sys.argv[1]).read(), 10)  # page contents of file on cmdline
```

When the `more.py` file is imported, we pass an explicit string to its `more` function, and this is exactly the sort of utility we need for documentation text. Running this utility on the `sys` module's documentation string gives us a bit more information in human-readable form about what's available to scripts:

```
C:\...\PP4E\System> python
>>> from more import more
>>> import sys
>>> more(sys.__doc__)
This module provides access to some objects used or maintained by the
interpreter and to functions that interact strongly with the interpreter.
```

Dynamic objects:

```
argv -- command line arguments; argv[0] is the script pathname if known
path -- module search path; path[0] is the script directory, else ''
modules -- dictionary of loaded modules

displayhook -- called to show results in an interactive session
excepthook -- called to handle any uncaught exception other than SystemExit
    To customize printing in an interactive session or to install a custom
    top-level exception handler, assign other functions to replace these.
```

```
stdin -- standard input file object; used by input()
More?
```

Pressing “y” or “Y” here makes the function display the next few lines of documentation, and then prompt again, unless you've run past the end of the lines list. Try this on your own machine to see what the rest of the module's documentation string looks like. Also try experimenting by passing a different window size in the second argument—`more(sys.__doc__, 5)` shows just 5 lines at a time.

Python Library Manuals

If that still isn't enough detail, your next step is to read the Python library manual's entry for `sys` to get the full story. All of Python's standard manuals are available online, and they often install alongside Python itself. On Windows, the standard manuals are installed automatically, but here are a few simple pointers:

- On Windows, click the Start button, pick All Programs, select the Python entry there, and then choose the Python Manuals item. The manuals should magically appear on your display; as of Python 2.4, the manuals are provided as a Windows help file and so support searching and navigation.
- On Linux or Mac OS X, you may be able to click on the manuals' entries in a file explorer or start your browser from a shell command line and navigate to the library manual's HTML files on your machine.

- If you can't find the manuals on your computer, you can always read them online. Go to Python's website at <http://www.python.org> and follow the documentation links there. This website also has a simple searching utility for the manuals.

However you get started, be sure to pick the Library manual for things such as `sys`; this manual documents all of the standard library, built-in types and functions, and more. Python's standard manual set also includes a short tutorial, a language reference, extending references, and more.

Commercially Published References

At the risk of sounding like a marketing droid, I should mention that you can also purchase the Python manual set, printed and bound; see the book information page at <http://www.python.org> for details and links. Commercially published Python reference books are also available today, including *Python Essential Reference*, *Python in a Nutshell*, *Python Standard Library*, and *Python Pocket Reference*. Some of these books are more complete and come with examples, but the last one serves as a convenient memory jogger once you've taken a library tour or two.[†]

Introducing the `sys` Module

But enough about documentation sources (and scripting basics)—let's move on to system module details. As mentioned earlier, the `sys` and `os` modules form the core of much of Python's system-related tool set. To see how, we'll turn to a quick, interactive tour through some of the tools in these two modules before applying them in bigger examples. We'll start with `sys`, the smaller of the two; remember that to see a full list of all the attributes in `sys`, you need to pass it to the `dir` function (or see where we did so earlier in this chapter).

Platforms and Versions

Like most modules, `sys` includes both informational names and functions that take action. For instance, its attributes give us the name of the underlying operating system on which the platform code is running, the largest possible “natively sized” integer on this machine (though integers can be arbitrarily long in Python 3.X), and the version number of the Python interpreter running our code:

```
C:\...\PP4E\System> python
>>> import sys
```

[†] Full disclosure: I also wrote the last of the books listed as a replacement for the reference appendix that appeared in the first edition of this book; it's meant to be a supplement to the text you're reading, and its latest edition also serves as a translation resource for Python 2.X readers. As explained in the Preface, the book you're holding is meant as tutorial, not reference, so you'll probably want to find some sort of reference resource eventually (though I'm nearly narcissistic enough to require that it be mine).

```
>>> sys.platform, sys.maxsize, sys.version
('win32', 2147483647, '3.1.1 (r311:74483, Aug 17 2009, 17:02:12) ...more deleted...')

>>> if sys.platform[:3] == 'win': print('hello windows')
...
hello windows
```

If you have code that must act differently on different machines, simply test the `sys.platform` string as done here; although most of Python is cross-platform, nonportable tools are usually wrapped in `if` tests like the one here. For instance, we'll see later that some program launch and low-level console interaction tools may vary per platform—simply test `sys.platform` to pick the right tool for the machine on which your script is running.

The Module Search Path

The `sys` module also lets us inspect the module search path both interactively and within a Python program. `sys.path` is a list of directory name strings representing the true search path in a running Python interpreter. When a module is imported, Python scans this list from left to right, searching for the module's file on each directory named in the list. Because of that, this is the place to look to verify that your search path is really set as intended.[‡]

The `sys.path` list is simply initialized from your `PYTHONPATH` setting—the content of any `.pth` path files located in Python's directories on your machine plus system defaults—when the interpreter is first started up. In fact, if you inspect `sys.path` interactively, you'll notice quite a few directories that are not on your `PYTHONPATH`: `sys.path` also includes an indicator for the script's home directory (an empty string—something I'll explain in more detail after we meet `os.getcwd`) and a set of standard library directories that may vary per installation:

```
>>> sys.path
[ '', 'C:\\PP4thEd\\Examples', ...plus standard library paths deleted... ]
```

Surprisingly, `sys.path` can actually be *changed* by a program, too. A script can use list operations such as `append`, `extend`, `insert`, `pop`, and `remove`, as well as the `del` statement to configure the search path at runtime to include all the source directories to which it needs access. Python always uses the current `sys.path` setting to import, no matter what you've changed it to:

```
>>> sys.path.append(r'C:\\mydir')
>>> sys.path
[ '', 'C:\\PP4thEd\\Examples', ...more deleted..., 'C:\\\\mydir' ]
```

[‡] It's not impossible that Python sees `PYTHONPATH` differently than you do. A syntax error in your system shell configuration files may botch the setting of `PYTHONPATH`, even if it looks fine to you. On Windows, for example, if a space appears around the `=` of a DOS `set` command in your configuration file (e.g., `set NAME = VALUE`), you may actually set `NAME` to an empty string, not to `VALUE`!

Changing `sys.path` directly like this is an alternative to setting your `PYTHONPATH` shell variable, but not a very permanent one. Changes to `sys.path` are retained only until the Python process ends, and they must be remade every time you start a new Python program or session. However, some types of programs (e.g., scripts that run on a web server) may not be able to depend on `PYTHONPATH` settings; such scripts can instead configure `sys.path` on startup to include all the directories from which they will need to import modules. For a more concrete use case, see [Example 1-34](#) in the prior chapter—there we had to tweak the search path dynamically this way, because the web server violated our import path assumptions.

Windows Directory Paths

Notice the use of a raw string literal in the `sys.path` configuration code: because backslashes normally introduce escape code sequences in Python strings, Windows users should be sure to either double up on backslashes when using them in DOS directory path strings (e.g., in "C:\\\\dir", \\ is an escape sequence that really means \\), or use raw string constants to retain backslashes literally (e.g., `r"C:\\dir"`).

If you inspect directory paths on Windows (as in the `sys.path` interaction listing), Python prints double \\ to mean a single \\. Technically, you can get away with a single \\ in a string if it is followed by a character Python does not recognize as the rest of an escape sequence, but doubles and raw strings are usually easier than memorizing escape code tables.

Also note that most Python library calls accept either forward (/) or backward (\) slashes as directory path separators, regardless of the underlying platform. That is, / usually works on Windows too and aids in making scripts portable to Unix. Tools in the `os` and `os.path` modules, described later in this chapter, further aid in script path portability.

The Loaded Modules Table

The `sys` module also contains hooks into the interpreter; `sys.modules`, for example, is a dictionary containing one *name:module* entry for every module imported in your Python session or program (really, in the calling Python process):

```
>>> sys.modules
{'reprlib': <module 'reprlib' from 'c:\\python31\\lib\\reprlib.py'>, ...more deleted...

>>> list(sys.modules.keys())
['reprlib', 'heapq', '__future__', 'sre_compile', '__collections', 'locale', '__sre',
'functools', 'encodings', 'site', 'operator', 'io', '__main__', ...more deleted... ]

>>> sys
<module 'sys' (built-in)>
>>> sys.modules['sys']
<module 'sys' (built-in)>
```

We might use such a hook to write programs that display or otherwise process all the modules loaded by a program (just iterate over the keys of `sys.modules`).

Also in the interpret hooks category, an object's reference count is available via `sys.getrefcount`, and the names of modules built-in to the Python executable are listed in `sys.builtin_module_names`. See Python's library manual for details; these are mostly Python internals information, but such hooks can sometimes become important to programmers writing tools for other programmers to use.

Exception Details

Other attributes in the `sys` module allow us to fetch all the information related to the most recently raised Python exception. This is handy if we want to process exceptions in a more generic fashion. For instance, the `sys.exc_info` function returns a tuple with the latest exception's type, value, and traceback object. In the all class-based exception model that Python 3 uses, the first two of these correspond to the most recently raised exception's class, and the instance of it which was raised:

```
>>> try:  
...     raise IndexError  
... except:  
...     print(sys.exc_info())  
...  
(<class 'IndexError', IndexError(), <traceback object at 0x019B8288>)
```

We might use such information to format our own error message to display in a GUI pop-up window or HTML web page (recall that by default, uncaught exceptions terminate programs with a Python error display). The first two items returned by this call have reasonable string displays when printed directly, and the third is a traceback object that can be processed with the standard `traceback` module:

```
>>> import traceback, sys  
>>> def grail(x):  
...     raise TypeError('already got one')  
...  
>>> try:  
...     grail('arthur')  
... except:  
...     exc_info = sys.exc_info()  
...     print(exc_info[0])  
...     print(exc_info[1])  
...     traceback.print_tb(exc_info[2])  
...  
<class 'TypeError'>  
already got one  
  File "<stdin>", line 2, in <module>  
  File "<stdin>", line 2, in grail
```

The `traceback` module can also format messages as strings and route them to specific file objects; see the Python library manual for more details.

Other sys Module Exports

The `sys` module exports additional commonly-used tools that we will meet in the context of larger topics and examples introduced later in this part of the book. For instance:

- Command-line arguments show up as a list of strings called `sys.argv`.
- Standard streams are available as `sys.stdin`, `sys.stdout`, and `sys.stderr`.
- Program exit can be forced with `sys.exit` calls.

Since these lead us to bigger topics, though, we will cover them in sections of their own.

Introducing the os Module

As mentioned, `os` is the larger of the two core system modules. It contains all of the usual operating-system calls you use in C programs and shell scripts. Its calls deal with directories, processes, shell variables, and the like. Technically, this module provides POSIX tools—a portable standard for operating-system calls—along with platform-independent directory processing tools as the nested module `os.path`. Operationally, `os` serves as a largely portable interface to your computer’s system calls: scripts written with `os` and `os.path` can usually be run unchanged on any platform. On some platforms, `os` includes extra tools available just for that platform (e.g., low-level process calls on Unix); by and large, though, it is as cross-platform as is technically feasible.

Tools in the os Module

Let’s take a quick look at the basic interfaces in `os`. As a preview, [Table 2-1](#) summarizes some of the most commonly used tools in the `os` module, organized by functional area.

Table 2-1. Commonly used os module tools

Tasks	Tools
Shell variables	<code>os.environ</code>
Running programs	<code>os.system</code> , <code>os.popen</code> , <code>os.execv</code> , <code>os.spawnv</code>
Spawning processes	<code>os.fork</code> , <code>os.pipe</code> , <code>os.waitpid</code> , <code>os.kill</code>
Descriptor files, locks	<code>os.open</code> , <code>os.read</code> , <code>os.write</code>
File processing	<code>os.remove</code> , <code>os.rename</code> , <code>os.mkfifo</code> , <code>os.mkdir</code> , <code>os.rmdir</code>
Administrative tools	<code>os.getcwd</code> , <code>os.chdir</code> , <code>os.chmod</code> , <code>os.getpid</code> , <code>os.listdir</code> , <code>os.access</code>
Portability tools	<code>os.sep</code> , <code>os.pathsep</code> , <code>os.curdir</code> , <code>os.path.split</code> , <code>os.path.join</code>
Pathname tools	<code>os.path.exists('path')</code> , <code>os.path.isdir('path')</code> , <code>os.path.getsize('path')</code>

If you inspect this module’s attributes interactively, you get a huge list of names that will vary per Python release, will likely vary per platform, and isn’t incredibly useful

until you've learned what each name means (I've let this line-wrap and removed most of this list to save space—run the command on your own):

```
>>> import os
>>> dir(os)
['F_OK', 'MutableMapping', 'O_APPEND', 'O_BINARY', 'O_CREAT', 'O_EXCL', 'O_NOINH
ERIT', 'O_RANDOM', 'O_RDONLY', 'O_RDWR', 'O_SEQUENTIAL', 'O_SHORT_LIVED', 'O_TEM
PORARY', 'O_TEXT', 'O_TRUNC', 'O_WRONLY', 'P_DETACH', 'P_NOWAIT', 'P_NOWAITO', '
P_OVERLAY', 'P_WAIT', 'R_OK', 'SEEK_CUR', 'SEEK_END', 'SEEK_SET', 'TMP_MAX',
...9 lines removed here...
'pardir', 'path', 'pathsep', 'pipe', 'popen', 'putenv', 'read', 'remove', 'rem
ovedirs', 'rename', 'renames', 'rmdir', 'sep', 'spawnl', 'spawnle', 'spawnv', 's
pawnve', 'startfile', 'stat', 'stat_float_times', 'stat_result', 'statvfs_result
', 'strerror', 'sys', 'system', 'times', 'umask', 'unlink', 'urandom', 'utime',
'waitpid', 'walk', 'write']
```

Besides all of these, the nested `os.path` module exports even more tools, most of which are related to processing file and directory names portably:

```
>>> dir(os.path)
['__all__', '__builtins__', '__doc__', '__file__', '__name__', '__package__',
'_get_altsep', '_get_bothseps', '_get_colon', '_get_dot', '_get_empty',
'_get_sep', '_getfullpathname', 'abspath', 'altsep', 'basename', 'commonprefix',
'curdir', 'defpath', 'devnull', 'dirname', 'exists', 'expanduser', 'expandvars',
'extsep', 'genericpath', 'getatime', 'getctime', 'getmtime', 'getsize', 'isabs',
'isdir', '.isfile', 'islink', 'ismount', 'join', 'lexists', 'normcase', 'normpath',
'os', 'pardir', 'pathsep', 'realpath', 'relpath', 'sep', 'split', 'splitdrive',
'splitext', 'splitunc', 'stat', 'supports_unicode_filenames', 'sys']
```

Administrative Tools

Just in case those massive listings aren't quite enough to go on, let's experiment interactively with some of the more commonly used `os` tools. Like `sys`, the `os` module comes with a collection of informational and administrative tools:

```
>>> os.getpid()
7980
>>> os.getcwd()
'C:\\\\PP4thEd\\\\Examples\\\\PP4E\\\\System'
>>> os.chdir(r'C:\\Users')
>>> os.getcwd()
'C:\\\\Users'
```

As shown here, the `os.getpid` function gives the calling process's process ID (a unique system-defined identifier for a running program, useful for process control and unique name creation), and `os.getcwd` returns the current working directory. The current working directory is where files opened by your script are assumed to live, unless their names include explicit directory paths. That's why earlier I told you to run the following command in the directory where `more.py` lives:

```
C:\\...\\PP4E\\System> python more.py more.py
```

The input filename argument here is given without an explicit directory path (though you could add one to page files in another directory). If you need to run in a different working directory, call the `os.chdir` function to change to a new directory; your code will run relative to the new directory for the rest of the program (or until the next `os.chdir` call). The next chapter will have more to say about the notion of a current working directory, and its relation to module imports when it explores script execution context.

Portability Constants

The `os` module also exports a set of names designed to make cross-platform programming simpler. The set includes platform-specific settings for path and directory separator characters, parent and current directory indicators, and the characters used to terminate lines on the underlying computer.

```
>>> os.pathsep, os.sep, os.pardir, os.curdir, os.linesep
(';', '\\', '..', '.', '\r\n')
```

`os.sep` is whatever character is used to separate directory components on the platform on which Python is running; it is automatically preset to `\` on Windows, `/` for POSIX machines, and `:` on some Macs. Similarly, `os.pathsep` provides the character that separates directories on directory lists, `:` for POSIX and `;` for DOS and Windows.

By using such attributes when composing and decomposing system-related strings in our scripts, we make the scripts fully portable. For instance, a call of the form `dirpath.split(os.sep)` will correctly split platform-specific directory names into components, though `dirpath` may look like `dir\dir` on Windows, `dir/dir` on Linux, and `dir:dir` on some Macs. As mentioned, on Windows you can usually use forward slashes rather than backward slashes when giving filenames to be opened, but these portability constants allow scripts to be platform neutral in directory processing code.

Notice also how `os.linesep` comes back as `\r\n` here—the symbolic escape code which reflects the carriage-return + line-feed line terminator convention on Windows, which you don’t normally notice when processing text files in Python. We’ll learn more about end-of-line translations in [Chapter 4](#).

Common `os.path` Tools

The nested module `os.path` provides a large set of directory-related tools of its own. For example, it includes portable functions for tasks such as checking a file’s type (`isdir`, `isfile`, and others); testing file existence (`exists`); and fetching the size of a file by name (`getsize`):

```
>>> os.path.isdir(r'C:\Users'), os.path.isfile(r'C:\Users')
(True, False)
>>> os.path.isdir(r'C:\config.sys'), os.path.isfile(r'C:\config.sys')
(False, True)
>>> os.path.isdir('nonesuch'), os.path.isfile('nonesuch')
```

```
(False, False)

>>> os.path.exists(r'c:\Users\Brian')
False
>>> os.path.exists(r'c:\Users\Default')
True
>>> os.path.getsize(r'C:\autoexec.bat')
24
```

The `os.path.isdir` and `os.path.isfile` calls tell us whether a filename is a directory or a simple file; both return `False` if the named file does not exist (that is, nonexistence implies negation). We also get calls for splitting and joining directory path strings, which automatically use the directory name conventions on the platform on which Python is running:

```
>>> os.path.split(r'C:\temp\data.txt')
('C:\\temp', 'data.txt')

>>> os.path.join(r'C:\temp', 'output.txt')
'C:\\temp\\output.txt'

>>> name = r'C:\temp\data.txt'                                # Windows paths
>>> os.path.dirname(name), os.path.basename(name)
('C:\\temp', 'data.txt')

>>> name = '/home/lutz/temp/data.txt'                         # Unix-style paths
>>> os.path.dirname(name), os.path.basename(name)
('/home/lutz/temp', 'data.txt')

>>> os.path.splitext(r'C:\PP4thEd\Examples\PP4E\PyDemos.pyw')
('C:\\PP4thEd\\Examples\\PP4E\\PyDemos', '.pyw')
```

`os.path.split` separates a filename from its directory path, and `os.path.join` puts them back together—all in entirely portable fashion using the path conventions of the machine on which they are called. The `dirname` and `basename` calls here return the first and second items returned by a `split` simply as a convenience, and `splitext` strips the file extension (after the last `.`). Subtle point: it's almost equivalent to use string `split` and `join` method calls with the portable `os.sep` string, but not exactly:

```
>>> os.sep
'\\'
>>> pathname = r'C:\PP4thEd\Examples\PP4E\PyDemos.pyw'

>>> os.path.split(pathname)                                     # split file from dir
('C:\\PP4thEd\\Examples\\PP4E', 'PyDemos.pyw')

>>> pathname.split(os.sep)                                    # split on every slash
['C:', 'PP4thEd', 'Examples', 'PP4E', 'PyDemos.pyw']

>>> os.sep.join(pathname.split(os.sep))
'C:\\PP4thEd\\Examples\\PP4E\\PyDemos.pyw'

>>> os.path.join(*pathname.split(os.sep))
'C:\\PP4thEd\\Examples\\PP4E\\PyDemos.pyw'
```

The last join call require individual arguments (hence the *) but doesn't insert a first slash because of the Windows drive syntax; use the preceding `str.join` method instead if the difference matters. The `normpath` call comes in handy if your paths become a jumble of Unix and Windows separators:

```
>>> mixed
'C:\\temp\\public/files/index.html'
>>> os.path.normpath(mixed)
'C:\\temp\\public\\files\\index.html'
>>> print(os.path.normpath(r'C:\\temp\\sub\\.\\file.ext'))
C:\\temp\\sub\\file.ext
```

This module also has an `abspath` call that portably returns the full directory pathname of a file; it accounts for adding the current directory as a path prefix, .. parent syntax, and more:

```
>>> os.chdir(r'C:\\Users')
>>> os.getcwd()
'C:\\Users'
>>> os.path.abspath('')
# empty string means the cwd
'C:\\Users'

>>> os.path.abspath('temp')
'C:\\Users\\temp'
>>> os.path.abspath(r'PP4E\\dev')
# partial paths relative to cwd
'C:\\Users\\PP4E\\dev'

>>> os.path.abspath('.')
# relative path syntax expanded
'C:\\Users'
>>> os.path.abspath('..')
'C:\\\\'
>>> os.path.abspath(r'..\\examples')
'C:\\examples'

>>> os.path.abspath(r'C:\\PP4thEd\\chapters')
# absolute paths unchanged
'C:\\PP4thEd\\chapters'
>>> os.path.abspath(r'C:\\temp\\spam.txt')
'C:\\temp\\spam.txt'
```

Because filenames are relative to the current working directory when they aren't fully specified paths, the `os.path.abspath` function helps if you want to show users what directory is truly being used to store a file. On Windows, for example, when GUI-based programs are launched by clicking on file explorer icons and desktop shortcuts, the execution directory of the program is the clicked file's home directory, but that is not always obvious to the person doing the clicking; printing a file's `abspath` can help.

Running Shell Commands from Scripts

The `os` module is also the place where we run shell commands from within Python scripts. This concept is intertwined with others, such as streams, which we won't cover fully until the next chapter, but since this is a key concept employed throughout this

part of the book, let's take a quick first look at the basics here. Two `os` functions allow scripts to run any command line that you can type in a console window:

`os.system`

Runs a shell command from a Python script

`os.popen`

Runs a shell command and connects to its input or output streams

In addition, the relatively new `subprocess` module provides finer-grained control over streams of spawned shell commands and can be used as an alternative to, and even for the implementation of, the two calls above (albeit with some cost in extra code complexity).

What's a shell command?

To understand the scope of the calls listed above, we first need to define a few terms. In this text, the term *shell* means the system that reads and runs command-line strings on your computer, and *shell command* means a command-line string that you would normally enter at your computer's shell prompt.

For example, on Windows, you can start an MS-DOS console window (a.k.a. “Command Prompt”) and type DOS commands there—commands such as `dir` to get a directory listing, `type` to view a file, names of programs you wish to start, and so on. DOS is the system shell, and commands such as `dir` and `type` are shell commands. On Linux and Mac OS X, you can start a new shell session by opening an `xterm` or terminal window and typing shell commands there too—`ls` to list directories, `cat` to view files, and so on. A variety of shells are available on Unix (e.g., `csh`, `ksh`), but they all read and run command lines. Here are two shell commands typed and run in an MS-DOS console box on Windows:

```
C:\...\PP4E\System> dir /B  
helloshell.py ...type a shell command line  
more.py ...its output shows up here  
more.pyc ...DOS is the shell on Windows  
spam.txt  
__init__.py  
  
C:\...\PP4E\System> type helloshell.py  
# a Python program  
print('The Meaning of Life')
```

Running shell commands

None of this is directly related to Python, of course (despite the fact that Python command-line scripts are sometimes confusingly called “shell tools”). But because the `os` module's `system` and `popen` calls let Python scripts run any sort of command that the underlying system shell understands, our scripts can make use of every command-line tool available on the computer, whether it's coded in Python or not. For example, here

is some Python code that runs the two DOS shell commands typed at the shell prompt shown previously:

```
C:\...\PP4E\System> python
>>> import os
>>> os.system('dir /B')
helloshell.py
more.py
more.pyc
spam.txt
__init__.py
0
>>> os.system('type helloshell.py')
# a Python program
print('The Meaning of Life')
0

>>> os.system('type hellshell.py')
The system cannot find the file specified.
1
```

The `os`s at the end of the first two commands here are just the return values of the system call itself (its exit status; zero generally means success). The system call can be used to run any command line that we could type at the shell's prompt (here, `C:\...\PP4E\System>`). The command's output normally shows up in the Python session's or program's standard output stream.

Communicating with shell commands

But what if we want to grab a command's output within a script? The `os.system` call simply runs a shell command line, but `os.popen` also connects to the standard input or output streams of the command; we get back a file-like object connected to the command's output by default (if we pass a `w` mode flag to `popen`, we connect to the command's input stream instead). By using this object to read the output of a command spawned with `popen`, we can intercept the text that would normally appear in the console window where a command line is typed:

```
>>> open('helloshell.py').read()
"# a Python program\nprint('The Meaning of Life')\n"

>>> text = os.popen('type helloshell.py').read()
>>> text
"# a Python program\nprint('The Meaning of Life')\n"

>>> listing = os.popen('dir /B').readlines()
>>> listing
['helloshell.py\n', 'more.py\n', 'more.pyc\n', 'spam.txt\n', '__init__.py\n']
```

Here, we first fetch a file's content the usual way (using Python files), then as the output of a shell `type` command. Reading the output of a `dir` command lets us get a listing of files in a directory that we can then process in a loop. We'll learn other ways to obtain such a list in [Chapter 4](#); there we'll also learn how file *iterators* make the `readlines` call

in the `os.popen` example above unnecessary in most programs, except to display the list interactively as we did here (see also “[subprocess, os.popen, and Iterators](#)” on page 101 for more on the subject).

So far, we’ve run basic DOS commands; because these calls can run any command line that we can type at a shell prompt, they can also be used to launch other Python scripts. Assuming your system search path is set to locate your Python (so that you can use the shorter “`python`” in the following instead of the longer “`C:\Python31\python`”):

```
>>> os.system('python helloshell.py')      # run a Python program
The Meaning of Life
0
>>> output = os.popen('python helloshell.py').read()
>>> output
'The Meaning of Life\n'
```

In all of these examples, the command-line strings sent to `system` and `popen` are hardcoded, but there’s no reason Python programs could not construct such strings at runtime using normal string operations (+, %, etc.). Given that commands can be dynamically built and run this way, `system` and `popen` turn Python scripts into flexible and portable tools for launching and orchestrating other programs. For example, a Python test “driver” script can be used to run programs coded in any language (e.g., C++, Java, Python) and analyze their output. We’ll explore such a script in [Chapter 6](#). We’ll also revisit `os.popen` in the next chapter in conjunction with stream redirection; as we’ll find, this call can also send `input` to programs.

The subprocess module alternative

As mentioned, in recent releases of Python the `subprocess` module can achieve the same effect as `os.system` and `os.popen`; it generally requires extra code but gives more control over how streams are connected and used. This becomes especially useful when streams are tied in more complex ways.

For example, to run a simple shell command like we did with `os.system` earlier, this new module’s `call` function works roughly the same (running commands like “`type`” that are built into the shell on Windows requires extra protocol, though normal executables like “`python`” do not):

```
>>> import subprocess
>>> subprocess.call('python helloshell.py')          # roughly like os.system()
The Meaning of Life
0
>>> subprocess.call('cmd /C "type helloshell.py"')    # built-in shell cmd
# a Python program
print('The Meaning of Life')
0
>>> subprocess.call('type helloshell.py', shell=True)   # alternative for built-ins
# a Python program
print('The Meaning of Life')
0
```

Notice the `shell=True` in the last command here. This is a subtle and platform-dependent requirement:

- On Windows, we need to pass a `shell=True` argument to `subprocess` tools like `call` and `Popen` (shown ahead) in order to run commands built into the shell. Windows commands like “type” require this extra protocol, but normal executables like “python” do not.
- On Unix-like platforms, when `shell` is `False` (its default), the program command line is run directly by `os.execvp`, a call we’ll meet in [Chapter 5](#). If this argument is `True`, the command-line string is run through a shell instead, and you can specify the shell to use with additional arguments.

More on some of this later; for now, it’s enough to note that you may need to pass `shell=True` to run some of the examples in this section and book in Unix-like environments, if they rely on shell features like program path lookup. Since I’m running code on Windows, this argument will often be omitted here.

Besides imitating `os.system`, we can similarly use this module to emulate the `os.popen` call used earlier, to run a shell command and obtain its standard output text in our script:

```
>>> pipe = subprocess.Popen('python helloshell.py', stdout=subprocess.PIPE)
>>> pipe.communicate()
(b'The Meaning of Life\r\n', None)
>>> pipe.returncode
0
```

Here, we connect the `stdout` stream to a pipe, and `communicate` to run the command to completion and receive its standard output and error streams’ text; the command’s exit status is available in an attribute after it completes. Alternatively, we can use other interfaces to read the command’s standard output directly and wait for it to exit (which returns the exit status):

```
>>> pipe = subprocess.Popen('python helloshell.py', stdout=subprocess.PIPE)
>>> pipe.stdout.read()
b'The Meaning of Life\r\n'
>>> pipe.wait()
0
```

In fact, there are direct mappings from `os.popen` calls to `subprocess.Popen` objects:

```
>>> from subprocess import Popen, PIPE
>>> Popen('python helloshell.py', stdout=PIPE).communicate()[0]
b'The Meaning of Life\r\n'
>>>
>>> import os
>>> os.popen('python helloshell.py').read()
'The Meaning of Life\n'
```

As you can probably tell, `subprocess` is extra work in these relatively simple cases. It starts to look better, though, when we need to control additional streams in flexible ways. In fact, because it also allows us to process a command’s error and input streams

in similar ways, in Python 3.X `subprocess` replaces the original `os.popen2`, `os.popen3`, and `os.popen4` calls which were available in Python 2.X; these are now just use cases for `subprocess` object interfaces. Because more advanced use cases for this module deal with standard streams, we'll postpone additional details about this module until we study stream redirection in the next chapter.

Shell command limitations

Before we move on, you should keep in mind two limitations of `system` and `popen`. First, although these two functions themselves are fairly portable, their use is really only as portable as the commands that they run. The preceding examples that run DOS `dir` and `type` shell commands, for instance, work only on Windows, and would have to be changed in order to run `ls` and `cat` commands on Unix-like platforms.

Second, it is important to remember that running Python files as programs this way is very different and generally much slower than importing program files and calling functions they define. When `os.system` and `os.popen` are called, they must start a brand-new, independent program running on your operating system (they generally run the command in a new process). When importing a program file as a module, the Python interpreter simply loads and runs the file's code in the same process in order to generate a module object. No other program is spawned along the way.[§]

There are good reasons to build systems as separate programs, too, and in the next chapter we'll explore things such as command-line arguments and streams that allow programs to pass information back and forth. But in many cases, imported modules are a faster and more direct way to compose systems.

If you plan to use these calls in earnest, you should also know that the `os.system` call normally blocks—that is, pauses—its caller until the spawned command line exits. On Linux and Unix-like platforms, the spawned command can generally be made to run independently and in parallel with the caller by adding an & shell background operator at the end of the command line:

```
os.system("python program.py arg arg &")
```

On Windows, spawning with a DOS `start` command will usually launch the command in parallel too:

```
os.system("start program.py arg arg")
```

[§] The Python code `exec(open(file).read())` also runs a program file's code, but within the same process that called it. It's similar to an import in that regard, but it works more as if the file's text had been *pasted* into the calling program at the place where the `exec` call appears (unless explicit global or local namespace dictionaries are passed). Unlike imports, such an `exec` unconditionally reads and executes a file's code (it may be run more than once per process), no module object is generated by the file's execution, and unless optional namespace dictionaries are passed in, assignments in the file's code may overwrite variables in the scope where the `exec` appears; see other resources or the Python library manual for more details.

In fact, this is so useful that an `os.startfile` call was added in recent Python releases. This call opens a file with whatever program is listed in the Windows registry for the file’s type—as though its icon has been clicked with the mouse cursor:

```
os.startfile("webpage.html")    # open file in your web browser
os.startfile("document.doc")    # open file in Microsoft Word
os.startfile("myscript.py")     # run file with Python
```

The `os.popen` call does not generally block its caller (by definition, the caller must be able to read or write the file object returned) but callers may still occasionally become blocked under both Windows and Linux if the pipe object is closed—e.g., when garbage is collected—before the spawned program exits or the pipe is read exhaustively (e.g., with its `read()` method). As we will see later in this part of the book, the Unix `os.fork/exec` and Windows `os.spawnv` calls can also be used to run parallel programs without blocking.

Because the `os` module’s `system` and `popen` calls, as well as the `subprocess` module, also fall under the category of program launchers, stream redirectors, and cross-process communication devices, they will show up again in the following chapters, so we’ll defer further details for the time being. If you’re looking for more details right away, be sure to see the stream redirection section in the next chapter and the directory listings section in [Chapter 4](#).

Other `os` Module Exports

That’s as much of a tour around `os` as we have space for here. Since most other `os` module tools are even more difficult to appreciate outside the context of larger application topics, we’ll postpone a deeper look at them until later chapters. But to let you sample the flavor of this module, here is a quick preview for reference. Among the `os` module’s other weapons are these:

`os.environ`

Fetches and sets shell environment variables

`os.fork`

Spawns a new child process on Unix-like systems

`os.pipe`

Communicates between programs

`os.execlp`

Starts new programs

`os.spawnv`

Starts new programs with lower-level control

`os.open`

Opens a low-level descriptor-based file

`os.mkdir`

Creates a new directory

`os.mkfifo`

Creates a new named pipe

`os.stat`

Fetches low-level file information

`os.remove`

Deletes a file by its pathname

`os.walk`

Applies a function or loop body to all parts of an entire directory tree

And so on. One caution up front: the `os` module provides a set of file `open`, `read`, and `write` calls, but all of these deal with low-level file access and are entirely distinct from Python’s built-in `stdio` file objects that we create with the built-in `open` function. You should normally use the built-in `open` function, not the `os` module, for all but very special file-processing needs (e.g., opening with exclusive access file locking).

In the next chapter we will apply `sys` and `os` tools such as those we’ve introduced here to implement common system-level tasks, but this book doesn’t have space to provide an exhaustive list of the contents of modules we will meet along the way. Again, if you have not already done so, you should become acquainted with the contents of modules such as `os` and `sys` using the resources described earlier. For now, let’s move on to explore additional system tools in the context of broader system programming concepts—the context surrounding a running script.

subprocess, os.popen, and Iterators

In [Chapter 4](#), we’ll explore file iterators, but you’ve probably already studied the basics prior to picking up this book. Because `os.popen` objects have an iterator that reads one line at a time, their `readlines` method call is usually superfluous. For example, the following steps through lines produced by another program without any explicit reads:

```
>>> import os
>>> for line in os.popen('dir /B *.py'): print(line, end='')
...
helloshell.py
more.py
__init__.py
```

Interestingly, Python 3.1 implements `os.popen` using the `subprocess.Popen` object that we studied in this chapter. You can see this for yourself in file `os.py` in the Python standard library on your machine (see `C:\Python31\Lib` on Windows); the `os.popen` result is an object that manages the `Popen` object and its piped stream:

```
>>> I = os.popen('dir /B *.py')
>>> I
<os._wrap_close object at 0x013BC750>
```

Because this pipe wrapper object defines an `__iter__` method, it supports line iteration, both automatic (e.g., the `for` loop above) and manual. Curiously, although the pipe wrapper object supports direct `__next__` method calls as though it were its own iterator

(just like simple files), it does not support the `next` built-in function, even though the latter is supposed to simply call the former:

```
>>> I = os.popen('dir /B *.py')
>>> I.__next__()
'helloshell.py\n'

>>> I = os.popen('dir /B *.py')
>>> next(I)
TypeError: _wrap_close object is not an iterator
```

The reason for this is subtle—direct `__next__` calls are intercepted by a `__getattr__` defined in the pipe wrapper object, and are properly delegated to the wrapped object; but `next` function calls invoke Python’s operator overloading machinery, which in 3.X bypasses the wrapper’s `__getattr__` for special method names like `__next__`. Since the pipe wrapper object doesn’t define a `__next__` of its own, the call is not caught and delegated, and the `next` built-in fails. As explained in full in the book *Learning Python*, the wrapper’s `__getattr__` isn’t tried because 3.X begins such searches at the class, not the instance.

This behavior may or may not have been anticipated, and you don’t need to care if you iterate over pipe lines automatically with `for` loops, comprehensions, and other tools. To code manual iterations robustly, though, be sure to call the `iter` built-in first—this invokes the `__iter__` defined in the pipe wrapper object itself, to correctly support both flavors of advancement:

```
>>> I = os.popen('dir /B *.py')
>>> I = iter(I)                      # what for loops do
>>> I.__next__()                   # now both forms work
'helloshell.py\n'
>>> next(I)
'more.py\n'
```

Script Execution Context

“I’d Like to Have an Argument, Please”

Python scripts don’t run in a vacuum (despite what you may have heard). Depending on platforms and startup procedures, Python programs may have all sorts of enclosing context—information automatically passed in to the program by the operating system when the program starts up. For instance, scripts have access to the following sorts of system-level inputs and interfaces:

Current working directory

`os.getcwd` gives access to the directory from which a script is started, and many file tools use its value implicitly.

Command-line arguments

`sys.argv` gives access to words typed on the command line that are used to start the program and that serve as script inputs.

Shell variables

`os.environ` provides an interface to names assigned in the enclosing shell (or a parent program) and passed in to the script.

Standard streams

`sys.stdin`, `stdout`, and `stderr` export the three input/output streams that are at the heart of command-line shell tools, and can be leveraged by scripts with `print` options, the `os.popen` call and `subprocess` module introduced in [Chapter 2](#), the `io.StringIO` class, and more.

Such tools can serve as inputs to scripts, configuration parameters, and so on. In this chapter, we will explore all these four context’s tools—both their Python interfaces and their typical roles.

Current Working Directory

The notion of the current working directory (CWD) turns out to be a key concept in some scripts' execution: it's always the implicit place where files processed by the script are assumed to reside unless their names have absolute directory paths. As we saw earlier, `os.getcwd` lets a script fetch the CWD name explicitly, and `os.chdir` allows a script to move to a new CWD.

Keep in mind, though, that filenames without full pathnames map to the CWD and have nothing to do with your `PYTHONPATH` setting. Technically, a script is always launched from the CWD, not the directory containing the script file. Conversely, imports always first search the directory containing the script, not the CWD (unless the script happens to also be located in the CWD). Since this distinction is subtle and tends to trip up beginners, let's explore it in a bit more detail.

CWD, Files, and Import Paths

When you run a Python script by typing a shell command line such as `python dir1\dir2\file.py`, the CWD is the directory you were in when you typed this command, not `dir1\dir2`. On the other hand, Python automatically adds the identity of the script's home directory to the front of the module search path such that `file.py` can always import other files in `dir1\dir2` no matter where it is run from. To illustrate, let's write a simple script to echo both its CWD and its module search path:

```
C:\...\PP4E\System> type whereami.py
import os, sys
print('my os.getcwd =>', os.getcwd())           # show my cwd execution dir
print('my sys.path  =>', sys.path[:6])          # show first 6 import paths
input()                                         # wait for keypress if clicked
```

Now, running this script in the directory in which it resides sets the CWD as expected and adds it to the front of the module import search path. We met the `sys.path` module search path earlier; its first entry might also be the empty string to designate CWD when you're working interactively, and most of the CWD has been truncated to “...” here for display:

```
C:\...\PP4E\System> set PYTHONPATH=C:\PP4thEd\Examples
C:\...\PP4E\System> python whereami.py
my os.getcwd => C:\...\PP4E\System
my sys.path  => ['C:\...\PP4E\System', 'C:\\PP4thEd\\Examples', ...more... ]
```

But if we run this script from other places, the CWD moves with us (it's the directory where we type commands), and Python adds a directory to the front of the module search path that allows the script to still see files in its own home directory. For instance, when running from one level up (..), the `System` name added to the front of `sys.path` will be the first directory that Python searches for imports within `whereami.py`; it points imports back to the directory containing the script that was run. Filenames without

complete paths, though, will be mapped to the CWD (`C:\PP4thEd\Examples\PP4E`), not the *System* subdirectory nested there:

```
C:\...\PP4E\System> cd ..
C:\...\PP4E> python System\whereami.py
my os.getcwd => C:\...\PP4E
my sys.path => ['C:\...\PP4E\System', 'C:\PP4thEd\Examples', ...more... ]

C:\...\PP4E> cd System\temp
C:\...\PP4E\System\temp> python ..\whereami.py
my os.getcwd => C:\...\PP4E\System\temp
my sys.path => ['C:\...\PP4E\System', 'C:\PP4thEd\Examples', ...]
```

The net effect is that filenames without directory paths in a script will be mapped to the place where the *command* was typed (`os.getcwd`), but imports still have access to the directory of the *script* being run (via the front of `sys.path`). Finally, when a file is launched by clicking its icon, the CWD is just the directory that contains the clicked file. The following output, for example, appears in a new DOS console box when *whereami.py* is double-clicked in Windows Explorer:

```
my os.getcwd => C:\...\PP4E\System
my sys.path => ['C:\...\PP4E\System', ...more... ]
```

In this case, both the CWD used for filenames and the first import search directory are the directory containing the script file. This all usually works out just as you expect, but there are two pitfalls to avoid:

- Filenames might need to include complete directory paths if scripts cannot be sure from where they will be run.
- Command-line scripts cannot always rely on the CWD to gain import visibility to files that are not in their own directories; instead, use `PYTHONPATH` settings and package import paths to access modules in other directories.

For example, scripts in this book, regardless of how they are run, can always import other files in their own home directories without package path imports (`import file here`), but must go through the PP4E package root to find files anywhere else in the examples tree (`from PP4E.dir1.dir2 import filethere`), even if they are run from the directory containing the desired external module. As usual for modules, the `PP4E\dir1\dir2` directory name could also be added to `PYTHONPATH` to make files there visible everywhere without package path imports (though adding more directories to `PYTHONPATH` increases the likelihood of name clashes). In either case, though, imports are always resolved to the script's home directory or other Python search path settings, not to the CWD.

CWD and Command Lines

This distinction between the CWD and import search paths explains why many scripts in this book designed to operate in the current working directory (instead of one whose name is passed in) are run with command lines such as this one:

```
C:\temp> python C:\...\PP4E\Tools\cleanpyc.py process cwd
```

In this example, the Python script file itself lives in the directory *C:\...\PP4E\Tools*, but because it is run from *C:\temp*, it processes the files located in *C:\temp* (i.e., in the CWD, not in the script's home directory). To process files elsewhere with such a script, simply *cd* to the directory to be processed to change the CWD:

```
C:\temp> cd C:\PP4thEd\Examples  
C:\PP4thEd\Examples> python C:\...\PP4E\Tools\cleanpyc.py process cwd
```

Because the CWD is always implied, a *cd* command tells the script which directory to process in no less certain terms than passing a directory name to the script explicitly, like this (portability note: you may need to add quotes around the **.py* in this and other command-line examples to prevent it from being expanded in some Unix shells):

```
C:\...\PP4E\Tools> python find.py *.py C:\temp process named dir
```

In this command line, the CWD is the directory containing the script to be run (notice that the script filename has no directory path prefix); but since this script processes a directory named explicitly on the command line (*C:\temp*), the CWD is irrelevant. Finally, if we want to run such a script located in some other directory in order to process files located in yet another directory, we can simply give directory paths to both:

```
C:\temp> python C:\...\PP4E\Tools\find.py *.cxx C:\PP4thEd\Examples\PP4E
```

Here, the script has import visibility to files in its *PP4E\Tools* home directory and processes files in the directory named on the command line, but the CWD is something else entirely (*C:\temp*). This last form is more to type, of course, but watch for a variety of CWD and explicit script-path command lines like these in this book.

Command-Line Arguments

The *sys* module is also where Python makes available the words typed on the command that is used to start a Python script. These words are usually referred to as command-line arguments and show up in *sys.argv*, a built-in list of strings. C programmers may notice its similarity to the C *argv* array (an array of C strings). It's not much to look at interactively, because no command-line arguments are passed to start up Python in this mode:

```
>>> import sys  
>>> sys.argv  
['']
```

To really see what arguments are about, we need to run a script from the shell command line. [Example 3-1](#) shows an unreasonably simple one that just prints the `argv` list for inspection.

Example 3-1. PP4E\System\testargv.py

```
import sys  
print(sys.argv)
```

Running this script prints the command-line arguments list; note that the first item is always the name of the executed Python script file itself, no matter how the script was started (see “[Executable Scripts on Unix](#)” on page 108).

```
C:\...\PP4E\System> python testargv.py  
['testargv.py']  
  
C:\...\PP4E\System> python testargv.py spam eggs cheese  
['testargv.py', 'spam', 'eggs', 'cheese']  
  
C:\...\PP4E\System> python testargv.py -i data.txt -o results.txt  
['testargv.py', '-i', 'data.txt', '-o', 'results.txt']
```

The last command here illustrates a common convention. Much like function arguments, command-line options are sometimes passed by position and sometimes by name using a “-name value” word pair. For instance, the pair `-i data.txt` means the `-i` option’s value is `data.txt` (e.g., an input filename). Any words can be listed, but programs usually impose some sort of structure on them.

Command-line arguments play the same role in programs that function arguments do in functions: they are simply a way to pass information to a program that can vary per program run. Because they don’t have to be hardcoded, they allow scripts to be more generally useful. For example, a file-processing script can use a command-line argument as the name of the file it should process; see [Chapter 2’s more.py script](#) ([Example 2-1](#)) for a prime example. Other scripts might accept processing mode flags, Internet addresses, and so on.

Parsing Command-Line Arguments

Once you start using command-line arguments regularly, though, you’ll probably find it inconvenient to keep writing code that fishes through the list looking for words. More typically, programs translate the arguments list on startup into structures that are more conveniently processed. Here’s one way to do it: the script in [Example 3-2](#) scans the `argv` list looking for `-optionname optionvalue` word pairs and stuffs them into a dictionary by option name for easy retrieval.

Example 3-2. PP4E\System\testargv2.py

```
"collect command-line options in a dictionary"

def getopt(argv):
    opts = {}
    while argv:
        if argv[0][0] == '-':
            opts[argv[0]] = argv[1]           # find "-name value" pairs
            argv = argv[2:]                 # dict key is "-name" arg
        else:
            argv = argv[1:]
    return opts

if __name__ == '__main__':
    from sys import argv             # example client code
    myargs = getopt(argv)
    if '-i' in myargs:
        print(myargs['-i'])
    print(myargs)
```

You might import and use such a function in all your command-line tools. When run by itself, this file just prints the formatted argument dictionary:

```
C:\...\PP4E\System> python testargv2.py
{}

C:\...\PP4E\System> python testargv2.py -i data.txt -o results.txt
data.txt
{'o': 'results.txt', 'i': 'data.txt'}
```

Naturally, we could get much more sophisticated here in terms of argument patterns, error checking, and the like. For more complex command lines, we could also use command-line processing tools in the Python standard library to parse arguments:

- The `getopt` module, modeled after a Unix/C utility of the same name
- The `optparse` module, a newer alternative, generally considered to be more powerful

Both of these are documented in Python's library manual, which also provides usage examples which we'll defer to here in the interest of space. In general, the more configurable your scripts, the more you must invest in command-line processing logic complexity.

Executable Scripts on Unix

Unix and Linux users: you can also make text files of Python source code directly executable by adding a special line at the top with the path to the Python interpreter and giving the file executable permission. For instance, type this code into a text file called *myscript*:

```
#!/usr/bin/python
print('And nice red uniforms')
```

The first line is normally taken as a comment by Python (it starts with a #); but when this file is run, the operating system sends lines in this file to the interpreter listed after #! in line 1. If this file is made directly executable with a shell command of the form `chmod +x myscript`, it can be run directly without typing `python` in the command, as though it were a binary executable program:

```
% myscript a b c  
And nice red uniforms
```

When run this way, `sys.argv` will still have the script's name as the first word in the list: `["myscript", "a", "b", "c"]`, exactly as if the script had been run with the more explicit and portable command form `python myscript a b c`. Making scripts directly executable is actually a Unix trick, not a Python feature, but it's worth pointing out that it can be made a bit less machine dependent by listing the Unix `env` command at the top instead of a hardcoded path to the Python executable:

```
#!/usr/bin/env python  
print('Wait for it...')
```

When coded this way, the operating system will employ your environment variable settings to locate your Python interpreter (your `PATH` variable, on most platforms). If you run the same script on many machines, you need only change your environment settings on each machine (you don't need to edit Python script code). Of course, you can always run Python files with a more explicit command line:

```
% python myscript a b c
```

This assumes that the `python` interpreter program is on your system's search path setting (otherwise, you need to type its full path), but it works on any Python platform with a command line. Since this is more portable, I generally use this convention in the book's examples, but consult your Unix manpages for more details on any of the topics mentioned here. Even so, these special `#!` lines will show up in many examples in this book just in case readers want to run them as executables on Unix or Linux; on other platforms, they are simply ignored as Python comments.

Note that on recent flavors of Windows, you can usually also type a script's filename directly (without the word `python`) to make it go, and you don't have to add a `#!` line at the top. Python uses the Windows registry on this platform to declare itself as the program that opens files with Python extensions (`.py` and others). This is also why you can launch files on Windows by clicking on them.

Shell Environment Variables

Shell variables, sometimes known as environment variables, are made available to Python scripts as `os.environ`, a Python dictionary-like object with one entry per variable setting in the shell. Shell variables live outside the Python system; they are often set at your system prompt or within startup files or control-panel GUIs and typically serve as system-wide configuration inputs to programs.

In fact, by now you should be familiar with a prime example: the `PYTHONPATH` module search path setting is a shell variable used by Python to import modules. By setting it once in your operating system, its value is available every time a Python program is run. Shell variables can also be set by programs to serve as inputs to other programs in an application; because their values are normally inherited by spawned programs, they can be used as a simple form of interprocess communication.

Fetching Shell Variables

In Python, the surrounding shell environment becomes a simple preset object, not special syntax. Indexing `os.environ` by the desired shell variable's name string (e.g., `os.environ['USER']`) is the moral equivalent of adding a dollar sign before a variable name in most Unix shells (e.g., `$USER`), using surrounding percent signs on DOS (`%USER%`), and calling `getenv("USER")` in a C program. Let's start up an interactive session to experiment (run in Python 3.1 on a Windows 7 laptop):

```
>>> import os
>>> os.environ.keys()
KeysView(<os._Environ object at 0x013B8C70>)

>>> list(os.environ.keys())
['TMP', 'COMPUTERNAME', 'USERDOMAIN', 'PSMODULEPATH', 'COMMONPROGRAMFILES',
...many more deleted...
'NUMBER_OF_PROCESSORS', 'PROCESSOR_LEVEL', 'USERPROFILE', 'OS', 'PUBLIC', 'QTJAVA']

>>> os.environ['TEMP']
'C:\\\\Users\\\\mark\\\\AppData\\\\Local\\\\Temp'
```

Here, the `keys` method returns an iterable of assigned variables, and indexing fetches the value of the shell variable `TEMP` on Windows. This works the same way on Linux, but other variables are generally preset when Python starts up. Since we know about `PYTHONPATH`, let's peek at its setting within Python to verify its content (as I wrote this, mine was set to the root of the book examples tree for this fourth edition, as well as a temporary development location):

```
>>> os.environ['PYTHONPATH']
'C:\\PP4thEd\\\\Examples;C:\\Users\\\\Mark\\\\temp'

>>> for srcdir in os.environ['PYTHONPATH'].split(os.pathsep):
...     print(srcdir)
...
C:\\PP4thEd\\\\Examples
C:\\Users\\\\Mark\\\\temp

>>> import sys
>>> sys.path[:3]
[ '', 'C:\\PP4thEd\\\\Examples', 'C:\\Users\\\\Mark\\\\temp' ]
```

`PYTHONPATH` is a string of directory paths separated by whatever character is used to separate items in such paths on your platform (e.g., ; on DOS/Windows, : on Unix and Linux). To split it into its components, we pass to the `split` string method an

`os.pathsep` delimiter—a portable setting that gives the proper separator for the underlying machine. As usual, `sys.path` is the actual search path at runtime, and reflects the result of merging in the `PYTHONPATH` setting after the current directory.

Changing Shell Variables

Like normal dictionaries, the `os.environ` object supports both key indexing and *assignment*. As for dictionaries, assignments change the value of the key:

```
>>> os.environ['TEMP']
'C:\\\\Users\\\\mark\\\\AppData\\\\Local\\\\Temp'
>>> os.environ['TEMP'] = r'c:\\temp'
>>> os.environ['TEMP']
'c:\\temp'
```

But something extra happens here. In all recent Python releases, values assigned to `os.environ` keys in this fashion are automatically *exported* to other parts of the application. That is, key assignments change both the `os.environ` object in the Python program as well as the associated variable in the enclosing *shell* environment of the running program’s process. Its new value becomes visible to the Python program, all linked-in C modules, and any programs spawned by the Python process.

Internally, key assignments to `os.environ` call `os.putenv`—a function that changes the shell variable outside the boundaries of the Python interpreter. To demonstrate how this works, we need a couple of scripts that set and fetch shell variables; the first is shown in [Example 3-3](#).

Example 3-3. PP4E\System\Environment\setenv.py

```
import os
print('setenv...', end=' ')
print(os.environ['USER'])                      # show current shell variable value

os.environ['USER'] = 'Brian'                     # runs os.putenv behind the scenes
os.system('python echoenv.py')

os.environ['USER'] = 'Arthur'                   # changes passed to spawned programs
os.system('python echoenv.py')                  # and linked-in C library modules

os.environ['USER'] = input('?')
print(os.popen('python echoenv.py').read())
```

This `setenv.py` script simply changes a shell variable, `USER`, and spawns another script that echoes this variable’s value, as shown in [Example 3-4](#).

Example 3-4. PP4E\System\Environment\echoenv.py

```
import os
print('echoenv...', end=' ')
print('Hello,', os.environ['USER'])
```

No matter how we run `echoenv.py`, it displays the value of `USER` in the enclosing shell; when run from the command line, this value comes from whatever we've set the variable to in the shell itself:

```
C:\...\PP4E\System\Environment> set USER=Bob  
C:\...\PP4E\System\Environment> python echoenv.py  
echoenv... Hello, Bob
```

When spawned by another script such as `setenv.py` using the `os.system` and `os.popen` tools we met earlier, though, `echoenv.py` gets whatever `USER` settings its parent program has made:

```
C:\...\PP4E\System\Environment> python setenv.py  
setenv... Bob  
echoenv... Hello, Brian  
echoenv... Hello, Arthur  
?Gumby  
echoenv... Hello, Gumby  
  
C:\...\PP4E\System\Environment> echo %USER%  
Bob
```

This works the same way on Linux. In general terms, a spawned program always *inherits* environment settings from its parents. *Spawned* programs are programs started with Python tools such as `os.spawnv`, the `os.fork/exec` combination on Unix-like platforms, and `os.popen`, `os.system`, and the `subprocess` module on a variety of platforms. All programs thus launched get the environment variable settings that exist in the parent at launch time.*

From a larger perspective, setting shell variables like this before starting a new program is one way to pass information into the new program. For instance, a Python configuration script might tailor the `PYTHONPATH` variable to include custom directories just before launching another Python script; the launched script will have the custom search path in its `sys.path` because shell variables are passed down to children (in fact, watch for such a launcher script to appear at the end of [Chapter 6](#)).

Shell Variable Fine Points: Parents, `putenv`, and `getenv`

Notice the last command in the preceding example—the `USER` variable is back to its original value after the top-level Python program exits. Assignments to `os.environ` keys are passed outside the interpreter and *down* the spawned programs chain, but never back *up* to parent program processes (including the system shell). This is also true in C programs that use the `putenv` library call, and it isn't a Python limitation per se.

* This is by default. Some program-launching tools also let scripts pass environment settings that are different from their own to child programs. For instance, the `os.spawnve` call is like `os.spawnv`, but it accepts a dictionary argument representing the shell environment to be passed to the started program. Some `os.exec*` variants (ones with an “e” at the end of their names) similarly accept explicit environments; see the `os.exec*` call formats in [Chapter 5](#) for more details.

It's also likely to be a nonissue if a Python script is at the top of your application. But keep in mind that shell settings made within a program usually endure only for that program's run and for the run of its spawned children. If you need to export a shell variable setting so that it lives on after Python exits, you may be able to find platform-specific extensions that do this; search <http://www.python.org> or the Web at large.

Another subtlety: as implemented today, changes to `os.environ` automatically call `os.putenv`, which runs the `putenv` call in the C library if it is available on your platform to export the setting outside Python to any linked-in C code. However, although `os.environ` changes call `os.putenv`, direct calls to `os.putenv` do not update `os.environ` to reflect the change. Because of this, the `os.environ` mapping interface is generally preferred to `os.putenv`.

Also note that environment settings are loaded into `os.environ` on startup and not on each fetch; hence, changes made by linked-in C code after startup may not be reflected in `os.environ`. Python does have a more focused `os.getenv` call today, but it is simply translated into an `os.environ` fetch on most platforms (or all, in 3.X), not into a call to `getenv` in the C library. Most applications won't need to care, especially if they are pure Python code. On platforms without a `putenv` call, `os.environ` can be passed as a parameter to program startup tools to set the spawned program's environment.

Standard Streams

The `sys` module is also the place where the standard input, output, and error streams of your Python programs live; these turn out to be another common way for programs to communicate:

```
>>> import sys  
>>> for f in (sys.stdin, sys.stdout, sys.stderr): print(f)  
...  
<_io.TextIOWrapper name='<stdin>' encoding='cp437'>  
<_io.TextIOWrapper name='<stdout>' encoding='cp437'>  
<_io.TextIOWrapper name='<stderr>' encoding='cp437'>
```

The standard streams are simply preopened Python file objects that are automatically connected to your program's standard streams when Python starts up. By default, all of them are tied to the console window where Python (or a Python program) was started. Because the `print` and `input` built-in functions are really nothing more than user-friendly interfaces to the standard output and input streams, they are similar to using `stdout` and `stdin` in `sys` directly:

```
>>> print('hello stdout world')  
hello stdout world  
  
>>> sys.stdout.write('hello stdout world' + '\n')  
hello stdout world  
19
```

```
>>> input('hello stdin world>')
hello stdin world>spam
'spam'

>>> print('hello stdin world>'); sys.stdin.readline()[:-1]
hello stdin world>
eggs
'eggs'
```

Standard Streams on Windows

Windows users: if you click a .py Python program’s filename in a Windows file explorer to start it (or launch it with `os.system`), a DOS console window automatically pops up to serve as the program’s standard stream. If your program makes windows of its own, you can avoid this console pop-up window by naming your program’s source-code file with a .pyw extension, not with a .py extension. The .pyw extension simply means a .py source file without a DOS pop up on Windows (it uses Windows registry settings to run a custom version of Python). A .pyw file may also be imported as usual.

Also note that because printed output goes to this DOS pop up when a program is clicked, scripts that simply print text and exit will generate an odd “flash”—the DOS console box pops up, output is printed into it, and the pop up goes away immediately (not the most user-friendly of features!). To keep the DOS pop-up box around so that you can read printed output, simply add an `input()` call at the bottom of your script to pause for an Enter key press before exiting.

Redirecting Streams to Files and Programs

Technically, standard output (and `print`) text appears in the console window where a program was started, standard input (and `input`) text comes from the keyboard, and standard error text is used to print Python error messages to the console window. At least that’s the default. It’s also possible to *redirect* these streams both to files and to other programs at the system shell, as well as to arbitrary objects within a Python script. On most systems, such redirections make it easy to reuse and combine general-purpose command-line utilities.

Redirection is useful for things like canned (precoded) test inputs: we can apply a single test script to any set of inputs by simply redirecting the standard input stream to a different file each time the script is run. Similarly, redirecting the standard output stream lets us save and later analyze a program’s output; for example, testing systems might compare the saved standard output of a script with a file of expected output to detect failures.

Although it’s a powerful paradigm, redirection turns out to be straightforward to use. For instance, consider the simple read-evaluate-print loop program in [Example 3-5](#).

```

Example 3-5. PP4E\System\Streams\teststreams.py
"read numbers till eof and show squares"

def interact():
    print('Hello stream world')                      # print sends to sys.stdout
    while True:
        try:
            reply = input('Enter a number>')          # input reads sys.stdin
        except EOFError:
            break                                     # raises an except on eof
        else:
            num = int(reply)                         # input given as a string
            print("%d squared is %d" % (num, num ** 2))
    print('Bye')

if __name__ == '__main__':
    interact()                                     # when run, not imported

```

As usual, the `interact` function here is automatically executed when this file is run, not when it is imported. By default, running this file from a system command line makes that standard stream appear where you typed the Python command. The script simply reads numbers until it reaches end-of-file in the standard input stream (on Windows, end-of-file is usually the two-key combination Ctrl-Z; on Unix, type Ctrl-D instead[†]):

```

C:\...\PP4E\System\Streams> python teststreams.py
Hello stream world
Enter a number>12
12 squared is 144
Enter a number>10
10 squared is 100
Enter a number>^Z
Bye

```

But on both Windows and Unix-like platforms, we can redirect the standard input stream to come from a file with the <*filename*> shell syntax. Here is a command session in a DOS console box on Windows that forces the script to read its input from a text file, *input.txt*. It's the same on Linux, but replace the DOS `type` command with a Unix `cat` command:

```

C:\...\PP4E\System\Streams> type input.txt
8
6

C:\...\PP4E\System\Streams> python teststreams.py < input.txt
Hello stream world

```

[†] Notice that `input` raises an exception to signal end-of-file, but file read methods simply return an empty string for this condition. Because `input` also strips the end-of-line character at the end of lines, an empty string result means an empty line, so an exception is necessary to specify the end-of-file condition. File read methods retain the end-of-line character and denote an empty line as "`\n`" instead of "`"`". This is one way in which reading `sys.stdin` directly differs from `input`. The latter also accepts a prompt string that is automatically printed before input is accepted.

```
Enter a number>8 squared is 64
Enter a number>6 squared is 36
Enter a number>Bye
```

Here, the *input.txt* file automates the input we would normally type interactively—the script reads from this file rather than from the keyboard. Standard output can be similarly redirected to go to a file with the `> filename` shell syntax. In fact, we can combine input and output redirection in a single command:

```
C:\...\PP4E\System\Streams> python teststreams.py < input.txt > output.txt

C:\...\PP4E\System\Streams> type output.txt
Hello stream world
Enter a number>8 squared is 64
Enter a number>6 squared is 36
Enter a number>Bye
```

This time, the Python script’s input and output are both mapped to text files, not to the interactive console session.

Chaining programs with pipes

On Windows and Unix-like platforms, it’s also possible to send the standard output of one program to the standard input of another using the `|` shell character between two commands. This is usually called a “pipe” operation because the shell creates a pipeline that connects the output and input of two commands. Let’s send the output of the Python script to the standard `more` command-line program’s input to see how this works:

```
C:\...\PP4E\System\Streams> python teststreams.py < input.txt | more

Hello stream world
Enter a number>8 squared is 64
Enter a number>6 squared is 36
Enter a number>Bye
```

Here, `teststreams`’s standard input comes from a file again, but its output (written by `print` calls) is sent to another program, not to a file or window. The receiving program is `more`, a standard command-line paging program available on Windows and Unix-like platforms. Because Python ties scripts into the standard stream model, though, Python scripts can be used on both ends. One Python script’s output can always be piped into another Python script’s input:

```
C:\...\PP4E\System\Streams> type writer.py
print("Help! Help! I'm being repressed!")
print(42)

C:\...\PP4E\System\Streams> type reader.py
print('Got this: "%s" % input())
import sys
data = sys.stdin.readline()[:-1]
print('The meaning of life is', data, int(data) * 2)
```

```
C:\...\PP4E\System\Streams> python writer.py
Help! Help! I'm being repressed!
42

C:\...\PP4E\System\Streams> python writer.py | python reader.py
Got this: "Help! Help! I'm being repressed!"
The meaning of life is 42 84
```

This time, two Python programs are connected. Script `reader` gets input from script `writer`; both scripts simply read and write, oblivious to stream mechanics. In practice, such chaining of programs is a simple form of cross-program communications. It makes it easy to *reuse* utilities written to communicate via `stdin` and `stdout` in ways we never anticipated. For instance, a Python program that sorts `stdin` text could be applied to any data source we like, including the output of other scripts. Consider the Python command-line utility scripts in Examples 3-6 and 3-7 which sort and sum lines in the standard input stream.

Example 3-6. PP4E\System\Streams\sorter.py

```
import sys
lines = sys.stdin.readlines()          # or sorted(sys.stdin)
lines.sort()                          # sort stdin input lines,
for line in lines: print(line, end='') # send result to stdout
                                    # for further processing
```

Example 3-7. PP4E\System\Streams\adder.py

```
import sys
sum = 0
while True:
    try:
        line = input()                # or call sys.stdin.readlines()
    except EOFError:
        break                         # or for line in sys.stdin:
                                         # input strips \n at end
    else:
        sum += int(line)             # was sting.atoi() in 2nd ed
print(sum)
```

We can apply such general-purpose tools in a variety of ways at the shell command line to sort and sum arbitrary files and program outputs (Windows note: on my prior XP machine and Python 2.X, I had to type “`python file.py`” here, not just “`file.py`,” or else the input redirection failed; with Python 3.X on Windows 7 today, either form works):

```
C:\...\PP4E\System\Streams> type data.txt
123
000
999
042

C:\...\PP4E\System\Streams> python sorter.py < data.txt           sort a file
000
042
123
999
```

```
C:\...\PP4E\System\Streams> python adder.py < data.txt           sum file
1164

C:\...\PP4E\System\Streams> type data.txt | python adder.py        sum type output
1164

C:\...\PP4E\System\Streams> type writer2.py
for data in (123, 0, 999, 42):
    print('%03d' % data)

C:\...\PP4E\System\Streams> python writer2.py | python sorter.py   sort py output
000
042
123
999

C:\...\PP4E\System\Streams> writer2.py | sorter.py                shorter form
...same output as prior command on Windows...

C:\...\PP4E\System\Streams> python writer2.py | python sorter.py | python adder.py
1164
```

The last command here connects three Python scripts by standard streams—the output of each prior script is fed to the input of the next via pipeline shell syntax.

Coding alternatives for adders and sorters

A few coding pointers here: if you look closely, you'll notice that *sorter.py* reads all of *stdin* at once with the *readlines* method, but *adder.py* reads one line at a time. If the input source is another program, some platforms run programs connected by pipes in *parallel*. On such systems, reading line by line works better if the data streams being passed are large, because readers don't have to wait until writers are completely finished to get busy processing data. Because *input* just reads *stdin*, the line-by-line scheme used by *adder.py* can always be coded with manual *sys.stdin.read* too:

```
C:\...\PP4E\System\Streams> type adder2.py
import sys
sum = 0
while True:
    line = sys.stdin.readline()
    if not line: break
    sum += int(line)
print(sum)
```

This version utilizes the fact that *int* allows the digits to be surrounded by whitespace (*readline* returns a line including its *\n*, but we don't have to use *[:-1]* or *rstrip()* to remove it for *int*). In fact, we can use Python's more recent file iterators to achieve the same effect—the *for* loop, for example, automatically grabs one line each time through when we iterate over a file object directly (more on file iterators in the next chapter):

```
C:\...\PP4E\System\Streams> type adder3.py
import sys
sum = 0
```

```
for line in sys.stdin: sum += int(line)
print(sum)
```

Changing `sorter` to read line by line this way may not be a big performance boost, though, because the list `sort` method requires that the list already be complete. As we'll see in [Chapter 18](#), manually coded sort algorithms are generally prone to be much slower than the Python list sorting method.

Interestingly, these two scripts can also be coded in a much more compact fashion in Python 2.4 and later by using the new `sorted` built-in function, generator expressions, and file iterators. The following work the same way as the originals, with noticeably less source-file real estate:

```
C:\...\PP4E\System\Streams> type sorterSmall.py
import sys
for line in sorted(sys.stdin): print(line, end='')

C:\...\PP4E\System\Streams> type adderSmall.py
import sys
print(sum(int(line) for line in sys.stdin))
```

In its argument to `sum`, the latter of these employs a generator expression, which is much like a list comprehension, but results are returned one at a time, not in a physical list. The net effect is space optimization. For more details, see a core language resource, such as the book [Learning Python](#).

Redirected Streams and User Interaction

Earlier in this section, we piped `teststreams.py` output into the standard `more` command-line program with a command like this:

```
C:\...\PP4E\System\Streams> python teststreams.py < input.txt | more
```

But since we already wrote our own “more” paging utility in Python in the preceding chapter, why not set it up to accept input from `stdin` too? For example, if we change the last three lines of the `more.py` file listed as [Example 2-1](#) in the prior chapter...

```
if __name__ == '__main__':
    import sys
    if len(sys.argv) == 1:
        more(sys.stdin.read())
    else:
        more(open(sys.argv[1]).read())
```

...it almost seems as if we should be able to redirect the standard output of `teststreams.py` into the standard input of `more.py`:

```
C:\...\PP4E\System\Streams> python teststreams.py < input.txt | python ..\more.py
Hello stream world
Enter a number>8 squared is 64
Enter a number>6 squared is 36
Enter a number>Bye
```

This technique generally works for Python scripts. Here, *teststreams.py* takes input from a file again. And, as in the last section, one Python program’s output is piped to another’s input—the *more.py* script in the parent (..) directory.

But there’s a subtle problem lurking in the preceding *more.py* command. Really, chaining worked there only by sheer luck: if the first script’s output is long enough that *more* has to ask the user if it should continue, the script will utterly fail (specifically, when *input* for user interaction triggers `EOFError`).

The problem is that the augmented *more.py* uses `stdin` for two disjointed purposes. It reads a reply from an interactive user on `stdin` by calling *input*, but now it *also* accepts the main input text on `stdin`. When the `stdin` stream is really redirected to an input file or pipe, we can’t use it to input a reply from an interactive user; it contains only the text of the input source. Moreover, because `stdin` is redirected before the program even starts up, there is no way to know what it meant prior to being redirected in the command line.

If we intend to accept input on `stdin` *and* use the console for user interaction, we have to do a bit more: we would also need to use special interfaces to read user replies from a keyboard directly, instead of standard input. On Windows, Python’s standard library `msvcrt` module provides such tools; on many Unix-like platforms, reading from device file `/dev/tty` will usually suffice.

Since this is an arguably obscure use case, we’ll delegate a complete solution to a suggested exercise. [Example 3-8](#) shows a Windows-only modified version of the *more* script that pages the standard input stream if called with no arguments, but also makes use of lower-level and platform-specific tools to converse with a user at a keyboard if needed.

Example 3-8. PP4E\System\Streams\moreplus.py

```
"""
split and interactively page a string, file, or stream of
text to stdout; when run as a script, page stdin or file
whose name is passed on cmdline; if input is stdin, can't
use it for user reply--use platform-specific tools or GUI;
"""

import sys

def getreply():
    """
    read a reply key from an interactive user
    even if stdin redirected to a file or pipe
    """
    if sys.stdin.isatty():                      # if stdin is console
        return input('?')                        # read reply line from stdin
    else:
        if sys.platform[:3] == 'win':           # if stdin was redirected
            import msvcrt                         # can't use to ask a user
            msvcrt.putch(b'?')
```

```

key = msvcrt.getche()                      # use windows console tools
msvcrt.putch(b'\n')                        # getch() does not echo key
return key
else:
    assert False, 'platform not supported'
#linux?: open('/dev/tty').readline()[:-1]

def more(text, numlines=10):
"""
page multiline string to stdout
"""

lines = text.splitlines()
while lines:
    chunk = lines[:numlines]
    lines = lines[numlines:]
    for line in chunk: print(line)
    if lines and getreply() not in [b'y', b'Y']: break

if __name__ == '__main__':                   # when run, not when imported
    if len(sys.argv) == 1:                  # if no command-line arguments
        more(sys.stdin.read())            # page stdin, no inputs
    else:
        more(open(sys.argv[1]).read())     # else page filename argument

```

Most of the new code in this version shows up in its `getreply` function. The file's `isatty` method tells us whether `stdin` is connected to the console; if it is, we simply read replies on `stdin` as before. Of course, we have to add such extra logic only to scripts that intend to interact with console users *and* take input on `stdin`. In a GUI application, for example, we could instead pop up dialogs, bind keyboard-press events to run callbacks, and so on (we'll meet GUIs in [Chapter 7](#)).

Armed with the reusable `getreply` function, though, we can safely run our `moreplus` utility in a variety of ways. As before, we can import and call this module's function directly, passing in whatever string we wish to page:

```

>>> from moreplus import more
>>> more(open('adderSmall.py').read())
import sys
print(sum(int(line) for line in sys.stdin))

```

Also as before, when run with a command-line *argument*, this script interactively pages through the named file's text:

```

C:\...\PP4E\System\Streams> python moreplus.py adderSmall.py
import sys
print(sum(int(line) for line in sys.stdin))

C:\...\PP4E\System\Streams> python moreplus.py moreplus.py
"""
split and interactively page a string, file, or stream of
text to stdout; when run as a script, page stdin or file
whose name is passed on cmdline; if input is stdin, can't
use it for user reply--use platform-specific tools or GUI;
"""

```

```
import sys
```

```
def getreply():
?n
```

But now the script also correctly pages text redirected into `stdin` from either a *file* or a command *pipe*, even if that text is too long to fit in a single display chunk. On most shells, we send such input via redirection or pipe operators like these:

```
C:\...\PP4E\System\Streams> python moreplus.py < moreplus.py
"""

```

```
split and interactively page a string, file, or stream of
text to stdout; when run as a script, page stdin or file
whose name is passed on cmdline; if input is stdin, can't
use it for user reply--use platform-specific tools or GUI;
"""

```

```
import sys
```

```
def getreply():
?n
```

```
C:\...\PP4E\System\Streams> type moreplus.py | python moreplus.py
"""

```

```
split and interactively page a string, file, or stream of
text to stdout; when run as a script, page stdin or file
whose name is passed on cmdline; if input is stdin, can't
use it for user reply--use platform-specific tools or GUI;
"""

```

```
import sys
```

```
def getreply():
?n
```

Finally, piping one Python script's output into this script's input now works as expected, without botching user interaction (and not just because we got lucky):

```
.....\System\Streams> python teststreams.py < input.txt | python moreplus.py
Hello stream world
Enter a number>8 squared is 64
Enter a number>6 squared is 36
Enter a number>Bye
```

Here, the standard *output* of one Python script is fed to the standard *input* of another Python script located in the same directory: *moreplus.py* reads the output of *teststreams.py*.

All of the redirections in such command lines work only because scripts don't care what standard input and output really are—interactive users, files, or pipes between programs. For example, when run as a script, *moreplus.py* simply reads stream `sys.stdin`; the command-line shell (e.g., DOS on Windows, csh on Linux) attaches such streams to the source implied by the command line before the script is started.

Scripts use the preopened `stdin` and `stdout` file objects to access those sources, regardless of their true nature.

And for readers keeping count, we have just run this single `more` pager script in four different ways: by importing and calling its function, by passing a filename command-line argument, by redirecting `stdin` to a file, and by piping a command’s output to `stdin`. By supporting importable functions, command-line arguments, and standard streams, Python system tools code can be reused in a wide variety of modes.

Redirecting Streams to Python Objects

All of the previous standard stream redirections work for programs written in any language that hook into the standard streams and rely more on the shell’s command-line processor than on Python itself. Command-line redirection syntax like `< filename` and `| program` is evaluated by the shell, not by Python. A more Pythonesque form of redirection can be done within scripts themselves by resetting `sys.stdin` and `sys.stdout` to file-like objects.

The main trick behind this mode is that anything that looks like a file in terms of methods will work as a standard stream in Python. The object’s interface (sometimes called its protocol), and not the object’s specific datatype, is all that matters. That is:

- Any object that provides file-like *read* methods can be assigned to `sys.stdin` to make input come from that object’s read methods.
- Any object that defines file-like *write* methods can be assigned to `sys.stdout`; all standard output will be sent to that object’s methods.

Because `print` and `input` simply call the `write` and `readline` methods of whatever objects `sys.stdout` and `sys.stdin` happen to reference, we can use this technique to both provide and intercept standard stream text with objects implemented as classes.

If you’ve already studied Python, you probably know that such plug-and-play compatibility is usually called *polymorphism*—it doesn’t matter what an object is, and it doesn’t matter what its interface does, as long as it provides the expected interface. This liberal approach to datatypes accounts for much of the conciseness and flexibility of Python code. Here, it provides a way for scripts to reset their own streams. [Example 3-9](#) shows a utility module that demonstrates this concept.

Example 3-9. PP4E\System\Streams\redirect.py

```
"""
```

```
file-like objects that save standard output text in a string and provide
standard input text from a string; redirect runs a passed-in function
with its output and input streams reset to these file-like class objects;
"""
```

```
import sys                      # get built-in modules

class Output:                    # simulated output file
```

```

def __init__(self):
    self.text = ''                      # empty string when created
def write(self, string):                 # add a string of bytes
    self.text += string
def writelines(self, lines):             # add each line in a list
    for line in lines: self.write(line)

class Input:                           # simulated input file
    def __init__(self, input=''):
        self.text = input               # default argument
    def read(self, size=None):         # save string when created
        if size == None:              # optional argument
            res, self.text = self.text, ''
        else:
            res, self.text = self.text[:size], self.text[size:]
        return res
    def readline(self):               # read N bytes, or all
        eoln = self.text.find('\n')      # find offset of next eoln
        if eoln == -1:                  # slice off through eoln
            res, self.text = self.text, ''
        else:
            res, self.text = self.text[:eoln+1], self.text[eoln+1:]
        return res

    def redirect(function, pargs, kargs, input):   # redirect stdin/out
        savestreams = sys.stdin, sys.stdout          # run a function object
        sys.stdin  = Input(input)                     # return stdout text
        sys.stdout = Output()
        try:
            result = function(*pargs, **kargs)       # run function with args
            output = sys.stdout.text
        finally:
            sys.stdin, sys.stdout = savestreams      # restore if exc or not
        return (result, output)                      # return result if no exc

```

This module defines two classes that masquerade as real files:

Output

Provides the write method interface (a.k.a. protocol) expected of output files but saves all output in an in-memory string as it is written.

Input

Provides the interface expected of input files, but provides input on demand from an in-memory string passed in at object construction time.

The `redirect` function at the bottom of this file combines these two objects to run a single function with input and output redirected entirely to Python class objects. The passed-in function to run need not know or care that its `print` and `input` function calls and `stdin` and `stdout` method calls are talking to a class rather than to a real file, pipe, or user.

To demonstrate, import and run the `interact` function at the heart of the `test streams` script of [Example 3-5](#) that we've been running from the shell (to use the

redirection utility function, we need to deal in terms of functions, not files). When run directly, the function reads from the keyboard and writes to the screen, just as if it were run as a program without redirection:

```
C:\...\PP4E\System\Streams> python
>>> from teststreams import interact
>>> interact()
Hello stream world
Enter a number>2
2 squared is 4
Enter a number>3
3 squared is 9
Enter a number^Z
Bye
>>>
```

Now, let's run this function under the control of the redirection function in *redirect.py* and pass in some canned input text. In this mode, the *interact* function takes its input from the string we pass in ('4\n5\n6\n')—three lines with explicit end-of-line characters), and the result of running the function is a tuple with its return value plus a string containing all the text written to the standard output stream:

```
>>> from redirect import redirect
>>> (result, output) = redirect(interact, (), {}, '4\n5\n6\n')
>>> print(result)
None
>>> output
'Hello stream world\nEnter a number>4 squared is 16\nEnter a number>5 squared
is 25\nEnter a number>6 squared is 36\nEnter a number>Bye\n'
```

The output is a single, long string containing the concatenation of all text written to standard output. To make this look better, we can pass it to *print* or split it up with the string object's *splitlines* method:

```
>>> for line in output.splitlines(): print(line)
...
Hello stream world
Enter a number>4 squared is 16
Enter a number>5 squared is 25
Enter a number>6 squared is 36
Enter a number>Bye
```

Better still, we can reuse the *more.py* module we wrote in the preceding chapter ([Example 2-1](#)); it's less to type and remember, and it's already known to work well (the following, like all cross-directory imports in this book's examples, assumes that the directory containing the PP4E root is on your module search path—change your PYTHON PATH setting as needed):

```
>>> from PP4E.System.more import more
>>> more(output)
Hello stream world
Enter a number>4 squared is 16
Enter a number>5 squared is 25
```

```
Enter a number>6 squared is 36
Enter a number>Bye
```

This is an artificial example, of course, but the techniques illustrated are widely applicable. For instance, it's straightforward to add a GUI interface to a program written to interact with a command-line user. Simply intercept standard output with an object such as the `Output` class instance shown earlier and throw the text string up in a window. Similarly, standard input can be reset to an object that fetches text from a graphical interface (e.g., a popped-up dialog box). Because classes are plug-and-play compatible with real files, we can use them in any tool that expects a file. Watch for a GUI stream-redirection module named `guiStreams` in [Chapter 10](#) that provides a concrete implementation of some of these ideas.

The `io.StringIO` and `io.BytesIO` Utility Classes

The prior section's technique of redirecting streams to objects proved so handy that now a standard library module automates the task for many use cases (though some use cases, such as GUIs, may still require more custom code). The standard library tool provides an object that maps a file object interface to and from in-memory strings. For example:

```
>>> from io import StringIO
>>> buff = StringIO()                      # save written text to a string
>>> buff.write('spam\n')
5
>>> buff.write('eggs\n')
5
>>> buff.getvalue()
'spam\neggs\n'

>>> buff = StringIO('ham\nspam\n')        # provide input from a string
>>> buff.readline()
'ham\n'
>>> buff.readline()
'spam\n'
>>> buff.readline()
''
```

As in the prior section, instances of `StringIO` objects can be assigned to `sys.stdin` and `sys.stdout` to redirect streams for `input` and `print` calls and can be passed to any code that was written to expect a real file object. Again, in Python, the object *interface*, not the concrete datatype, is the name of the game:

```
>>> from io import StringIO
>>> import sys
>>> buff = StringIO()

>>> temp = sys.stdout
>>> sys.stdout = buff
>>> print(42, 'spam', 3.141)             # or print(..., file=buff)
```

```
>>> sys.stdout = temp                      # restore original stream
>>> buff.getvalue()
'42 spam 3.141\n'
```

Note that there is also an `io.BytesIO` class with similar behavior, but which maps file operations to an in-memory bytes buffer, instead of a `str` string:

```
>>> from io import BytesIO
>>> stream = BytesIO()
>>> stream.write(b'spam')
>>> stream.getvalue()
b'spam'

>>> stream = BytesIO(b'dspam')
>>> stream.read()
b'dspam'
```

Due to the sharp distinction that Python 3X draws between text and binary data, this alternative may be better suited for scripts that deal with binary data. We'll learn more about the text-versus-binary issue in the next chapter when we explore files.

Capturing the `stderr` Stream

We've been focusing on `stdin` and `stdout` redirection, but `stderr` can be similarly reset to files, pipes, and objects. Although some shells support this, it's also straightforward within a Python script. For instance, assigning `sys.stderr` to another instance of a class such as `Output` or a `StringIO` object in the preceding section's example allows your script to intercept text written to standard error, too.

Python itself uses standard error for error message text (and the IDLE GUI interface intercepts it and colors it red by default). However, no higher-level tools for standard error do what `print` and `input` do for the output and input streams. If you wish to print to the error stream, you'll want to call `sys.stderr.write()` explicitly or read the next section for a `print` call trick that makes this easier.

Redirecting standard errors from a shell command line is a bit more complex and less portable. On most Unix-like systems, we can usually capture `stderr` output by using shell-redirection syntax of the form `command > output 2>&1`. This may not work on some platforms, though, and can even vary per Unix shell; see your shell's manpages for more details.

Redirection Syntax in Print Calls

Because resetting the stream attributes to new objects was so popular, the Python `print` built-in is also extended to include an explicit file to which output is to be sent. A statement of this form:

```
print(stuff, file=afile)                  # afile is an object, not a string name
```

prints `stuff` to `afile` instead of to `sys.stdout`. The net effect is similar to simply assigning `sys.stdout` to an object, but there is no need to save and restore in order to return to the original output stream (as shown in the section on redirecting streams to objects). For example:

```
import sys
print('spam' * 2, file=sys.stderr)
```

will send text the standard error stream object rather than `sys.stdout` for the duration of this single print call only. The next normal print statement (without `file`) prints to standard output as usual. Similarly, we can use either our custom class or the standard library’s class as the output file with this hook:

```
>>> from io import StringIO
>>> buff = StringIO()
>>> print(42, file=buff)
>>> print('spam', file=buff)
>>> print(buff.getvalue())
42
spam

>>> from redirect import Output
>>> buff = Output()
>>> print(43, file=buff)
>>> print('eggs', file=buff)
>>> print(buff.text)
43
eggs
```

Other Redirection Options: `os.popen` and `subprocess` Revisited

Near the end of the preceding chapter, we took a first look at the built-in `os.popen` function and its `subprocess.Popen` relative, which provide a way to redirect another command’s streams from within a Python program. As we saw, these tools can be used to run a shell command line (a string we would normally type at a DOS or `csh` prompt) but also provide a Python file-like object connected to the command’s output stream—reading the file object allows a script to read another program’s output. I suggested that these tools may be used to tap into input streams as well.

Because of that, the `os.popen` and `subprocess` tools are another way to redirect streams of spawned programs and are close cousins to some of the techniques we just met. Their effect is much like the `shell | command-line pipe` syntax for redirecting streams to programs (in fact, their names mean “pipe open”), but they are run within a script and provide a file-like interface to piped streams. They are similar in spirit to the `redirect` function, but are based on running programs (not calling functions), and the command’s streams are processed in the spawning script as files (not tied to class objects). These tools redirect the streams of a program that a script starts, instead of redirecting the streams of the script itself.

Redirecting input or output with `os.popen`

In fact, by passing in the desired mode flag, we redirect either a spawned program's output *or* input streams to a file in the calling scripts, and we can obtain the spawned program's exit status code from the `close` method (`None` means "no error" here). To illustrate, consider the following two scripts:

```
C:\...\PP4E\System\Streams> type hello-out.py
print('Hello shell world')

C:\...\PP4E\System\Streams> type hello-in.py
inp = input()
open('hello-in.txt', 'w').write('Hello ' + inp + '\n')
```

These scripts can be run from a system shell window as usual:

```
C:\...\PP4E\System\Streams> python hello-out.py
Hello shell world

C:\...\PP4E\System\Streams> python hello-in.py
Brian

C:\...\PP4E\System\Streams> type hello-in.txt
Hello Brian
```

As we saw in the prior chapter, Python scripts can read *output* from other programs and scripts like these, too, using code like the following:

```
C:\...\PP4E\System\Streams> python
>>> import os
>>> pipe = os.popen('python hello-out.py')           # 'r' is default--read stdout
>>> pipe.read()
'Hello shell world\n'
>>> print(pipe.close())                            # exit status: None is good
None
```

But Python scripts can also provide *input* to spawned programs' standard input streams—passing a "w" mode argument, instead of the default "r", connects the returned object to the spawned program's input stream. What we write on the spawning end shows up as input in the program started:

```
>>> pipe = os.popen('python hello-in.py', 'w')      # 'w'--write to program stdin
>>> pipe.write('Gumby\n')
6
>>> pipe.close()                                     # \n at end is optional
>>> open('hello-in.txt').read()                      # output sent to a file
'Hello Gumby\n'
```

The `popen` call is also smart enough to run the command string as an independent process on platforms that support such a notion. It accepts an optional third argument that can be used to control buffering of written text, which we'll finesse here.

Redirecting input and output with subprocess

For even more control over the streams of spawned programs, we can employ the `subprocess` module we introduced in the preceding chapter. As we learned earlier, this module can emulate `os.popen` functionality, but it can also achieve feats such as bidirectional stream communication (accessing both a program's input and output) and tying the output of one program to the input of another.

For instance, this module provides multiple ways to spawn a program and get both its standard output text and exit status. Here are three common ways to leverage this module to start a program and redirect its *output* stream (recall from [Chapter 2](#) that you may need to pass a `shell=True` argument to `Popen` and `call` to make this section's examples work on Unix-like platforms as they are coded here):

```
C:\...\PP4E\System\Streams> python
>>> from subprocess import Popen, PIPE, call
>>> X = call('python hello-out.py')                                # convenience
Hello shell world
>>> X
0

>>> pipe = Popen('python hello-out.py', stdout=PIPE)
>>> pipe.communicate()[0]                                         # (stdout, stderr)
b'Hello shell world\r\n'
>>> pipe.returncode                                              # exit status
0

>>> pipe = Popen('python hello-out.py', stdout=PIPE)
>>> pipe.stdout.read()                                           # exit status
b'Hello shell world\r\n'
>>> pipe.wait()
```

The `call` in the first of these three techniques is just a convenience function (there are more of these which you can look up in the Python library manual), and the `communicate` in the second is roughly a convenience for the third (it sends data to `stdin`, reads data from `stdout` until end-of-file, and waits for the process to end):

Redirecting and connecting to the spawned program's *input* stream is just as simple, though a bit more complex than the `os.popen` approach with '`w`' file mode shown in the preceding section (as mentioned in the last chapter, `os.popen` is implemented with `subprocess`, and is thus itself just something of a convenience function today):

```
>>> pipe = Popen('python hello-in.py', stdin=PIPE)
>>> pipe.stdin.write(b'Pokey\n')
6
>>> pipe.stdin.close()
>>> pipe.wait()
0
>>> open('hello-in.txt').read()                                     # output sent to a file
'Hello Pokey\n'
```

In fact, we can use obtain *both the input and output* streams of a spawned program with this module. Let's reuse the simple writer and reader scripts we wrote earlier to demonstrate:

```
C:\...\PP4E\System\Streams> type writer.py
print("Help! Help! I'm being repressed!")
print(42)

C:\...\PP4E\System\Streams> type reader.py
print('Got this: "%s"' % input())
import sys
data = sys.stdin.readline()[:-1]
print('The meaning of life is', data, int(data) * 2)
```

Code like the following can both read from and write to the reader script—the pipe object has two file-like objects available as attached attributes, one connecting to the input stream, and one to the output (Python 2.X users might recognize these as equivalent to the tuple returned by the now-defunct `os.popen2`):

```
>>> pipe = Popen('python reader.py', stdin=PIPE, stdout=PIPE)
>>> pipe.stdin.write(b'Lumberjack\n')
11
>>> pipe.stdin.write(b'12\n')
3
>>> pipe.stdin.close()
>>> output = pipe.stdout.read()
>>> pipe.wait()
0
>>> output
b'Got this: "Lumberjack"\r\nThe meaning of life is 12 24\r\n'
```

As we'll learn in [Chapter 5](#), we have to be cautious when talking back and forth to a program like this; buffered output streams can lead to deadlock if writes and reads are interleaved, and we may eventually need to consider tools like the `Pexpect` utility as a workaround (more on this later).

Finally, even more exotic stream control is possible—the following *connects two programs*, by piping the output of one Python script into another, first with shell syntax, and then with the `subprocess` module:

```
C:\...\PP4E\System\Streams> python writer.py | python reader.py
Got this: "Help! Help! I'm being repressed!"
The meaning of life is 42 84
```

```
C:\...\PP4E\System\Streams> python
>>> from subprocess import Popen, PIPE
>>> p1 = Popen('python writer.py', stdout=PIPE)
>>> p2 = Popen('python reader.py', stdin=p1.stdout, stdout=PIPE)
>>> output = p2.communicate()[0]
>>> output
b'Got this: "Help! Help! I'm being repressed!"\r\nThe meaning of life is 42 84\r\n'
>>> p2.returncode
0
```

We can get close to this with `os.popen`, but that the fact that its pipes are read or write (and not both) prevents us from catching the second script's output in our code:

```
>>> import os
>>> p1 = os.popen('python writer.py', 'r')
>>> p2 = os.popen('python reader.py', 'w')
>>> p2.write( p1.read() )
36
>>> X = p2.close()
Got this: "Help! Help! I'm being repressed!"
The meaning of life is 42 84
>>> print(X)
None
```

From the broader perspective, the `os.popen` call and `subprocess` module are Python's portable equivalents of Unix-like shell syntax for redirecting the streams of spawned programs. The Python versions also work on Windows, though, and are the most platform-neutral way to launch another program from a Python script. The command-line strings you pass to them may vary per platform (e.g., a directory listing requires an `ls` on Unix but a `dir` on Windows), but the call itself works on all major Python platforms.

On Unix-like platforms, the combination of the calls `os.fork`, `os.pipe`, `os.dup`, and some `os.exec` variants can also be used to start a new independent program with streams connected to the parent program's streams. As such, it's yet another way to redirect streams and a low-level equivalent to tools such as `os.popen` (`os.fork` is available in Cygwin's Python on Windows).

Since these are all more advanced parallel processing tools, though, we'll defer further details on this front until [Chapter 5](#), especially its coverage of pipes and exit status codes. And we'll resurrect `subprocess` again in [Chapter 6](#), to code a regression tester that intercepts all *three* standard streams of spawned test scripts—inputs, outputs, and errors.

But first, [Chapter 4](#) continues our survey of Python system interfaces by exploring the tools available for processing files and directories. Although we'll be shifting focus somewhat, we'll find that some of what we've learned here will already begin to come in handy as general system-related tools. Spawning shell commands, for instance, provides ways to inspect directories, and the file interface we will expand on in the next chapter is at the heart of the stream processing techniques we have studied here.

Python Versus csh

If you are familiar with other common shell script languages, it might be useful to see how Python compares. Here is a simple script in a Unix shell language called `csh` that mails all the files in the current working directory with a suffix of `.py` (i.e., all Python source files) to a hopefully fictitious address:

```
#!/bin/csh
foreach x (*.py)
    echo $x
    mail eric@halfabee.com -s $x < $x
end
```

An equivalent Python script looks similar:

```
#!/usr/bin/python
import os, glob
for x in glob.glob('*.py'):
    print(x)
    os.system('mail eric@halfabee.com -s %s < %s' % (x, x))
```

but is slightly more verbose. Since Python, unlike `csh`, isn't meant just for shell scripts, system interfaces must be imported and called explicitly. And since Python isn't just a string-processing language, character strings must be enclosed in quotes, as in C.

Although this can add a few extra keystrokes in simple scripts like this, being a general-purpose language makes Python a better tool once we leave the realm of trivial programs. We could, for example, extend the preceding script to do things like transfer files by FTP, pop up a GUI message selector and status bar, fetch messages from an SQL database, and employ COM objects on Windows, all using standard Python tools.

Python scripts also tend to be more portable to other platforms than `csh`. For instance, if we used the Python SMTP interface module to send mail instead of relying on a Unix command-line mail tool, the script would run on any machine with Python and an Internet link (as we'll see in [Chapter 13](#), SMTP requires only sockets). And like C, we don't need `$` to evaluate variables; what else would you expect in a free language?

File and Directory Tools

“Erase Your Hard Drive in Five Easy Steps!”

This chapter continues our look at system interfaces in Python by focusing on file and directory-related tools. As you’ll see, it’s easy to process files and directory trees with Python’s built-in and standard library support. Because files are part of the core Python language, some of this chapter’s material is a review of file basics covered in books like *Learning Python*, Fourth Edition, and we’ll defer to such resources for more background details on some file-related concepts. For example, iteration, context managers, and the file object’s support for Unicode encodings are demonstrated along the way, but these topics are not repeated in full here. This chapter’s goal is to tell enough of the file story to get you started writing useful scripts.

File Tools

External files are at the heart of much of what we do with system utilities. For instance, a testing system may read its inputs from one file, store program results in another file, and check expected results by loading yet another file. Even user interface and Internet-oriented programs may load binary images and audio clips from files on the underlying computer. It’s a core programming concept.

In Python, the built-in `open` function is the primary tool scripts use to access the files on the underlying computer system. Since this function is an inherent part of the Python language, you may already be familiar with its basic workings. When called, the `open` function returns a new *file object* that is connected to the external file; the file object has methods that transfer data to and from the file and perform a variety of file-related operations. The `open` function also provides a *portable* interface to the underlying file-system—it works the same way on every platform on which Python runs.

Other file-related modules built into Python allow us to do things such as manipulate lower-level descriptor-based files (`os`); copy, remove, and move files and collections of files (`os` and `shutil`); store data and objects in files by key (`dbm` and `shelve`); and access

SQL databases (`sqlite3` and third-party add-ons). The last two of these categories are related to database topics, addressed in [Chapter 17](#).

In this section, we'll take a brief tutorial look at the built-in file object and explore a handful of more advanced file-related topics. As usual, you should consult either Python's library manual or reference books such as [Python Pocket Reference](#) for further details and methods we don't have space to cover here. Remember, for quick interactive help, you can also run `dir(file)` on an open file object to see an attributes list that includes methods; `help(file)` for general help; and `help(file.read)` for help on a specific method such as `read`, though the file object implementation in 3.1 provides less information for `help` than the library manual and other resources.

The File Object Model in Python 3.X

Just like the string types we noted in [Chapter 2](#), file support in Python 3.X is a bit richer than it was in the past. As we noted earlier, in Python 3.X `str` strings always represent Unicode text (ASCII or wider), and `bytes` and `bytearray` strings represent raw binary data. Python 3.X draws a similar and related distinction between files containing text and binary data:

- *Text files* contain Unicode text. In your script, text file content is always a `str` string—a sequence of characters (technically, Unicode “code points”). Text files perform the automatic line-end translations described in this chapter by default and automatically apply Unicode encodings to file content: they encode to and decode from raw binary bytes on transfers to and from the file, according to a provided or default encoding name. Encoding is trivial for ASCII text, but may be sophisticated in other cases.
- *Binary files* contain raw 8-bit bytes. In your script, binary file content is always a byte string, usually a `bytes` object—a sequence of small integers, which supports most `str` operations and displays as ASCII characters whenever possible. Binary files perform no translations of data when it is transferred to and from files: no line-end translations or Unicode encodings are performed.

In practice, text files are used for all truly text-related data, and binary files store items like packed binary data, images, audio files, executables, and so on. As a programmer you distinguish between the two file types in the mode string argument you pass to `open`: adding a “b” (e.g., `'rb'`, `'wb'`) means the file contains binary data. For coding new file content, use normal strings for text (e.g., `'spam'` or `bytes.decode()`) and byte strings for binary (e.g., `b'spam'` or `str.encode()`).

Unless your file scope is limited to ASCII text, the 3.X text/binary distinction can sometimes impact your code. Text files create and require `str` strings, and binary files use byte strings; because you cannot freely mix the two string types in expressions, you must choose file mode carefully. Many built-in tools we'll use in this book make the choice for us; the `struct` and `pickle` modules, for instance, deal in byte strings in 3.X,

and the `xml` package in Unicode `str`. You must even be aware of the 3.X text/binary distinction when using system tools like pipe descriptors and *sockets*, because they transfer data as byte strings today (though their content can be decoded and encoded as Unicode text if needed).

Moreover, because text-mode files require that content be decodable per a Unicode encoding scheme, you must read undecodable file content in binary mode, as byte strings (or catch Unicode exceptions in `try` statements and skip the file altogether). This may include both truly binary files as well as text files that use encodings that are nondefault and unknown. As we'll see later in this chapter, because `str` strings are always Unicode in 3.X, it's sometimes also necessary to select byte string mode for the names of files in directory tools such as `os.listdir`, `glob.glob`, and `os.walk` if they cannot be decoded (passing in byte strings essentially suppresses decoding).

In fact, we'll see examples where the Python 3.X distinction between `str` text and `bytes` binary pops up in tools beyond basic files throughout this book—in Chapters 5 and 12 when we explore sockets; in Chapters 6 and 11 when we'll need to ignore Unicode errors in file and directory searches; in Chapter 12, where we'll see how client-side Internet protocol modules such as FTP and email, which run atop sockets, imply file modes and encoding requirements; and more.

But just as for string types, although we will see some of these concepts in action in this chapter, we're going to take much of this story as a given here. File and string objects are core language material and are prerequisite to this text. As mentioned earlier, because they are addressed by a 45-page chapter in the book *Learning Python*, Fourth Edition, I won't repeat their coverage in full in this book. If you find yourself confused by the Unicode and binary file and string concepts in the following sections, I encourage you to refer to that text or other resources for more background information in this domain.

Using Built-in File Objects

Despite the text/binary dichotomy in Python 3.X, files are still very straightforward to use. For most purposes, in fact, the `open` built-in function and its `files` objects are all you need to remember to process files in your scripts. The file object returned by `open` has methods for reading data (`read`, `readline`, `readlines`); writing data (`write`, `writelines`); freeing system resources (`close`); moving to arbitrary positions in the file (`seek`); forcing data in output buffers to be transferred to disk (`flush`); fetching the underlying file handle (`fileno`); and more. Since the built-in file object is so easy to use, let's jump right into a few interactive examples.

Output files

To make a new file, call `open` with two arguments: the external *name* of the file to be created and a *mode* string `w` (short for `write`). To store data on the file, call the file object's `write` method with a string containing the data to store, and then call the `close` method

to close the file. File `write` calls return the number of characters or bytes written (which we'll sometimes omit in this book to save space), and as we'll see, `close` calls are often optional, unless you need to open and read the file again during the same program or session:

```
C:\temp> python
>>> file = open('data.txt', 'w')          # open output file object: creates
>>> file.write('Hello file world!\n')      # writes strings verbatim
18
>>> file.write('Bye    file world.\n')      # returns number chars/bytes written
18
>>> file.close()                         # closed on gc and exit too
```

And that's it—you've just generated a brand-new text file on your computer, regardless of the computer on which you type this code:

```
C:\temp> dir data.txt /B
data.txt

C:\temp> type data.txt
Hello file world!
Bye    file world.
```

There is nothing unusual about the new file; here, I use the DOS `dir` and `type` commands to list and display the new file, but it shows up in a file explorer GUI, too.

Opening. In the `open` function call shown in the preceding example, the first argument can optionally specify a complete directory path as part of the filename string. If we pass just a simple filename without a path, the file will appear in Python's current working directory. That is, it shows up in the place where the code is run. Here, the directory `C:\temp` on my machine is implied by the bare filename `data.txt`, so this actually creates a file at `C:\temp\data.txt`. More accurately, the filename is relative to the current working directory if it does not include a complete absolute directory path. See “[Current Working Directory](#)” on page 104 (Chapter 3), for a refresher on this topic.

Also note that when opening in `w` mode, Python either creates the external file if it does not yet exist or erases the file's current contents if it is already present on your machine (so be careful out there—you'll delete whatever was in the file before).

Writing. Notice that we added an explicit `\n` end-of-line character to lines written to the file; unlike the `print` built-in function, file object `write` methods write exactly what they are passed without adding any extra formatting. The string passed to `write` shows up character for character on the external file. In text files, data written may undergo line-end or Unicode translations which we'll describe ahead, but these are undone when the data is later read back.

Output files also sport a `writelines` method, which simply writes all of the strings in a list one at a time without adding any extra formatting. For example, here is a `write lines` equivalent to the two `write` calls shown earlier:

```
file.writelines(['Hello file world!\n', 'Bye    file world.\n'])
```

This call isn't as commonly used (and can be emulated with a simple `for` loop or other iteration tool), but it is convenient in scripts that save output in a list to be written later.

Closing. The file `close` method used earlier finalizes file contents and frees up system resources. For instance, closing forces buffered output data to be flushed out to disk. Normally, files are automatically closed when the file object is garbage collected by the interpreter (that is, when it is no longer referenced). This includes all remaining open files when the Python session or program exits. Because of that, `close` calls are often optional. In fact, it's common to see file-processing code in Python in this idiom:

```
open('somefile.txt', 'w').write("G'day Bruce\n")      # write to temporary object
open('somefile.txt', 'r').read()                      # read from temporary object
```

Since both these expressions make a temporary file object, use it immediately, and do not save a reference to it, the file object is reclaimed right after data is transferred, and is automatically closed in the process. There is usually no need for such code to call the `close` method explicitly.

In some contexts, though, you may wish to explicitly close anyhow:

- For one, because the Jython implementation relies on Java's garbage collector, you can't always be as sure about when files will be reclaimed as you can in standard Python. If you run your Python code with Jython, you may need to close manually if many files are created in a short amount of time (e.g. in a loop), in order to avoid running out of file resources on operating systems where this matters.
- For another, some IDEs, such as Python's standard IDLE GUI, may hold on to your file objects longer than you expect (in stack tracebacks of prior errors, for instance), and thus prevent them from being garbage collected as soon as you might expect. If you write to an output file in IDLE, be sure to explicitly close (or flush) your file if you need to reliably read it back during the same IDLE session. Otherwise, output buffers might not be flushed to disk and your file may be incomplete when read.
- And while it seems very unlikely today, it's not impossible that this auto-close on reclaim file feature could change in future. This is technically a feature of the file object's implementation, which may or may not be considered part of the language definition over time.

For these reasons, manual close calls are not a bad idea in nontrivial programs, even if they are technically not required. Closing is a generally harmless but robust habit to form.

Ensuring file closure: Exception handlers and context managers

Manual file close method calls are easy in straight-line code, but how do you ensure file closure when exceptions might kick your program beyond the point where the close call is coded? First of all, make sure you must—files close themselves when they are collected, and this will happen eventually, even when exceptions occur.

If closure is required, though, there are two basic alternatives: the `try` statement’s `finally` clause is the most general, since it allows you to provide general exit actions for any type of exceptions:

```
myfile = open(filename, 'w')
try:
    ...process myfile...
finally:
    myfile.close()
```

In recent Python releases, though, the `with` statement provides a more concise alternative for some specific objects and exit actions, including closing files:

```
with open(filename, 'w') as myfile:
    ...process myfile, auto-closed on statement exit...
```

This statement relies on the file object’s context manager: code automatically run both on statement entry and on statement exit regardless of exception behavior. Because the file object’s exit code closes the file automatically, this guarantees file closure whether an exception occurs during the statement or not.

The `with` statement is notably shorter (3 lines) than the `try/finally` alternative, but it’s also less general—`with` applies only to objects that support the context manager protocol, whereas `try/finally` allows arbitrary exit actions for arbitrary exception contexts. While some other object types have context managers, too (e.g., thread locks), `with` is limited in scope. In fact, if you want to remember just one exit actions option, `try/finally` is the most inclusive. Still, `with` yields less code for files that must be closed and can serve well in such specific roles. It can even save a line of code when no exceptions are expected (albeit at the expense of further nesting and indenting file processing logic):

```
myfile = open(filename, 'w')                      # traditional form
...process myfile...
myfile.close()

with open(filename) as myfile:                     # context manager form
    ...process myfile...
```

In Python 3.1 and later, this statement can also specify multiple (a.k.a. nested) context managers—any number of context manager items may be separated by commas, and multiple items work the same as nested `with` statements. In general terms, the 3.1 and later code:

```
with A() as a, B() as b:
    ...statements...
```

Runs the same as the following, which works in 3.1, 3.0, and 2.6:

```
with A() as a:
    with B() as b:
        ...statements...
```

For example, when the `with` statement block exits in the following, both files' exit actions are automatically run to close the files, regardless of exception outcomes:

```
with open('data') as fin, open('results', 'w') as fout:  
    for line in fin:  
        fout.write(transform(line))
```

Context manager-dependent code like this seems to have become more common in recent years, but this is likely at least in part because newcomers are accustomed to languages that require manual close calls in all cases. In most contexts there is no need to wrap all your Python file-processing code in `with` statements—the files object's auto-close-on-collection behavior often suffices, and manual close calls are enough for many other scripts. You should use the `with` or `try` options outlined here only if you must close, and only in the presence of potential exceptions. Since standard C Python automatically closes files on collection, though, neither option is required in many (and perhaps most) scripts.

Input files

Reading data from external files is just as easy as writing, but there are more methods that let us load data in a variety of modes. Input text files are opened with either a mode flag of `r` (for “read”) or no mode flag at all—it defaults to `r` if omitted, and it commonly is. Once opened, we can read the lines of a text file with the `readlines` method:

```
C:\temp> python  
->>> file = open('data.txt')  
->>> lines = file.readlines()  
->>> for line in lines:  
...     print(line, end='')  
...  
Hello file world!  
Bye file world.
```

The `readlines` method loads the entire contents of the file into memory and gives it to our scripts as a list of line strings that we can step through in a loop. In fact, there are many ways to read an input file:

`file.read()`

Returns a string containing all the characters (or bytes) stored in the file

`file.read(N)`

Returns a string containing the next N characters (or bytes) from the file

`file.readline()`

Reads through the next `\n` and returns a line string

`file.readlines()`

Reads the entire file and returns a list of line strings

Let's run these method calls to read files, lines, and characters from a text file—the `seek(0)` call is used here before each test to rewind the file to its beginning (more on this call in a moment):

```
>>> file.seek(0)                                # go back to the front of file
>>> file.read()                                 # read entire file into string
'Hello file world!\nBye  file world.\n'

>>> file.seek(0)                                # read entire file into lines list
>>> file.readlines()
['Hello file world!\n', 'Bye  file world.\n']

>>> file.seek(0)                                # read one line at a time
>>> file.readline()
'Hello file world!\n'
>>> file.readline()
'Bye  file world.\n'
>>> file.readline()                             # empty string at end-of-file
''

>>> file.seek(0)                                # read N (or remaining) chars/bytes
>>> file.read(1), file.read(8)                  # empty string at end-of-file
('H', 'ello fil')
```

All of these input methods let us be specific about how much to fetch. Here are a few rules of thumb about which to choose:

- `read()` and `readlines()` load the *entire file* into memory all at once. That makes them handy for grabbing a file's contents with as little code as possible. It also makes them generally fast, but costly in terms of memory for huge files—loading a multigigabyte file into memory is not generally a good thing to do (and might not be possible at all on a given computer).
- On the other hand, because the `readline()` and `read(N)` calls fetch just *part of the file* (the next line or N-character-or-byte block), they are safer for potentially big files but a bit less convenient and sometimes slower. Both return an empty string when they reach end-of-file. If speed matters and your files aren't huge, `read` or `readlines` may be a generally better choice.
- See also the discussion of the newer file iterators in the next section. As we'll see, iterators combine the convenience of `readlines()` with the space efficiency of `readline()` and are the preferred way to read text files by lines today.

The `seek(0)` call used repeatedly here means “go back to the start of the file.” In our example, it is an alternative to reopening the file each time. In files, all read and write operations take place at the current position; files normally start at offset 0 when opened and advance as data is transferred. The `seek` call simply lets us move to a new position for the next transfer operation. More on this method later when we explore random access files.

Reading lines with file iterators

In older versions of Python, the traditional way to read a file line by line in a `for` loop was to read the file into a list that could be stepped through as usual:

```
>>> file = open('data.txt')
>>> for line in file.readlines():
...     print(line, end='')
```

If you've already studied the core language using a first book like *Learning Python*, you may already know that this coding pattern is actually more work than is needed today—both for you and your computer's memory. In recent Pythons, the file object includes an *iterator* which is smart enough to grab just one line per request in all iteration contexts, including `for` loops and list comprehensions. The practical benefit of this extension is that you no longer need to call `readlines` in a `for` loop to scan line by line—the iterator reads lines on request automatically:

```
>>> file = open('data.txt')
>>> for line in file:
...     print(line, end='')          # no need to call readlines
...                               # iterator reads next line each time
...
Hello file world!
Bye   file world.
```

Better still, you can open the file in the loop statement itself, as a temporary which will be automatically closed on garbage collection when the loop ends (that's normally the file's sole reference):

```
>>> for line in open('data.txt'):
...     print(line, end='')          # even shorter: temporary file object
...                               # auto-closed when garbage collected
...
Hello file world!
Bye   file world.
```

Moreover, this file line-iterator form does not load the entire file into a line's list all at once, so it will be more space efficient for large text files. Because of that, this is the prescribed way to read line by line today. If you want to see what really happens inside the `for` loop, you can use the iterator manually; it's just a `_next_` method (run by the `next` built-in function), which is similar to calling the `readline` method each time through, except that read methods return an empty string at end-of-file (EOF) and the iterator raises an exception to end the iteration:

```
>>> file = open('data.txt')      # read methods: empty at EOF
>>> file.readline()
'Hello file world!\n'
>>> file.readline()
'Bye   file world.\n'
>>> file.readline()
'

>>> file = open('data.txt')      # iterators: exception at EOF
>>> file.__next__()
# no need to call iter(file) first,
# since files are their own iterator
'Hello file world!\n'
```

```
>>> file.__next__()  
'Bye    file world.\n'  
>>> file.__next__()  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
StopIteration
```

Interestingly, iterators are automatically used in all iteration contexts, including the `list` constructor call, list comprehension expressions, `map` calls, and `in` membership checks:

```
>>> open('data.txt').readlines()                                # always read lines  
['Hello file world!\n', 'Bye    file world.\n']  
  
>>> list(open('data.txt'))                                    # force line iteration  
['Hello file world!\n', 'Bye    file world.\n']  
  
>>> lines = [line.rstrip() for line in open('data.txt')]      # comprehension  
>>> lines  
['Hello file world!', 'Bye    file world.']  
  
>>> lines = [line.upper() for line in open('data.txt')]      # arbitrary actions  
>>> lines  
['HELLO FILE WORLD!\n', 'BYE    FILE WORLD.\n']  
  
>>> list(map(str.split, open('data.txt')))                  # apply a function  
[['Hello', 'file', 'world!'], ['Bye', 'file', 'world.']]  
  
>>> line = 'Hello file world!\n'  
>>> line in open('data.txt')                                 # line membership  
True
```

Iterators may seem somewhat implicit at first glance, but they're representative of the many ways that Python makes developers' lives easier over time.

Other open options

Besides the `w` and (default) `r` file open modes, most platforms support an `a` mode string, meaning "append." In this output mode, `write` methods add data to the end of the file, and the `open` call will not erase the current contents of the file:

```
>>> file = open('data.txt', 'a')                            # open in append mode: doesn't erase  
>>> file.write('The Life of Brian')                      # added at end of existing data  
>>> file.close()  
>>>  
>>> open('data.txt').read()                             # open and read entire file  
'Hello file world!\nBye    file world.\nThe Life of Brian'
```

In fact, although most files are opened using the sorts of calls we just ran, `open` actually supports additional arguments for more specific processing needs, the first three of which are the most commonly used—the filename, the open mode, and a buffering specification. All but the first of these are optional: if omitted, the open mode argument

defaults to `r` (input), and the buffering policy is to enable full buffering. For special needs, here are a few things you should know about these three `open` arguments:

Filename

As mentioned earlier, filenames can include an explicit directory path to refer to files in arbitrary places on your computer; if they do not, they are taken to be names relative to the current working directory (described in the prior chapter). In general, most filename forms you can type in your system shell will work in an `open` call. For instance, a relative filename argument `r'..\temp\spam.txt'` on Windows means `spam.txt` in the `temp` subdirectory of the current working directory's parent—up one, and down to directory `temp`.

Open mode

The `open` function accepts other modes, too, some of which we'll see at work later in this chapter: `r+`, `w+`, and `a+` to open for reads *and* writes, and any mode string with a `b` to designate binary mode. For instance, mode `r+` means both reads and writes are allowed on an existing file; `w+` allows reads and writes but creates the file anew, erasing any prior content; `rb` and `wb` read and write data in binary mode without any translations; and `wb+` and `r+b` both combine binary mode and input plus output. In general, the mode string defaults to `r` for read but can be `w` for write and `a` for append, and you may add a `+` for update, as well as a `b` or `t` for binary or text mode; order is largely irrelevant.

As we'll see later in this chapter, the `+` modes are often used in conjunction with the file object's `seek` method to achieve random read/write access. Regardless of mode, file contents are always strings in Python programs—read methods return a string, and we pass a string to write methods. As also described later, though, the mode string implies which type of string is used: `str` for text mode or `bytes` and other byte string types for binary mode.

Buffering policy

The `open` call also takes an optional third buffering policy argument which lets you control buffering for the file—the way that data is queued up before being transferred, to boost performance. If passed, 0 means file operations are unbuffered (data is transferred immediately, but allowed in binary modes only), 1 means they are line buffered, and any other positive value means to use a full buffering (which is the default, if no buffering argument is passed).

As usual, Python's library manual and reference texts have the full story on additional `open` arguments beyond these three. For instance, the `open` call supports additional arguments related to the *end-of-line* mapping behavior and the automatic Unicode *encoding* of content performed for text-mode files. Since we'll discuss both of these concepts in the next section, let's move ahead.

Binary and Text Files

All of the preceding examples process simple text files, but Python scripts can also open and process files containing *binary* data—JPEG images, audio clips, packed binary data produced by FORTRAN and C programs, encoded text, and anything else that can be stored in files as bytes. The primary difference in terms of your code is the *mode* argument passed to the built-in `open` function:

```
>>> file = open('data.txt', 'wb')      # open binary output file
>>> file = open('data.txt', 'rb')      # open binary input file
```

Once you've opened binary files in this way, you may read and write their contents using the same methods just illustrated: `read`, `write`, and so on. The `readline` and `readlines` methods as well as the file's line iterator still work here for text files opened in binary mode, but they don't make sense for truly binary data that isn't line oriented (end-of-line bytes are meaningless, if they appear at all).

In all cases, data transferred between files and your programs is represented as Python strings within scripts, even if it is binary data. For binary mode files, though, file content is represented as *byte strings*. Continuing with our text file from preceding examples:

```
>>> open('data.txt').read()          # text mode: str
'Hello file world!\nBye   file world.\nThe Life of Brian'

>>> open('data.txt', 'rb').read()     # binary mode: bytes
b'Hello file world!\r\nBye   file world.\r\n\r\nThe Life of Brian'

>>> file = open('data.txt', 'rb')
>>> for line in file: print(line)
...
b'Hello file world!\r\n'
b'Bye   file world.\r\n'
b'The Life of Brian'
```

This occurs because Python 3.X treats text-mode files as Unicode, and automatically decodes content on input and encodes it on output. Binary mode files instead give us access to file content as raw byte strings, with no translation of content—they reflect exactly what is stored on the file. Because `str` strings are always Unicode text in 3.X, the special `bytes` string is required to represent binary data as a sequence of byte-size integers which may contain any 8-bit value. Because normal and byte strings have almost identical operation sets, many programs can largely take this on faith; but keep in mind that you really *must* open truly binary data in binary mode for input, because it will not generally be decodable as Unicode text.

Similarly, you must also supply byte strings for binary mode output—normal strings are not raw binary data, but are decoded Unicode characters (a.k.a. code points) which are encoded to binary on text-mode output:

```
>>> open('data.bin', 'wb').write(b'Spam\n')
5
>>> open('data.bin', 'rb').read()
```

```
b'Spam\n'  
>>> open('data.bin', 'wb').write('spam\n')  
TypeError: must be bytes or buffer, not str
```

But notice that this file's line ends with just \n, instead of the Windows \r\n that showed up in the preceding example for the text file in binary mode. Strictly speaking, binary mode disables Unicode encoding translation, but it also prevents the automatic end-of-line character translation performed by text-mode files by default. Before we can understand this fully, though, we need to study the two main ways in which text files differ from binary.

Unicode encodings for text files

As mentioned earlier, text-mode file objects always translate data according to a default or provided Unicode encoding type, when the data is transferred to and from external file. Their content is encoded on files, but decoded in memory. Binary mode files don't perform any such translation, which is what we want for truly binary data. For instance, consider the following string, which embeds a Unicode character whose binary value is outside the normal 7-bit range of the ASCII encoding standard:

```
>>> data = 'sp\xe4m'  
>>> data  
'späm'  
>>> 0xe4, bin(0xe4), chr(0xe4)  
(228, '0b11100100', 'ä')
```

It's possible to manually encode this string according to a variety of Unicode encoding types—its raw binary byte string form is different under some encodings:

```
>>> data.encode('latin1')                      # 8-bit characters: ascii + extras  
b'sp\xe4m'  
  
>>> data.encode('utf8')                      # 2 bytes for special characters only  
b'sp\xc3\xa4m'  
  
>>> data.encode('ascii')                      # does not encode per ascii  
UnicodeEncodeError: 'ascii' codec can't encode character '\xe4' in position 2:  
ordinal not in range(128)
```

Python displays printable characters in these strings normally, but nonprintable bytes show as \xNN hexadecimal escapes which become more prevalent under more sophisticated encoding schemes (cp500 in the following is an EBCDIC encoding):

```
>>> data.encode('utf16')                      # 2 bytes per character plus preamble  
b'\xff\xfe\x00\x00\x00\xe4\x00\x00'  
  
>>> data.encode('cp500')                      # an ebcDIC encoding: very different  
b'\xa2\x97\x94'
```

The encoded results here reflect the string's raw binary form when stored in files. Manual encoding is usually unnecessary, though, because text files handle encodings automatically on data transfers—reads decode and writes encode, according

to the encoding name passed in (or a default for the underlying platform: see `sys.getdefaultencoding`). Continuing our interactive session:

```
>>> open('data.txt', 'w', encoding='latin1').write(data)
4
>>> open('data.txt', 'r', encoding='latin1').read()
'späm'
>>> open('data.txt', 'rb').read()
b'sp\xe4m'
```

If we open in binary mode, though, no encoding translation occurs—the last command in the preceding example shows us what’s actually stored on the file. To see how file content differs for other encodings, let’s save the same string again:

```
>>> open('data.txt', 'w', encoding='utf8').write(data)      # encode data per utf8
4
>>> open('data.txt', 'r', encoding='utf8').read()          # decode: undo encoding
'späm'
>>> open('data.txt', 'rb').read()                         # no data translations
b'sp\xc3\xa4m'
```

This time, raw file content is different, but text mode’s auto-decoding makes the string the same by the time it’s read back by our script. Really, encodings pertain only to strings while they are in files; once they are loaded into memory, strings are simply sequences of Unicode characters (“code points”). This translation step is what we want for text files, but not for binary. Because binary modes skip the translation, you’ll want to use them for truly binary data. If fact, you usually must—trying to write unencodable data and attempting to read undecodable data is an error:

```
>>> open('data.txt', 'w', encoding='ascii').write(data)
UnicodeEncodeError: 'ascii' codec can't encode character '\xe4' in position 2:
ordinal not in range(128)

>>> open(r'C:\Python31\python.exe', 'r').read()
UnicodeDecodeError: 'charmap' codec can't decode byte 0x90 in position 2:
character maps to <undefined>
```

Binary mode is also a last resort for reading text files, if they cannot be decoded per the underlying platform’s default, and the encoding type is unknown—the following re-creates the original strings if encoding type is known, but fails if it is not known unless binary mode is used (such failure may occur either on inputting the data or printing it, but it fails nevertheless):

```
>>> open('data.txt', 'w', encoding='cp500').writelines(['spam\n', 'ham\n'])
>>> open('data.txt', 'r', encoding='cp500').readlines()
['spam\n', 'ham\n']

>>> open('data.txt', 'r').readlines()
UnicodeDecodeError: 'charmap' codec can't decode byte 0x81 in position 2:
character maps to <undefined>

>>> open('data.txt', 'rb').readlines()
[b'\xa2\x97\x81\x94\r%\x88\x81\x94\r%']
```

```
>>> open('data.txt', 'rb').read()
b'\xa2\x97\x81\x94\r%\x88\x81\x94\r%'
```

If all your text is ASCII you generally can ignore encoding altogether; data in files maps directly to characters in strings, because ASCII is a subset of most platforms' default encodings. If you must process files created with other encodings, and possibly on different platforms (obtained from the Web, for instance), binary mode may be required if encoding type is unknown. Keep in mind, however, that text in still-encoded binary form might not work as you expect: because it is encoded per a given encoding scheme, it might not accurately compare or combine with text encoded in other schemes.

Again, see other resources for more on the Unicode story. We'll revisit the Unicode story at various points in this book, especially in [Chapter 9](#), to see how it relates to the `tkinter Text` widget, and in [Part IV](#), covering Internet programming, to learn what it means for data shipped over networks by protocols such as FTP, email, and the Web at large. Text files have another feature, though, which is similarly a nonfeature for binary data: line-end translations, the topic of the next section.

End-of-line translations for text files

For historical reasons, the end of a line of text in a file is represented by different characters on different platforms. It's a single `\n` character on Unix-like platforms, but the two-character sequence `\r\n` on Windows. That's why files moved between Linux and Windows may look odd in your text editor after transfer—they may still be stored using the original platform's end-of-line convention.

For example, most Windows editors handle text in Unix format, but Notepad has been a notable exception—text files copied from Unix or Linux may look like one long line when viewed in Notepad, with strange characters inside (`\n`). Similarly, transferring a file from Windows to Unix in binary mode retains the `\r` characters (which often appear as `^M` in text editors).

Python scripts that process text files don't normally have to care, because the `files` object automatically maps the DOS `\r\n` sequence to a single `\n`. It works like this by default—when scripts are run on Windows:

- For files opened in text mode, `\r\n` is translated to `\n` when input.
- For files opened in text mode, `\n` is translated to `\r\n` when output.
- For files opened in binary mode, no translation occurs on input or output.

On Unix-like platforms, no translations occur, because `\n` is used in files. You should keep in mind two important consequences of these rules. First, the end-of-line character for text-mode files is almost always represented as a single `\n` within Python scripts, regardless of how it is stored in external files on the underlying platform. By mapping to and from `\n` on input and output, Python hides the platform-specific difference.

The second consequence of the mapping is subtler: when processing binary files, binary open modes (e.g., `rb`, `wb`) effectively turn off line-end translations. If they did not, the

translations listed previously could very well corrupt data as it is input or output—a random `\r` in data might be dropped on input, or added for a `\n` in the data on output. The net effect is that your binary data would be trashed when read and written—probably not quite what you want for your audio files and images!

This issue has become almost secondary in Python 3.X, because we generally cannot use binary data with text-mode files anyhow—because text-mode files automatically apply Unicode encodings to content, transfers will generally fail when the data cannot be decoded on input or encoded on output. Using binary mode avoids Unicode errors, and automatically disables line-end translations as well (Unicode error can be caught in `try` statements as well). Still, the fact that binary mode prevents end-of-line translations to protect file content is best noted as a separate feature, especially if you work in an ASCII-only world where Unicode encoding issues are irrelevant.

Here's the end-of-line translation at work in Python 3.1 on Windows—text mode translates to and from the platform-specific line-end sequence so our scripts are portable:

```
>>> open('temp.txt', 'w').write('shrubbery\n')      # text output mode: \n -> \r\n
10
>>> open('temp.txt', 'rb').read()                  # binary input: actual file bytes
b'shrubbery\r\n'
>>> open('temp.txt', 'r').read()                  # test input mode: \r\n -> \n
'shrubbery\n'
```

By contrast, writing data in binary mode prevents all translations as expected, even if the data happens to contain bytes that are part of line-ends in text mode (byte strings print their characters as ASCII if printable, else as hexadecimal escapes):

```
>>> data = b'a\x0b\xrc\xr\nd'                      # 4 escape code bytes, 4 normal
>>> len(data)
8
>>> open('temp.bin', 'wb').write(data)            # write binary data to file as is
8
>>> open('temp.bin', 'rb').read()                  # read as binary: no translation
b'a\x00b\xrc\xr\nd'
```

But reading binary data in text mode, whether accidental or not, can corrupt the data when transferred because of line-end translations (assuming it passes as decodable at all; ASCII bytes like these do on this Windows platform):

```
>>> open('temp.bin', 'r').read()                  # text mode read: botches \r !
'a\x00b\xnc\xnd'
```

Similarly, writing binary data in text mode can have as the same effect—line-end bytes may be changed or inserted (again, assuming the data is encodable per the platform's default):

```
>>> open('temp.bin', 'w').write(data)            # must pass str for text mode
TypeError: must be str, not bytes
>>> data.decode()
'a\x00b\xrc\xr\nd'
```

```
>>> open('temp.bin', 'w').write(data.decode())
8
>>> open('temp.bin', 'rb').read()           # text mode write: added \r !
b'a\x00b\rc\r\r\n'
>>> open('temp.bin', 'r').read()          # again drops, alters \r on input
'a\x00b\nc\n\n'
```

The short story to remember here is that you should generally use `\n` to refer to end-line in all your text file content, and you should always open binary data in binary file modes to suppress both end-of-line translations and any Unicode encodings. A file’s content generally determines its open mode, and file open modes usually process file content exactly as we want.

Keep in mind, though, that you might also need to use binary file modes for text in special contexts. For instance, in [Chapter 6](#)’s examples, we’ll sometimes open text files in binary mode to avoid possible Unicode decoding errors, for files generated on arbitrary platforms that may have been encoded in arbitrary ways. Doing so avoids encoding errors, but also can mean that some text might not work as expected—searches might not always be accurate when applied to such raw text, since the search key must be in bytes string formatted and encoded according to a specific and possibly incompatible encoding scheme.

In [Chapter 11](#)’s PyEdit, we’ll also need to catch Unicode exceptions in a “grep” directory file search utility, and we’ll go further to allow Unicode encodings to be specified for file content across entire trees. Moreover, a script that attempts to translate between different platforms’ end-of-line character conventions explicitly may need to read text in binary mode to retain the original line-end representation truly present in the file; in text mode, they would already be translated to `\n` by the time they reached the script.

It’s also possible to disable or further tailor end-of-line translations in text mode with additional `open` arguments we will finesse here. See the `newline` argument in `open` reference documentation for details; in short, passing an empty string to this argument also prevents line-end translation but retains other text-mode behavior. For this chapter, let’s turn next to two common use cases for binary data files: packed binary data and random access.

Parsing packed binary data with the `struct` module

By using the letter `b` in the `open` call, you can open binary datafiles in a platform-neutral way and read and write their content with normal file object methods. But how do you process binary data once it has been read? It will be returned to your script as a simple string of bytes, most of which are probably not printable characters.

If you just need to pass binary data along to another file or program, your work is done—for instance, simply pass the byte string to another file opened in binary mode. And if you just need to extract a number of bytes from a specific position, string slicing will do the job; you can even follow up with bitwise operations if you need to. To get

at the contents of binary data in a structured way, though, as well as to construct its contents, the standard library `struct` module is a more powerful alternative.

The `struct` module provides calls to pack and unpack binary data, as though the data was laid out in a C-language `struct` declaration. It is also capable of composing and decomposing using any endian-ness you desire (endian-ness determines whether the most significant bits of binary numbers are on the left or right side). Building a binary datafile, for instance, is straightforward—pack Python values into a byte string and write them to a file. The format string here in the `pack` call means big-endian (`>`), with an integer, four-character string, half integer, and floating-point number:

```
>>> import struct
>>> data = struct.pack('>i4shf', 2, 'spam', 3, 1.234)
>>> data
b'\x00\x00\x00\x02spam\x00\x03?\x9d\xf3\xb6'
>>> file = open('data.bin', 'wb')
>>> file.write(data)
14
>>> file.close()
```

Notice how the `struct` module returns a bytes string: we're in the realm of binary data here, not text, and must use binary mode files to store. As usual, Python displays most of the packed binary data's bytes here with `\xNN` hexadecimal escape sequences, because the bytes are not printable characters. To parse data like that which we just produced, read it off the file and pass it to the `struct` module with the same format string—you get back a tuple containing the values parsed out of the string and converted to Python objects:

```
>>> import struct
>>> file  = open('data.bin', 'rb')
>>> bytes = file.read()
>>> values = struct.unpack('>i4shf', data)
>>> values
(2, b'spam', 3, 1.2339999675750732)
```

Parsed-out strings are byte strings again, and we can apply string and bitwise operations to probe deeper:

```
>>> bin(values[0] | 0b1)                                     # accessing bits and bytes
'0b11'
>>> values[1], list(values[1]), values[1][0]
(b'spam', [115, 112, 97, 109], 115)
```

Also note that slicing comes in handy in this domain; to grab just the four-character string in the middle of the packed binary data we just read, we can simply slice it out. Numeric values could similarly be sliced out and then passed to `struct.unpack` for conversion:

```
>>> bytes
b'\x00\x00\x00\x02spam\x00\x03?\x9d\xf3\xb6'
>>> bytes[4:8]
b'spam'
```

```
>>> number = bytes[8:10]
>>> number
b'\x00\x03'
>>> struct.unpack('>h', number)
(3,)
```

Packed binary data crops up in many contexts, including some networking tasks, and in data produced by other programming languages. Because it's not part of every programming job's description, though, we'll defer to the `struct` module's entry in the Python library manual for more details.

Random access files

Binary files also typically see action in random access processing. Earlier, we mentioned that adding a `+` to the `open` mode string allows a file to be both read and written. This mode is typically used in conjunction with the file object's `seek` method to support random read/write access. Such flexible file processing modes allow us to read bytes from one location, write to another, and so on. When scripts combine this with binary file modes, they may fetch and update arbitrary bytes within a file.

We used `seek` earlier to rewind files instead of closing and reopening. As mentioned, read and write operations always take place at the current position in the file; files normally start at offset 0 when opened and advance as data is transferred. The `seek` call lets us move to a new position for the next transfer operation by passing in a byte offset.

Python's `seek` method also accepts an optional second argument that has one of three values—0 for absolute file positioning (the default); 1 to seek relative to the current position; and 2 to seek relative to the file's end. That's why passing just an offset of 0 to `seek` is roughly a file *rewind* operation: it repositions the file to its absolute start. In general, `seek` supports random access on a byte-offset basis. Seeking to a multiple of a record's size in a binary file, for instance, allows us to fetch a record by its relative position.

Although you can use `seek` without `+` modes in `open` (e.g., to just read from random locations), it's most flexible when combined with input/output files. And while you can perform random access in *text mode*, too, the fact that text modes perform Unicode encodings and line-end translations make them difficult to use when absolute byte offsets and lengths are required for seeks and reads—your data may look very different when stored in files. Text mode may also make your data nonportable to platforms with different default encodings, unless you're willing to always specify an explicit encoding for `open`s. Except for simple unencoded ASCII text without line-ends, `seek` tends to work best with binary mode files.

To demonstrate, let's create a file in `w+b` mode (equivalent to `wb+`) and write some data to it; this mode allows us to both read and write, but initializes the file to be empty if it's already present (all `w` modes do). After writing some data, we seek back to file start to read its content (some integer return values are omitted in this example again for brevity):

Now, let's reopen our file in `r+b` mode; this mode allows both reads and writes again, but does not initialize the file to be empty. This time, we seek and read in multiples of the size of data items (“records”) stored, to both fetch and update them at random:

```
c:\temp> python
>>> file = open('random.bin', 'r+b')
>>> print(file.read())
b'sssssssssppppppaaaaaaammmmmmmm'
# read entire file

>>> record = b'X' * 8
>>> file.seek(0)
>>> file.write(record)
# update first record

>>> file.seek(len(record) * 2)
# update third record

>>> file.write(b'Y' * 8)

>>> file.seek(8)
>>> file.read(len(record))
b'pppppppp'
# fetch second record

>>> file.read(len(record))
b'YYYYYYYY'
# fetch next (third) record

>>> file.seek(0)
>>> file.read()
b'XXXXXXXXppppppppYYYYYYYYYmmmmmmmm'
# read entire file

c:\temp> type random.bin
XXXXXXXXppppppppYYYYYYYYYmmmmmmmm
# the view outside Python
```

Finally, keep in mind that `seek` can be used to achieve random access, even if it's just for input. The following seeks in multiples of record size to read (but not write) fixed-length records at random. Notice that it also uses `r` text mode: since this data is simple ASCII text bytes and has no line-ends, text and binary modes work the same on this platform:

```
c:\temp> python
>>> file = open('random.bin', 'r')          # text mode ok if no encoding/endlines
>>> reclen = 8
>>> file.seek(reclen * 3)                  # fetch record 4
>>> file.read(reclen)
'aaaaaaaa'
>>> file.seek(reclen * 1)                  # fetch record 2
>>> file.read(reclen)
```

```
'pppppppp'

>>> file = open('random.bin', 'rb')      # binary mode works the same here
>>> file.seek(reclen * 2)                # fetch record 3
>>> file.read(reclen)                  # returns byte strings
b'YYYYYYYY'
```

But unless your file's content is always a simple unencoded text form like ASCII and has no translated line-ends, text mode should not generally be used if you are going to seek—line-ends may be translated on Windows and Unicode encodings may make arbitrary transformations, both of which can make absolute seek offsets difficult to use. In the following, for example, the positions of characters after the first non-ASCII no longer match between the string in Python and its encoded representation on the file:

```
>>> data = 'sp\xe4m'                      # data to your script
>>> data, len(data)                       # 4 unicode chars, 1 nonascii
('späm', 4)
>>> data.encode('utf8'), len(data.encode('utf8'))    # bytes written to file
(b'sp\xc3\xaa\xd4m', 5)

>>> f = open('test', mode='w+', encoding='utf8')    # use text mode, encoded
>>> f.write(data)
>>> f.flush()
>>> f.seek(0); f.read(1)                   # ascii bytes work
's'
>>> f.seek(2); f.read(1)                   # as does 2-byte nonascii
'ä'
>>> data[3]                                # but offset 3 is not 'm' !
'm'
>>> f.seek(3); f.read(1)
UnicodeDecodeError: 'utf8' codec can't decode byte 0xa4 in position 0:
unexpected code byte
```

As you can see, Python's file modes provide flexible file processing for programs that require it. In fact, the `os` module offers even more file processing options, as the next section describes.

Lower-Level File Tools in the `os` Module

The `os` module contains an additional set of file-processing functions that are distinct from the built-in file *object* tools demonstrated in previous examples. For instance, here is a partial list of `os` file-related calls:

```
os.open( path, flags, mode )
    Opens a file and returns its descriptor
os.read( descriptor, N )
    Reads at most N bytes and returns a byte string
os.write( descriptor, string )
    Writes bytes in byte string string to the file
```

```
os.lseek( descriptor, position , how )
```

Moves to *position* in the file

Technically, `os` calls process files by their *descriptors*, which are integer codes or “handles” that identify files in the operating system. Descriptor-based files deal in raw bytes, and have no notion of the line-end or Unicode translations for text that we studied in the prior section. In fact, apart from extras like buffering, descriptor-based files generally correspond to binary mode file objects, and we similarly read and write `bytes` strings, not `str` strings. However, because the descriptor-based file tools in `os` are lower level and more complex than the built-in file objects created with the built-in `open` function, you should generally use the latter for all but very special file-processing needs.*

Using `os.open` files

To give you the general flavor of this tool set, though, let’s run a few interactive experiments. Although built-in file objects and `os` module descriptor files are processed with distinct tool sets, they are in fact related—the file system used by file objects simply adds a layer of logic on top of descriptor-based files.

In fact, the `fileno` file object method returns the integer descriptor associated with a built-in file object. For instance, the standard stream file objects have descriptors 0, 1, and 2; calling the `os.write` function to send data to `stdout` by descriptor has the same effect as calling the `sys.stdout.write` method:

```
>>> import sys
>>> for stream in (sys.stdin, sys.stdout, sys.stderr):
...     print(stream.fileno())
...
0
1
2

>>> sys.stdout.write('Hello stdio world\n')      # write via file method
Hello stdio world
18
>>> import os
>>> os.write(1, b'Hello descriptor world\n')      # write via os module
Hello descriptor world
23
```

Because file objects we open explicitly behave the same way, it’s also possible to process a given real external file on the underlying computer through the built-in `open` function, tools in the `os` module, or both (some integer return values are omitted here for brevity):

* For instance, to process *pipes*, described in [Chapter 5](#). The Python `os.pipe` call returns two file descriptors, which can be processed with `os` module file tools or wrapped in a file object with `os.fdopen`. When used with descriptor-based file tools in `os`, pipes deal in byte strings, not text. Some device files may require lower-level control as well.

```

>>> file = open(r'C:\temp\spam.txt', 'w')           # create external file, object
>>> file.write('Hello stdio file\n')                 # write via file object method
>>> file.flush()                                    # else os.write to disk first!
>>> fd = file.fileno()                            # get descriptor from object
>>> fd
3
>>> import os
>>> os.write(fd, b'Hello descriptor file\n')      # write via os module
>>> file.close()

C:\temp> type spam.txt                         # lines from both schemes
Hello stdio file
Hello descriptor file

```

os.open mode flags

So why the extra file tools in `os`? In short, they give more low-level control over file processing. The built-in `open` function is easy to use, but it may be limited by the underlying filesystem that it uses, and it adds extra behavior that we do not want. The `os` module lets scripts be more specific—for example, the following opens a descriptor-based file in read-write and binary modes by performing a binary “or” on two mode flags exported by `os`:

```

>>> fdfile = os.open(r'C:\temp\spam.txt', (os.O_RDWR | os.O_BINARY))
>>> os.read(fdfile, 20)
b'Hello stdio file\r\nHe'

>>> os.lseek(fdfile, 0, 0)                      # go back to start of file
>>> os.read(fdfile, 100)                         # binary mode retains "\r\n"
b'Hello stdio file\r\nHello descriptor file\n'

>>> os.lseek(fdfile, 0, 0)
>>> os.write(fdfile, b'HELLO')                  # overwrite first 5 bytes
5

C:\temp> type spam.txt
HELLO stdio file
Hello descriptor file

```

In this case, binary mode strings `rb+` and `r+b` in the basic `open` call are equivalent:

```

>>> file = open(r'C:\temp\spam.txt', 'rb+')        # same but with open/objects
>>> file.read(20)
b'HELLO stdio file\r\nHe'
>>> file.seek(0)
>>> file.read(100)
b'HELLO stdio file\r\nHello descriptor file\n'
>>> file.seek(0)
>>> file.write(b'Jello')
5
>>> file.seek(0)
>>> file.read()
b'Jello stdio file\r\nHello descriptor file\n'

```

But on some systems, `os.open` flags let us specify more advanced things like *exclusive access* (`O_EXCL`) and *nonblocking* modes (`O_NONBLOCK`) when a file is opened. Some of these flags are not portable across platforms (another reason to use built-in file objects most of the time); see the library manual or run a `dir(os)` call on your machine for an exhaustive list of other open flags available.

One final note here: using `os.open` with the `O_EXCL` flag is the most portable way to *lock files* for concurrent updates or other process synchronization in Python today. We'll see contexts where this can matter in the next chapter, when we begin to explore multiprocessing tools. Programs running in parallel on a server machine, for instance, may need to lock files before performing updates, if multiple threads or processes might attempt such updates at the same time.

Wrapping descriptors in file objects

We saw earlier how to go from file object to field descriptor with the `fileno` file object method; given a descriptor, we can use `os` module tools for lower-level file access to the underlying file. We can also go the other way—the `os.fdopen` call wraps a file descriptor in a file object. Because conversions work both ways, we can generally use either tool set—file object or `os` module:

```
>>> fdfile = os.open(r'C:\temp\spam.txt', (os.O_RDWR | os.O_BINARY))
>>> fdfile
3
>>> objfile = os.fdopen(fdfile, 'rb')
>>> objfile.read()
b'Jello stdio file\r\nHello descriptor file\n'
```

In fact, we can wrap a file descriptor in either a binary or text-mode file object: in text mode, reads and writes perform the Unicode encodings and line-end translations we studied earlier and deal in `str` strings instead of `bytes`:

```
C:\...\PP4E\System> python
>>> import os
>>> fdfile = os.open(r'C:\temp\spam.txt', (os.O_RDWR | os.O_BINARY))
>>> objfile = os.fdopen(fdfile, 'r')
>>> objfile.read()
'Jello stdio file\nHello descriptor file\n'
```

In Python 3.X, the built-in `open` call also accepts a file descriptor instead of a file name string; in this mode it works much like `os.fdopen`, but gives you greater control—for example, you can use additional arguments to specify a nondefault Unicode encoding for text and suppress the default descriptor close. Really, though, `os.fdopen` accepts the same extra-control arguments in 3.X, because it has been redefined to do little but call back to the built-in `open` (see `os.py` in the standard library):

```
C:\...\PP4E\System> python
>>> import os
>>> fdfile = os.open(r'C:\temp\spam.txt', (os.O_RDWR | os.O_BINARY))
>>> fdfile
3
```

```

>>> objfile = open(fdfile, 'r', encoding='latin1', closefd=False)
>>> objfile.read()
'Jello stdio file\nHello descriptor file\n'

>>> objfile = os.fdopen(fdfile, 'r', encoding='latin1', closefd=True)
>>> objfile.seek(0)
>>> objfile.read()
'Jello stdio file\nHello descriptor file\n'

```

We'll make use of this file object wrapper technique to simplify text-oriented pipes and other descriptor-like objects later in this book (e.g., sockets have a `makefile` method which achieves similar effects).

Other `os` module file tools

The `os` module also includes an assortment of file tools that accept a file pathname string and accomplish file-related tasks such as renaming (`os.rename`), deleting (`os.remove`), and changing the file's owner and permission settings (`os.chown`, `os.chmod`). Let's step through a few examples of these tools in action:

```
>>> os.chmod('spam.txt', 0o777)      # enable all accesses
```

This `os.chmod` file permissions call passes a 9-bit string composed of three sets of three bits each. From left to right, the three sets represent the file's owning user, the file's group, and all others. Within each set, the three bits reflect read, write, and execute access permissions. When a bit is "1" in this string, it means that the corresponding operation is allowed for the assessor. For instance, octal 0777 is a string of nine "1" bits in binary, so it enables all three kinds of accesses for all three user groups; octal 0600 means that the file can be read and written only by the user that owns it (when written in binary, 0600 octal is really bits 110 000 000).

This scheme stems from Unix file permission settings, but the call works on Windows as well. If it's puzzling, see your system's documentation (e.g., a Unix manpage) for `chmod`. Moving on:

```

>>> os.rename(r'C:\temp\spam.txt', r'C:\temp\eggs.txt')      # from, to

>>> os.remove(r'C:\temp\spam.txt')                          # delete file?
WindowsError: [Error 2] The system cannot find the file specified: 'C:\\temp\\...'

>>> os.remove(r'C:\temp\eggs.txt')

```

The `os.rename` call used here changes a file's name; the `os.remove` file deletion call deletes a file from your system and is synonymous with `os.unlink` (the latter reflects the call's name on Unix but was obscure to users of other platforms).[†] The `os` module also exports the `stat` system call:

[†] For related tools, see also the `shutil` module in Python's standard library; it has higher-level tools for copying and removing files and more. We'll also write directory compare, copy, and search tools of our own in [Chapter 6](#), after we've had a chance to study the directory tools presented later in this chapter.

```

>>> open('spam.txt', 'w').write('Hello stat world\n')           # +1 for \r added
17
>>> import os
>>> info = os.stat(r'C:\temp\spam.txt')
>>> info
nt.stat_result(st_mode=33206, st_ino=0, st_dev=0, st_nlink=0, st_uid=0, st_gid=0,
st_size=18, st_atime=1267645806, st_mtime=1267646072, st_ctime=1267645806)

>>> info.st_mode, info.st_size                                # via named-tuple item attr names
(33206, 18)

>>> import stat
>>> info[stat.ST_MODE], info[stat.ST_SIZE]                  # via stat module presets
(33206, 18)
>>> stat.S_ISDIR(info.st_mode), stat.S_ISREG(info.st_mode)
(False, True)

```

The `os.stat` call returns a tuple of values (really, in 3.X a special kind of tuple with named items) giving low-level information about the named file, and the `stat` module exports constants and functions for querying this information in a portable way. For instance, indexing an `os.stat` result on offset `stat.ST_SIZE` returns the file's size, and calling `stat.S_ISDIR` with the mode item from an `os.stat` result checks whether the file is a directory. As shown earlier, though, both of these operations are available in the `os.path` module, too, so it's rarely necessary to use `os.stat` except for low-level file queries:

```

>>> path = r'C:\temp\spam.txt'
>>> os.path.isdir(path), os.path.isfile(path), os.path.getsize(path)
(False, True, 18)

```

File Scanners

Before we leave our file tools survey, it's time for something that performs a more tangible task and illustrates some of what we've learned so far. Unlike some shell-tool languages, Python doesn't have an implicit file-scanning loop procedure, but it's simple to write a general one that we can reuse for all time. The module in [Example 4-1](#) defines a general file-scanning routine, which simply applies a passed-in Python function to each line in an external file.

Example 4-1. PP4E\System\Filetools\scanfile.py

```

def scanner(name, function):
    file = open(name, 'r')                                     # create a file object
    while True:
        line = file.readline()                                # call file methods
        if not line: break                                    # until end-of-file
        function(line)                                       # call a function object
    file.close()

```

The `scanner` function doesn't care what line-processing function is passed in, and that accounts for most of its generality—it is happy to apply *any* single-argument function

that exists now or in the future to all of the lines in a text file. If we code this module and put it in a directory on the module search path, we can use it any time we need to step through a file line by line. [Example 4-2](#) is a client script that does simple line translations.

Example 4-2. PP4E\System\Filetools\commands.py

```
#!/usr/local/bin/python
from sys import argv
from scanfile import scanner
class UnknownCommand(Exception): pass

def processLine(line):                      # define a function
    if line[0] == '*':                      # applied to each line
        print("Ms.", line[1:-1])
    elif line[0] == '+':
        print("Mr.", line[1:-1])           # strip first and last char: \n
    else:
        raise UnknownCommand(line)        # raise an exception

filename = 'data.txt'
if len(argv) == 2: filename = argv[1]         # allow filename cmd arg
scanner(filename, processLine)               # start the scanner
```

The text file *hillbillies.txt* contains the following lines:

```
*Granny
+Jethro
*Elly May
+"Uncle Jed"
```

and our commands script could be run as follows:

```
C:\...\PP4E\System\Filetools> python commands.py hillbillies.txt
Ms. Granny
Mr. Jethro
Ms. Elly May
Mr. "Uncle Jed"
```

This works, but there are a variety of coding alternatives for both files, some of which may be better than those listed above. For instance, we could also code the command processor of [Example 4-2](#) in the following way; especially if the number of command options starts to become large, such a data-driven approach may be more concise and easier to maintain than a large `if` statement with essentially redundant actions (if you ever have to change the way output lines print, you'll have to change it in only one place with this form):

```
commands = {'*': 'Ms.', '+': 'Mr.'}      # data is easier to expand than code?

def processLine(line):
    try:
        print(commands[line[0]], line[1:-1])
    except KeyError:
        raise UnknownCommand(line)
```

The scanner could similarly be improved. As a rule of thumb, we can also usually speed things up by shifting processing from Python code to built-in tools. For instance, if we're concerned with speed, we can probably make our file scanner faster by using the file's *line iterator* to step through the file instead of the manual `readline` loop in [Example 4-1](#) (though you'd have to time this with your Python to be sure):

```
def scanner(name, function):
    for line in open(name, 'r'):      # scan line by line
        function(line)               # call a function object
```

And we can work more magic in [Example 4-1](#) with the iteration tools like the `map` built-in function, the list comprehension expression, and the generator expression. Here are three minimalist's versions; the `for` loop is replaced by `map` or a comprehension, and we let Python close the file for us when it is garbage collected or the script exits (these all build a temporary list of results along the way to run through their iterations, but this overhead is likely trivial for all but the largest of files):

```
def scanner(name, function):
    list(map(function, open(name, 'r')))

def scanner(name, function):
    [function(line) for line in open(name, 'r')]

def scanner(name, function):
    list(function(line) for line in open(name, 'r'))
```

File filters

The preceding works as planned, but what if we also want to *change* a file while scanning it? [Example 4-3](#) shows two approaches: one uses explicit files, and the other uses the standard input/output streams to allow for redirection on the command line.

Example 4-3. PP4E\System\Filetools\filters.py

```
import sys

def filter_files(name, function):          # filter file through function
    input_ = open(name, 'r')                # create file objects
    output = open(name + '.out', 'w')       # explicit output file too
    for line in input_:
        output.write(function(line))        # write the modified line
    input_.close()                         # output has a '.out' suffix
    output.close()

def filter_stream(function):               # no explicit files
    while True:                          # use standard streams
        line = sys.stdin.readline()        # or: input()
        if not line: break
        print(function(line), end='')     # or: sys.stdout.write()

if __name__ == '__main__':
    filter_stream(lambda line: line)      # copy stdin to stdout if run
```

Notice that the newer *context managers* feature discussed earlier could save us a few lines here in the file-based filter of [Example 4-3](#), and also guarantee immediate file closures if the processing function fails with an exception:

```
def filter_files(name, function):
    with open(name, 'r') as input, open(name + '.out', 'w') as output:
        for line in input:
            output.write(function(line))      # write the modified line
```

And again, file object *line iterators* could simplify the stream-based filter's code in this example as well:

```
def filter_stream(function):
    for line in sys.stdin:                  # read by lines automatically
        print(function(line), end='')
```

Since the standard streams are preopened for us, they're often easier to use. When run standalone, it simply parrots `stdin` to `stdout`:

```
C:\...\PP4E\System\Filetools> filters.py < hillbillies.txt
*Granny
+Jethro
*Elly May
+"Uncle Jed"
```

But this module is also useful when imported as a library (clients provide the line-processing function):

```
>>> from filters import filter_files
>>> filter_files('hillbillies.txt', str.upper)
>>> print(open('hillbillies.txt.out').read())
*GRANNY
+JETHRO
*ELLY MAY
+"UNCLE JED"
```

We'll see files in action often in the remainder of this book, especially in the more complete and functional system examples of [Chapter 6](#). First though, we turn to tools for processing our files' home.

Directory Tools

One of the more common tasks in the shell utilities domain is applying an operation to a set of files in a *directory*—a “folder” in Windows-speak. By running a script on a batch of files, we can automate (that is, *script*) tasks we might have to otherwise run repeatedly by hand.

For instance, suppose you need to search all of your Python files in a development directory for a global variable name (perhaps you've forgotten where it is used). There are many platform-specific ways to do this (e.g., the `find` and `grep` commands in Unix), but Python scripts that accomplish such tasks will work on every platform where Python works—Windows, Unix, Linux, Macintosh, and just about any other platform

commonly used today. If you simply copy your script to any machine you wish to use it on, it will work regardless of which other tools are available there; all you need is Python. Moreover, coding such tasks in Python also allows you to perform arbitrary actions along the way—replacements, deletions, and whatever else you can code in the Python language.

Walking One Directory

The most common way to go about writing such tools is to first grab a list of the names of the files you wish to process, and then step through that list with a Python `for` loop or other iteration tool, processing each file in turn. The trick we need to learn here, then, is how to get such a directory list within our scripts. For scanning directories there are at least three options: running shell listing commands with `os.popen`, matching filename patterns with `glob.glob`, and getting directory listings with `os.listdir`. They vary in interface, result format, and portability.

Running shell listing commands with `os.popen`

How did you go about getting directory file listings before you heard of Python? If you’re new to shell tools programming, the answer may be “Well, I started a Windows file explorer and clicked on things,” but I’m thinking here in terms of less GUI-oriented command-line mechanisms.

On Unix, directory listings are usually obtained by typing `ls` in a shell; on Windows, they can be generated with a `dir` command typed in an MS-DOS console box. Because Python scripts may use `os.popen` to run any command line that we can type in a shell, they are the most general way to grab a directory listing inside a Python program. We met `os.popen` in the prior chapters; it runs a shell command string and gives us a file object from which we can read the command’s output. To illustrate, let’s first assume the following directory structures—I have both the usual `dir` and a Unix-like `ls` command from Cygwin on my Windows laptop:

```
c:\temp> dir /B
parts
PP3E
random.bin
spam.txt
temp.bin
temp.txt

c:\temp> c:\cygwin\bin\ls
PP3E parts random.bin spam.txt temp.bin temp.txt

c:\temp> c:\cygwin\bin\ls parts
part0001 part0002 part0003 part0004
```

The `parts` and `PP3E` names are a nested subdirectory in `C:\temp` here (the latter is a copy of the prior edition’s examples tree, which I used occasionally in this text). Now, as

we've seen, scripts can grab a listing of file and directory names at this level by simply spawning the appropriate platform-specific command line and reading its output (the text normally thrown up on the console window):

```
C:\temp> python
>>> import os
>>> os.popen('dir /B').readlines()
['parts\n', 'PP3E\n', 'random.bin\n', 'spam.txt\n', 'temp.bin\n', 'temp.txt\n']
```

Lines read from a shell command come back with a trailing end-of-line character, but it's easy enough to slice it off; the `os.popen` result also gives us a line iterator just like normal files:

```
>>> for line in os.popen('dir /B'):
...     print(line[:-1])
...
parts
PP3E
random.bin
spam.txt
temp.bin
temp.txt

>>> lines = [line[:-1] for line in os.popen('dir /B')]
>>> lines
['parts', 'PP3E', 'random.bin', 'spam.txt', 'temp.bin', 'temp.txt']
```

For pipe objects, the effect of iterators may be even more useful than simply avoiding loading the entire result into memory all at once: `readlines` will always block the caller until the spawned program is completely finished, whereas the iterator might not.

The `dir` and `ls` commands let us be specific about filename patterns to be matched and directory names to be listed by using name patterns; again, we're just running shell commands here, so anything you can type at a shell prompt goes:

```
>>> os.popen('dir *.bin /B').readlines()
['random.bin\n', 'temp.bin\n']

>>> os.popen(r'c:\cygwin\bin\ls *.bin').readlines()
['random.bin\n', 'temp.bin\n']

>>> list(os.popen(r'dir parts /B'))
['part0001\n', 'part0002\n', 'part0003\n', 'part0004\n']

>>> [fname for fname in os.popen(r'c:\cygwin\bin\ls parts')]
['part0001\n', 'part0002\n', 'part0003\n', 'part0004\n']
```

These calls use general tools and work as advertised. As I noted earlier, though, the downsides of `os.popen` are that it requires using a platform-specific shell command and it incurs a performance hit to start up an independent program. In fact, different listing tools may sometimes produce different results:

```
>>> list(os.popen(r'dir parts\part* /B'))
['part0001\n', 'part0002\n', 'part0003\n', 'part0004\n']
```

```
>>>  
>>> list(os.popen(r'c:\cygwin\bin\ls parts/part*'))  
['parts/part0001\n', 'parts/part0002\n', 'parts/part0003\n', 'parts/part0004\n']
```

The next two alternative techniques do better on both counts.

The glob module

The term *globbing* comes from the * wildcard character in filename patterns; per computing folklore, a * matches a “glob” of characters. In less poetic terms, globbing simply means collecting the names of all entries in a directory—files and subdirectories—whose names match a given filename pattern. In Unix shells, globbing expands filename patterns within a command line into all matching filenames before the command is ever run. In Python, we can do something similar by calling the `glob.glob` built-in—a tool that accepts a filename pattern to expand, and returns a list (not a generator) of matching file names:

```
>>> import glob  
>>> glob.glob('*')  
['parts', 'PP3E', 'random.bin', 'spam.txt', 'temp.bin', 'temp.txt']  
  
>>> glob.glob('*.*')  
['random.bin', 'temp.bin']  
  
>>> glob.glob('parts')  
['parts']  
  
>>> glob.glob('parts/*')  
['parts\\part0001', 'parts\\part0002', 'parts\\part0003', 'parts\\part0004']  
  
>>> glob.glob('parts\part*')  
['parts\\part0001', 'parts\\part0002', 'parts\\part0003', 'parts\\part0004']
```

The `glob` call accepts the usual filename pattern syntax used in shells: ? means any one character, * means any number of characters, and [] is a character selection set.[‡] The pattern should include a directory path if you wish to glob in something other than the current working directory, and the module accepts either Unix or DOS-style directory separators (/ or \). This call is implemented without spawning a shell command (it uses `os.listdir`, described in the next section) and so is likely to be faster and more portable and uniform across all Python platforms than the `os.popen` schemes shown earlier.

Technically speaking, `glob` is a bit more powerful than described so far. In fact, using it to list files in one directory is just one use of its pattern-matching skills. For instance, it can also be used to collect matching names across multiple directories, simply because each level in a passed-in directory path can be a pattern too:

```
>>> for path in glob.glob(r'PP3E\Examples\PP3E\*\s*.py'): print(path)  
...
```

[‡] In fact, `glob` just uses the standard `fnmatch` module to match name patterns; see the `fnmatch` description in Chapter 6’s `find` module example for more details.

```
PP3E\Examples\PP3E\Lang\summer-alt.py  
PP3E\Examples\PP3E\Lang\summer.py  
PP3E\Examples\PP3E\PyTools\search_all.py
```

Here, we get back filenames from two different directories that match the `s*.py` pattern; because the directory name preceding this is a `*` wildcard, Python collects all possible ways to reach the base filenames. Using `os.popen` to spawn shell commands achieves the same effect, but only if the underlying shell or listing command does, too, and with possibly different result formats across tools and platforms.

The `os.listdir` call

The `os` module's `listdir` call provides yet another way to collect filenames in a Python list. It takes a simple directory name string, not a filename pattern, and returns a list containing the names of all entries in that directory—both simple files and nested directories—for use in the calling script:

```
>>> import os  
>>> os.listdir('.')
```

['parts', 'PP3E', 'random.bin', 'spam.txt', 'temp.bin', 'temp.txt']
>>>
>>> os.listdir(os.curdir)
['parts', 'PP3E', 'random.bin', 'spam.txt', 'temp.bin', 'temp.txt']
>>>
>>> os.listdir('parts')
['part0001', 'part0002', 'part0003', 'part0004']

This, too, is done without resorting to shell commands and so is both fast and portable to all major Python platforms. The result is not in any particular order across platforms (but can be sorted with the list `sort` method or `sorted` built-in function); returns base filenames without their directory path prefixes; does not include names `“.”` or `“..”` if present; and includes names of both files and directories at the listed level.

To compare all three listing techniques, let's run them here side by side on an explicit directory. They differ in some ways but are mostly just variations on a theme for this task—`os.popen` returns end-of-lines and may sort filenames on some platforms, `glob.glob` accepts a pattern and returns filenames with directory prefixes, and `os.listdir` takes a simple directory name and returns names without directory prefixes:

```
>>> os.popen('dir /b parts').readlines()  
['part0001\n', 'part0002\n', 'part0003\n', 'part0004\n']  
  
>>> glob.glob(r'parts\*')  
['parts\\part0001', 'parts\\part0002', 'parts\\part0003', 'parts\\part0004']  
  
>>> os.listdir('parts')  
['part0001', 'part0002', 'part0003', 'part0004']
```

Of these three, `glob` and `listdir` are generally better options if you care about script portability and result uniformity, and `listdir` seems fastest in recent Python releases (but gauge its performance yourself—implementations may change over time).

Splitting and joining listing results

In the last example, I pointed out that `glob` returns names with directory paths, whereas `listdir` gives raw base filenames. For convenient processing, scripts often need to split `glob` results into base files or expand `listdir` results into full paths. Such translations are easy if we let the `os.path` module do all the work for us. For example, a script that intends to copy all files elsewhere will typically need to first split off the base filenames from `glob` results so that it can add different directory names on the front:

```
>>> dirname = r'C:\temp\parts'
>>>
>>> import glob
>>> for file in glob.glob(dirname + '*'):
...     head, tail = os.path.split(file)
...     print(head, tail, '=>', ('C:\\\\Other\\\\' + tail))
...
C:\temp\parts part0001 => C:\\\\Other\\\\part0001
C:\temp\parts part0002 => C:\\\\Other\\\\part0002
C:\temp\parts part0003 => C:\\\\Other\\\\part0003
C:\temp\parts part0004 => C:\\\\Other\\\\part0004
```

Here, the names after the `=>` represent names that files might be moved to. Conversely, a script that means to process all files in a different directory than the one it runs in will probably need to prepend `listdir` results with the target directory name before passing filenames on to other tools:

```
>>> import os
>>> for file in os.listdir(dirname):
...     print(dirname, file, '=>', os.path.join(dirname, file))
...
C:\temp\parts part0001 => C:\\\\temp\\\\parts\\\\part0001
C:\\\\temp\\\\parts part0002 => C:\\\\temp\\\\parts\\\\part0002
C:\\\\temp\\\\parts part0003 => C:\\\\temp\\\\parts\\\\part0003
C:\\\\temp\\\\parts part0004 => C:\\\\temp\\\\parts\\\\part0004
```

When you begin writing realistic directory processing tools of the sort we'll develop in [Chapter 6](#), you'll find these calls to be almost habit.

Walking Directory Trees

You may have noticed that almost all of the techniques in this section so far return the names of files in only a *single* directory (globbing with more involved patterns is the only exception). That's fine for many tasks, but what if you want to apply an operation to every file in every directory and subdirectory in an entire directory *tree*?

For instance, suppose again that we need to find every occurrence of a global name in our Python scripts. This time, though, our scripts are arranged into a module *package*: a directory with nested subdirectories, which may have subdirectories of their own. We could rerun our hypothetical single-directory searcher manually in every directory in the tree, but that's tedious, error prone, and just plain not fun.

Luckily, in Python it's almost as easy to process a directory tree as it is to inspect a single directory. We can either write a recursive routine to traverse the tree, or use a tree-walker utility built into the `os` module. Such tools can be used to search, copy, compare, and otherwise process arbitrary directory trees on any platform that Python runs on (and that's just about everywhere).

The `os.walk` visitor

To make it easy to apply an operation to all files in a complete directory tree, Python comes with a utility that scans trees for us and runs code we provide at every directory along the way: the `os.walk` function is called with a directory root name and automatically walks the entire tree at root and below.

Operationally, `os.walk` is a *generator function*—at each directory in the tree, it yields a three-item tuple, containing the name of the current directory as well as lists of both all the files and all the subdirectories in the current directory. Because it's a generator, its walk is usually run by a `for` loop (or other iteration tool); on each iteration, the walker advances to the next subdirectory, and the loop runs its code for the next level of the tree (for instance, opening and searching all the files at that level).

That description might sound complex the first time you hear it, but `os.walk` is fairly straightforward once you get the hang of it. In the following, for example, the loop body's code is run for each directory in the tree rooted at the current working directory (`.`). Along the way, the loop simply prints the directory name and all the files at the current level after prepending the directory name. It's simpler in Python than in English (I removed the PP3E subdirectory for this test to keep the output short):

```
>>> import os
>>> for (dirname, subshere, fileshere) in os.walk('.'):
...     print('[' + dirname + ']')
...     for fname in fileshere:
...         print(os.path.join(dirname, fname))      # handle one file
...
[.]
.\random.bin
.\spam.txt
.\temp.bin
.\temp.txt
[.\parts]
.\parts\part0001
.\parts\part0002
.\parts\part0003
.\parts\part0004
```

In other words, we've coded our own custom and easily changed recursive directory listing tool in Python. Because this may be something we would like to tweak and reuse elsewhere, let's make it permanently available in a module file, as shown in [Example 4-4](#), now that we've worked out the details interactively.

Example 4-4. PP4E\System\Filetools\lister_walk.py

```
"list file tree with os.walk"

import sys, os

def lister(root):                                # for a root dir
    for (thisdir, subshere, fileshere) in os.walk(root): # generate dirs in tree
        print('[' + thisdir + ']')
        for fname in fileshere:                         # print files in this dir
            path = os.path.join(thisdir, fname)          # add dir name prefix
            print(path)

if __name__ == '__main__':
    lister(sys.argv[1])                            # dir name in cmdline
```

When packaged this way, the code can also be run from a shell command line. Here it is being launched with the root directory to be listed passed in as a command-line argument:

```
C:\...\PP4E\System\Filetools> python lister_walk.py C:\temp\test
[C:\temp\test]
C:\temp\test\random.bin
C:\temp\test\spam.txt
C:\temp\test\temp.bin
C:\temp\test\temp.txt
[C:\temp\test\parts]
C:\temp\test\parts\part0001
C:\temp\test\parts\part0002
C:\temp\test\parts\part0003
C:\temp\test\parts\part0004
```

Here's a more involved example of `os.walk` in action. Suppose you have a directory tree of files and you want to find all Python source files within it that reference the `mimetypes` module we'll study in [Chapter 6](#). The following is one (albeit hardcoded and overly specific) way to accomplish this task:

```
>>> import os
>>> matches = []
>>> for (dirname, dirshere, fileshere) in os.walk(r'C:\temp\PP3E\Examples'):
...     for filename in fileshere:
...         if filename.endswith('.py'):
...             pathname = os.path.join(dirname, filename)
...             if 'mimetypes' in open(pathname).read():
...                 matches.append(pathname)
...
>>> for name in matches: print(name)
...
C:\temp\PP3E\Examples\PP3E\Internet\Email\mailtools\mailParser.py
C:\temp\PP3E\Examples\PP3E\Internet\Email\mailtools\mailSender.py
C:\temp\PP3E\Examples\PP3E\Internet\Ftp\mirror\downloadflat.py
C:\temp\PP3E\Examples\PP3E\Internet\Ftp\mirror\downloadflat_modular.py
C:\temp\PP3E\Examples\PP3E\Internet\Ftp\mirror\ftptools.py
C:\temp\PP3E\Examples\PP3E\Internet\Ftp\mirror\uploadflat.py
C:\temp\PP3E\Examples\PP3E\System\Media\playfile.py
```

This code loops through all the files at each level, looking for files with `.py` at the end of their names and which contain the search string. When a match is found, its full name is appended to the results list object; alternatively, we could also simply build a list of all `.py` files and search each in a `for` loop after the walk. Since we're going to code much more general solution to this type of problem in [Chapter 6](#), though, we'll let this stand for now.

If you want to see what's really going on in the `os.walk` generator, call its `_next_` method (or equivalently, pass it to the `next` built-in function) manually a few times, just as the `for` loop does automatically; each time, you advance to the next subdirectory in the tree:

```
>>> gen = os.walk(r'C:\temp\test')
>>> gen._next_()
('C:\\temp\\test', ['parts'], ['random.bin', 'spam.txt', 'temp.bin', 'temp.txt'])
>>> gen._next_()
('C:\\temp\\test\\parts', [], ['part0001', 'part0002', 'part0003', 'part0004'])
>>> gen._next_()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

The library manual documents `os.walk` further than we will here. For instance, it supports bottom-up instead of top-down walks with its optional `topdown=False` argument, and callers may prune tree branches by deleting names in the subdirectories lists of the yielded tuples.

Internally, the `os.walk` call generates filename lists at each level with the `os.listdir` call we met earlier, which collects both file and directory names in no particular order and returns them without their directory paths; `os.walk` segregates this list into subdirectories and files (technically, nondirectories) before yielding a result. Also note that `walk` uses the very same subdirectories list it yields to callers in order to later descend into subdirectories. Because lists are mutable objects that can be changed in place, if your code modifies the yielded subdirectory names list, it will impact what `walk` does next. For example, deleting directory names will prune traversal branches, and sorting the list will order the walk.

Recursive `os.listdir` traversals

The `os.walk` tool does the work of tree traversals for us; we simply provide loop code with task-specific logic. However, it's sometimes more flexible and hardly any more work to do the walking ourselves. The following script recodes the directory listing script with a manual *recursive* traversal function (a function that calls itself to repeat its actions). The `mylister` function in [Example 4-5](#) is almost the same as `lister` in [Example 4-4](#) but calls `os.listdir` to generate file paths manually and calls itself recursively to descend into subdirectories.

```

Example 4-5. PP4E\System\Filetools\lister_recur.py

# list files in dir tree by recursion

import sys, os

def mylister(currdir):
    print('[' + currdir + ']')
    for file in os.listdir(currdir):
        path = os.path.join(currdir, file)           # list files here
        if not os.path.isdir(path):                  # add dir path back
            print(path)
        else:
            mylister(path)                         # recur into subdirs

if __name__ == '__main__':
    mylister(sys.argv[1])                      # dir name in cmdline

```

As usual, this file can be both imported and called or run as a script, though the fact that its result is printed text makes it less useful as an imported component unless its output stream is captured by another program.

When run as a script, this file's output is equivalent to that of [Example 4-4](#), but not identical—unlike the `os.walk` version, our recursive walker here doesn't order the walk to visit files before stepping into subdirectories. It could by looping through the filenames list twice (selecting files first), but as coded, the order is dependent on `os.listdir` results. For most use cases, the walk order would be irrelevant:

```

C:\...\PP4E\System\Filetools> python lister_recur.py C:\temp\test
[C:\temp\test]
[C:\temp\test\parts]
C:\temp\test\parts\part0001
C:\temp\test\parts\part0002
C:\temp\test\parts\part0003
C:\temp\test\parts\part0004
C:\temp\test\random.bin
C:\temp\test\spam.txt
C:\temp\test\temp.bin
C:\temp\test\temp.txt

```

We'll make better use of most of this section's techniques in later examples in [Chapter 6](#) and in this book at large. For example, scripts for copying and comparing directory trees use the tree-walker techniques introduced here. Watch for these tools in action along the way. We'll also code a *find* utility in [Chapter 6](#) that combines the tree traversal of `os.walk` with the filename pattern expansion of `glob.glob`.

Handling Unicode Filenames in 3.X: `listdir`, `walk`, `glob`

Because all normal strings are Unicode in Python 3.X, the directory and file names generated by `os.listdir`, `os.walk`, and `glob.glob` so far in this chapter are technically Unicode strings. This can have some ramifications if your directories contain unusual names that might not decode properly.

Technically, because filenames may contain arbitrary text, the `os.listdir` works in two modes in 3.X: given a `bytes` argument, this function will return filenames as encoded byte strings; given a normal `str` string argument, it instead returns filenames as Unicode strings, decoded per the filesystem's encoding scheme:

```
C:\...\PP4E\System\Filetools> python
>>> import os
>>> os.listdir('.')
['bigext-tree.py', 'bigpy-dir.py', 'bigpy-path.py', 'bigpy-tree.py']

>>> os.listdir(b'.')
[b'bigext-tree.py', b'bigpy-dir.py', b'bigpy-path.py', b'bigpy-tree.py']
```

The byte string version can be used if undecodable file names may be present. Because `os.walk` and `glob.glob` both work by calling `os.listdir` internally, they inherit this behavior by proxy. The `os.walk` tree walker, for example, calls `os.listdir` at each directory level; passing byte string arguments suppresses decoding and returns byte string results:

```
>>> for (dir, subs, files) in os.walk('..'): print(dir)
...
..
..\Environment
..\Filetools
..\Processes

>>> for (dir, subs, files) in os.walk(b".."): print(dir)
...
b'..'
b'..\Environment'
b'..\Filetools'
b'..\Processes'
```

The `glob.glob` tool similarly calls `os.listdir` internally before applying name patterns, and so also returns undecoded byte string names for byte string arguments:

```
>>> glob.glob('.\*')[3]
['.\bigext-out.txt', '.\bigext-tree.py', '.\bigpy-dir.py']
>>>
>>> glob.glob(b'.\*')[3]
[b'.\bigext-out.txt', b'.\bigext-tree.py', b'.\bigpy-dir.py']
```

Given a normal string name (as a command-line argument, for example), you can force the issue by converting to byte strings with manual encoding to suppress decoding:

```
>>> name = '.'
>>> os.listdir(name.encode())[4]
[b'bigext-out.txt', b'bigext-tree.py', b'bigpy-dir.py', b'bigpy-path.py']
```

The upshot is that if your directories may contain names which cannot be decoded according to the underlying platform's Unicode encoding scheme, you may need to pass byte strings to these tools to avoid Unicode encoding errors. You'll get byte strings back, which may be less readable if printed, but you'll avoid errors while traversing directories and files.

This might be especially useful on systems that use simple encodings such as ASCII or Latin-1, but may contain files with arbitrarily encoded names from cross-machine copies, the Web, and so on. Depending upon context, exception handlers may be used to suppress some types of encoding errors as well.

We'll see an example of how this can matter in the first section of [Chapter 6](#), where an undecodable directory name generates an error if printed during a full disk scan (although that specific error seems more related to printing than to decoding in general).

Note that the basic `open` built-in function allows the name of the file being opened to be passed as either Unicode `str` or raw `bytes`, too, though this is used only to name the file initially; the additional mode argument determines whether the file's content is handled in text or binary modes. Passing a byte string filename allows you to name files with arbitrarily encoded names.

Unicode policies: File content versus file names

In fact, it's important to keep in mind that there are two different Unicode concepts related to files: the encoding of file *content* and the encoding of file *name*. Python provides your platform's defaults for these settings in two different attributes; on Windows 7:

```
>>> import sys
>>> sys.getdefaultencoding()          # file content encoding, platform default
'utf-8'
>>> sys.getfilesystemencoding()      # file name encoding, platform scheme
'mbcs'
```

These settings allow you to be explicit when needed—the content encoding is used when data is read and written to the file, and the name encoding is used when dealing with names prior to transferring data. In addition, using `bytes` for file name tools may work around incompatibilities with the underlying file system's scheme, and opening files in binary mode can suppress Unicode decoding errors for content.

As we've seen, though, opening text files in *binary mode* may also mean that the raw and still-encoded text will not match search strings as expected: search strings must also be byte strings encoded per a specific and possibly incompatible encoding scheme. In fact, this approach essentially mimics the behavior of text files in Python 2.X, and underscores why elevating Unicode in 3.X is generally desirable—such text files sometimes may appear to work even though they probably shouldn't. On the other hand, opening text in binary mode to suppress Unicode content decoding and avoid decoding errors might still be useful if you do not wish to skip undecodable files and content is largely irrelevant.

As a rule of thumb, you should try to always provide an encoding name for text content if it might be outside the platform default, and you should rely on the default Unicode API for file names in most cases. Again, see Python’s manuals for more on the Unicode file name story than we have space to cover fully here, and see *Learning Python*, Fourth Edition, for more on Unicode in general.

In [Chapter 6](#), we’re going to put the tools we met in this chapter to realistic use. For example, we’ll apply file and directory tools to implement file splitters, testing systems, directory copies and compares, and a variety of utilities based on tree walking. We’ll find that Python’s directory tools we met here have an enabling quality that allows us to automate a large set of real-world tasks. First, though, [Chapter 5](#) concludes our basic tool survey, by exploring another system topic that tends to weave its way into a wide variety of application domains—parallel processing in Python.

Parallel System Tools

“Telling the Monkeys What to Do”

Most computers spend a lot of time doing nothing. If you start a system monitor tool and watch the CPU utilization, you’ll see what I mean—it’s rare to see one hit 100 percent, even when you are running multiple programs.* There are just too many delays built into software: disk accesses, network traffic, database queries, waiting for users to click a button, and so on. In fact, the majority of a modern CPU’s capacity is often spent in an idle state; faster chips help speed up performance demand peaks, but much of their power can go largely unused.

Early on in computing, programmers realized that they could tap into such unused processing power by running more than one program at the same time. By dividing the CPU’s attention among a set of tasks, its capacity need not go to waste while any given task is waiting for an external event to occur. The technique is usually called *parallel processing* (and sometimes “multiprocessing” or even “multitasking”) because many tasks seem to be performed at once, overlapping and parallel in time. It’s at the heart of modern operating systems, and it gave rise to the notion of multiple-active-window computer interfaces we’ve all come to take for granted. Even within a single program, dividing processing into tasks that run in parallel can make the overall system faster, at least as measured by the clock on your wall.

Just as important is that modern software systems are expected to be responsive to users regardless of the amount of work they must perform behind the scenes. It’s usually unacceptable for a program to stall while busy carrying out a request. Consider an email-browser user interface, for example; when asked to fetch email from a server, the program must download text from a server over a network. If you have enough email or a slow enough Internet link, that step alone can take minutes to finish. But while the

* To watch on Windows, click the Start button, select All Programs → Accessories → System Tools → Resource Monitor, and monitor CPU/Processor usage (Task Manager’s Performance tab may give similar results). The graph rarely climbed above single-digit percentages on my laptop machine while writing this footnote (at least until I typed `while True: pass` in a Python interactive session window...).

download task proceeds, the program as a whole shouldn't stall—it still must respond to screen redraws, mouse clicks, and so on.

Parallel processing comes to the rescue here, too. By performing such long-running tasks in parallel with the rest of the program, the system at large can remain responsive no matter how busy some of its parts may be. Moreover, the parallel processing model is a natural fit for structuring such programs and others; some tasks are more easily conceptualized and coded as components running as independent, parallel entities.

There are two fundamental ways to get tasks running at the same time in Python—*process forks* and *spawned threads*. Functionally, both rely on underlying operating system services to run bits of Python code in parallel. Procedurally, they are very different in terms of interface, portability, and communication. For instance, at this writing direct process forks are not supported on Windows under standard Python (though they are under Cygwin Python on Windows).

By contrast, Python's thread support works on all major platforms. Moreover, the `os.spawn` family of calls provides additional ways to launch programs in a platform-neutral way that is similar to forks, and the `os.popen` and `os.system` calls and `subprocess` module we studied in Chapters 2 and 3 can be used to portably spawn programs with shell commands. The newer `multiprocessing` module offers additional ways to run processes portably in many contexts.

In this chapter, which is a continuation of our look at system interfaces available to Python programmers, we explore Python's built-in tools for starting tasks in parallel, as well as communicating with those tasks. In some sense, we've already begun doing so—`os.system`, `os.popen`, and `subprocess`, which we learned and applied over the last three chapters, are a fairly portable way to spawn and speak with command-line programs, too. We won't repeat full coverage of those tools here.

Instead, our emphasis in this chapter is on introducing more direct techniques—forks, threads, pipes, signals, sockets, and other launching techniques—and on using Python's built-in tools that support them, such as the `os.fork` call and the `threading`, `queue`, and `multiprocessing` modules. In the next chapter (and in the remainder of this book), we use these techniques in more realistic programs, so be sure you understand the basics here before flipping ahead.

One note up front: although the process, thread, and IPC mechanisms we will explore in this chapter are the primary parallel processing tools in Python scripts, the third party domain offers additional options which may serve more advanced or specialized roles. As just one example, the MPI for Python system allows Python scripts to also employ the Message Passing Interface (MPI) standard, allowing Python programs to exploit multiple processors in various ways (see the Web for details). While such specific extensions are beyond our scope in this book, the fundamentals of multiprocessing that we will explore here should apply to more advanced techniques you may encounter in your parallel futures.

Forking Processes

Forked processes are a traditional way to structure parallel tasks, and they are a fundamental part of the Unix tool set. Forking is a straightforward way to start an independent program, whether it is different from the calling program or not. Forking is based on the notion of *copying* programs: when a program calls the fork routine, the operating system makes a new copy of that program and its process in memory and starts running that copy in parallel with the original. Some systems don't really copy the original program (it's an expensive operation), but the new copy works as if it were a literal copy.

After a fork operation, the original copy of the program is called the *parent* process, and the copy created by `os.fork` is called the *child* process. In general, parents can make any number of children, and children can create child processes of their own; all forked processes run independently and in parallel under the operating system's control, and children may continue to run after their parent exits.

This is probably simpler in practice than in theory, though. The Python script in [Example 5-1](#) forks new child processes until you type the letter `q` at the console.

Example 5-1. PP4E\System\Processes\fork1.py

```
"forks child processes until you type 'q'"  
  
import os  
  
def child():  
    print('Hello from child', os.getpid())  
    os._exit(0) # else goes back to parent loop  
  
def parent():  
    while True:  
        newpid = os.fork()  
        if newpid == 0:  
            child()  
        else:  
            print('Hello from parent', os.getpid(), newpid)  
            if input() == 'q': break  
  
parent()
```

Python's process forking tools, available in the `os` module, are simply thin wrappers over standard forking calls in the system library also used by C language programs. To start a new, parallel process, call the `os.fork` built-in function. Because this function generates a copy of the calling program, it returns a different value in each copy: zero in the child process and the process ID of the new child in the parent.

Programs generally test this result to begin different processing in the child only; this script, for instance, runs the `child` function in child processes only.[†]

Because forking is ingrained in the Unix programming model, this script works well on Unix, Linux, and modern Macs. Unfortunately, this script won't work on the standard version of Python for Windows today, because `fork` is too much at odds with the Windows model. Python scripts can always spawn threads on Windows, and the `multi processing` module described later in this chapter provides an alternative for running processes portably, which can obviate the need for process forks on Windows in contexts that conform to its constraints (albeit at some potential cost in low-level control).

The script in [Example 5-1](#) does work on Windows, however, if you use the Python shipped with the Cygwin system (or build one of your own from source-code with Cygwin's libraries). Cygwin is a free, open source system that provides full Unix-like functionality for Windows (and is described further in [“More on Cygwin Python for Windows” on page 185](#)). You can fork with Python on Windows under Cygwin, even though its behavior is not exactly the same as true Unix forks. Because it's close enough for this book's examples, though, let's use it to run our script live:

```
[C:\...\PP4E\System\Processes]$ python fork1.py
Hello from parent 7296 7920
Hello from child 7920

Hello from parent 7296 3988
Hello from child 3988

Hello from parent 7296 6796
Hello from child 6796
q
```

These messages represent three forked child processes; the unique identifiers of all the processes involved are fetched and displayed with the `os.getpid` call. A subtle point: the `child` process function is also careful to exit explicitly with an `os._exit` call. We'll discuss this call in more detail later in this chapter, but if it's not made, the child process would live on after the `child` function returns (remember, it's just a copy of the original process). The net effect is that the child would go back to the loop in `parent` and start forking children of its own (i.e., the parent would have grandchildren). If you delete the exit call and rerun, you'll likely have to type more than one `q` to stop, because multiple processes are running in the `parent` function.

In [Example 5-1](#), each process exits very soon after it starts, so there's little overlap in time. Let's do something slightly more sophisticated to better illustrate multiple forked processes running in parallel. [Example 5-2](#) starts up 5 copies of itself, each copy counting up to 5 with a one-second delay between iterations. The `time.sleep` standard library

[†] At least in the current Python implementation, calling `os.fork` in a Python script actually copies the Python interpreter process (if you look at your process list, you'll see two Python entries after a fork). But since the Python interpreter records everything about your running script, it's OK to think of `fork` as copying your program directly. It really will if Python scripts are ever compiled to binary machine code.

call simply pauses the calling process for a number of seconds (you can pass a floating-point value to pause for fractions of seconds).

Example 5-2. PP4E\System\Processes\fork-count.py

```
"""
fork basics: start 5 copies of this program running in parallel with
the original; each copy counts up to 5 on the same stdout stream--forks
copy process memory, including file descriptors; fork doesn't currently
work on Windows without Cygwin: use os.spawnv or multiprocessing on
Windows instead; spawnv is roughly like a fork+exec combination;
"""

import os, time

def counter(count):
    for i in range(count):                      # run in new process
        time.sleep(1)                           # simulate real work
        print('[%s] => %s' % (os.getpid(), i))

for i in range(5):
    pid = os.fork()
    if pid != 0:
        print('Process %d spawned' % pid)      # in parent: continue
    else:
        counter(5)                          # else in child/new process
        os._exit(0)

print('Main process exiting.')                  # parent need not wait
```

When run, this script starts 5 processes immediately and exits. All 5 forked processes check in with their first count display one second later and every second thereafter. Notice that child processes continue to run, even if the parent process that created them terminates:

```
[C:\...\PP4E\System\Processes]$ python fork-count.py
Process 4556 spawned
Process 3724 spawned
Process 6360 spawned
Process 6476 spawned
Process 6684 spawned
Main process exiting.
[4556] => 0
[3724] => 0
[6360] => 0
[6476] => 0
[6684] => 0
[4556] => 1
[3724] => 1
[6360] => 1
[6476] => 1
[6684] => 1
[4556] => 2
[3724] => 2
[6360] => 2
```

```
[6476] => 2  
[6684] => 2  
...more output omitted...
```

The output of all of these processes shows up on the same screen, because all of them share the standard output stream (and a system prompt may show up along the way, too). Technically, a forked process gets a copy of the original process's global memory, including open file descriptors. Because of that, global objects like files start out with the same values in a child process, so all the processes here are tied to the same single stream. But it's important to remember that global memory is copied, not shared; if a child process changes a global object, it changes only its own copy. (As we'll see, this works differently in threads, the topic of the next section.)

The `fork/exec` Combination

In Examples 5-1 and 5-2, child processes simply ran a function within the Python program and then exited. On Unix-like platforms, forks are often the basis of starting independently running programs that are completely different from the program that performed the `fork` call. For instance, Example 5-3 forks new processes until we type `q` again, but child processes run a brand-new program instead of calling a function in the same file.

Example 5-3. PP4E\System\Processes\fork-exec.py

```
"starts programs until you type 'q'"  
  
import os  
  
parm = 0  
while True:  
    parm += 1  
    pid = os.fork()  
    if pid == 0:  
        os.execl('python', 'python', 'child.py', str(parm)) # overlay program  
        assert False, 'error starting program' # shouldn't return  
    else:  
        print('Child is', pid)  
        if input() == 'q': break
```

If you've done much Unix development, the `fork/exec` combination will probably look familiar. The main thing to notice is the `os.execlp` call in this code. In a nutshell, this call replaces (overlays) the program running in the current process with a brand new program. Because of that, the *combination* of `os.fork` and `os.execlp` means start a new process and run a new program in that process—in other words, launch a new program in parallel with the original program.

os.exec call formats

The arguments to `os.execvp` specify the program to be run by giving command-line arguments used to start the program (i.e., what Python scripts know as `sys.argv`). If successful, the new program begins running and the call to `os.execvp` itself never returns (since the original program has been replaced, there's really nothing to return to). If the call does return, an error has occurred, so we code an `assert` after it that will always raise an exception if reached.

There are a handful of `os.exec` variants in the Python standard library; some allow us to configure environment variables for the new program, pass command-line arguments in different forms, and so on. All are available on both Unix and Windows, and they replace the calling program (i.e., the Python interpreter). `exec` comes in eight flavors, which can be a bit confusing unless you generalize:

`os.execv(program, commandlinesequence)`

The basic “v” `exec` form is passed an executable program’s name, along with a list or tuple of command-line argument strings used to run the executable (that is, the words you would normally type in a shell to start a program).

`os.execl(program, cmdarg1, cmdarg2,... cmdargN)`

The basic “l” `exec` form is passed an executable’s name, followed by one or more command-line arguments passed as individual function arguments. This is the same as `os.execv(program, (cmdarg1, cmdarg2,...))`.

`os.execlp`

`os.execvp`

Adding the letter p to the `execv` and `execl` names means that Python will locate the executable’s directory using your system search-path setting (i.e., `PATH`).

`os.execle`

`os.execve`

Adding a letter e to the `execv` and `execl` names means an extra, *last* argument is a dictionary containing shell environment variables to send to the program.

`os.execvpe`

`os.execle`

Adding the letters p and e to the basic `exec` names means to use the search path *and* to accept a shell environment settings dictionary.

So when the script in [Example 5-3](#) calls `os.execlp`, individually passed parameters specify a command line for the program to be run on, and the word `python` maps to an executable file according to the underlying system search-path setting environment variable (`PATH`). It’s as if we were running a command of the form `python child.py 1` in a shell, but with a different command-line argument on the end each time.

Spawned child program

Just as when typed at a shell, the string of arguments passed to `os.execvp` by the `fork-exec` script in [Example 5-3](#) starts another Python program file, as shown in [Example 5-4](#).

Example 5-4. PP4E\System\Processes\child.py

```
import os, sys
print('Hello from child', os.getpid(), sys.argv[1])
```

Here is this code in action on Linux. It doesn't look much different from the original `fork1.py`, but it's really running a new *program* in each forked process. More observant readers may notice that the child process ID displayed is the same in the parent program and the launched `child.py` program; `os.execvp` simply overlays a program in the same process:

```
[C:\...\PP4E\System\Processes]$ python fork-exec.py
Child is 4556
Hello from child 4556 1

Child is 5920
Hello from child 5920 2

Child is 316
Hello from child 316 3
q
```

There are other ways to start up programs in Python besides the `fork/exec` combination. For example, the `os.system` and `os.popen` calls and `subprocess` module which we explored in [Chapters 2](#) and [3](#) allow us to spawn shell commands. And the `os.spawnv` call and `multiprocessing` module, which we'll meet later in this chapter, allow us to start independent programs and processes more portably. In fact, we'll see later that `multiprocessing`'s process spawning model can be used as a sort of portable replacement for `os.fork` in some contexts (albeit a less efficient one) and used in conjunction with the `os.exec*` calls shown here to achieve a similar effect in standard Windows Python.

We'll see more process fork examples later in this chapter, especially in the program exits and process communication sections, so we'll forego additional examples here. We'll also discuss additional process topics in later chapters of this book. For instance, forks are revisited in [Chapter 12](#) to deal with servers and their *zombies*—dead processes lurking in system tables after their demise. For now, let's move on to threads, a subject which at least some programmers find to be substantially less frightening...

More on Cygwin Python for Windows

As mentioned, the `os.fork` call is present in the Cygwin version of Python on Windows. Even though this call is missing in the standard version of Python for Windows, you can fork processes on Windows with Python if you install and use Cygwin. However, the Cygwin fork call is not as efficient and does not work exactly the same as a fork on true Unix systems.

Cygwin is a free, open source package which includes a library that attempts to provide a Unix-like API for use on Windows machines, along with a set of command-line tools that implement a Unix-like environment. It makes it easier to apply Unix skills and code on Windows computers.

According to its FAQ documentation, though, “Cygwin fork() essentially works like a noncopy on write version of fork() (like old Unix versions used to do). Because of this it can be a little slow. In most cases, you are better off using the spawn family of calls if possible.” Since this book’s fork examples don’t need to care about performance, Cygwin’s fork suffices.

In addition to the fork call, Cygwin provides other Unix tools that would otherwise not be available on all flavors of Windows, including `os.mkfifo` (discussed later in this chapter). It also comes with a `gcc` compiler environment for building C extensions for Python on Windows that will be familiar to Unix developers. As long as you’re willing to use Cygwin libraries to build your application and power your Python, it’s very close to Unix on Windows.

Like all third-party libraries, though, Cygwin adds an extra dependency to your systems. Perhaps more critically, Cygwin currently uses the GNU GPL license, which adds distribution requirements beyond those of standard Python. Unlike using Python itself, shipping a program that uses Cygwin libraries may require that your program’s source code be made freely available (though RedHat offers a “buy-out” option which can relieve you of this requirement). Note that this is a complex legal issue, and you should study Cygwin’s license on your own if this may impact your programs. Its license does, however, impose more constraints than Python’s (Python uses a “BSD”-style license, not the GPL).

Despite licensing issue, Cygwin still can be a great way to get Unix-like functionality on Windows without installing a completely different operating system such as Linux—a more complete but generally more complex option. For more details, see <http://cygwin.com> or run a search for Cygwin on the Web.

See also the standard library’s `multiprocessing` module and `os.spawn` family of calls, covered later in this chapter, for alternative way to start parallel tasks and programs on Unix and Windows that do not require fork and exec calls. To run a simple function call in parallel on Windows (rather than on an external program), also see the section on standard library threads later in this chapter. Threads, `multiprocessing`, and `os.spawn` calls work on Windows in standard Python.

Fourth Edition Update: As I was updating this chapter in February 2010, Cygwin's official Python was still Python 2.5.2. To get Python 3.1 under Cygwin, it had to be built from its source code. If this is still required when you read this, make sure you have `gcc` and `make` installed on your Cygwin, then fetch Python's source code package from python.org, unpack it, and build Python with the usual commands:

```
./configure  
make  
make test  
sudo make install
```

This will install Python as `python3`. The same procedure works on all Unix-like platforms; on OS X and Cygwin, the executable is called `python.exe`; elsewhere it's named `python`. You can generally skip the last two of the above steps if you're willing to run Python 3.1 out of your own build directory. Be sure to also check if Python 3.X is a standard Cygwin package by the time you read this; when building from source you may have to tweak a few files (I had to comment-out a `#define` in `Modules/main.c`), but these are too specific and temporal to get into here.

Threads

Threads are another way to start activities running at the same time. In short, they run a call to a function (or any other type of callable object) in parallel with the rest of the program. Threads are sometimes called “lightweight processes,” because they run in parallel like forked processes, but all of them run within the same single process. While processes are commonly used to start independent programs, threads are commonly used for tasks such as nonblocking input calls and long-running tasks in a GUI. They also provide a natural model for algorithms that can be expressed as independently running tasks. For applications that can benefit from parallel processing, some developers consider threads to offer a number of advantages:

Performance

Because all threads run within the same process, they don't generally incur a big startup cost to copy the process itself. The costs of both copying forked processes and running threads can vary per platform, but threads are usually considered less expensive in terms of performance overhead.

Simplicity

To many observers, threads can be noticeably simpler to program, too, especially when some of the more complex aspects of processes enter the picture (e.g., process exits, communication schemes, and zombie processes, covered in [Chapter 12](#)).

Shared global memory

On a related note, because threads run in a single process, every thread shares the same global memory space of the process. This provides a natural and easy way for threads to communicate—by fetching and setting names or objects accessible to all the threads. To the Python programmer, this means that things like `global`

scope variables, passed objects and their attributes, and program-wide interpreter components such as imported modules are shared among all threads in a program; if one thread assigns a global variable, for instance, its new value will be seen by other threads. Some care must be taken to control access to shared items, but to some this seems generally simpler to use than the process communication tools necessary for forked processes, which we'll meet later in this chapter and book (e.g., pipes, streams, signals, sockets, etc.). Like much in programming, this is not a universally shared view, however, so you'll have to weigh the difference for your programs and platforms yourself.

Portability

Perhaps most important is the fact that threads are more portable than forked processes. At this writing, `os.fork` is not supported by the standard version of Python on Windows, but threads are. If you want to run parallel tasks portably in a Python script today and you are unwilling or unable to install a Unix-like library such as Cygwin on Windows, threads may be your best bet. Python's thread tools automatically account for any platform-specific thread differences, and they provide a consistent interface across all operating systems. Having said that, the relatively new `multiprocessing` module described later in this chapter offers another answer to the process portability issue in some use cases.

So what's the catch? There are three potential downsides you should be aware of before you start spinning your threads:

Function calls versus programs

First of all, threads are not a way—at least, not a direct way—to start up another *program*. Rather, threads are designed to run a call to a *function* (technically, any callable, including bound and unbound methods) in parallel with the rest of the program. As we saw in the prior section, by contrast, forked processes can either call a function or start a new program. Naturally, the threaded function can run scripts with the `exec` built-in function and can start new programs with tools such as `os.system`, `os.popen` and the `subprocess` module, especially if doing so is itself a long-running task. But fundamentally, threads run in-program functions.

In practice, this is usually not a limitation. For many applications, parallel functions are sufficiently powerful. For instance, if you want to implement nonblocking input and output and avoid blocking a GUI or its users with long-running tasks, threads do the job; simply spawn a thread to run a function that performs the potentially long-running task. The rest of the program will continue independently.

Thread synchronization and queues

Secondly, the fact that threads share objects and names in global process memory is both good news and bad news—it provides a communication mechanism, but we have to be careful to synchronize a variety of operations. As we'll see, even operations such as printing are a potential conflict since there is only one `sys.stdout` per process, which is shared by all threads.

Luckily, the Python `queue` module, described in this section, makes this simple: realistic threaded programs are usually structured as one or more *producer* (a.k.a. *worker*) threads that add data to a queue, along with one or more *consumer* threads that take the data off the queue and process it. In a typical threaded GUI, for example, producers may download or compute data and place it on the queue; the consumer—the main GUI thread—checks the queue for data periodically with a timer event and displays it in the GUI when it arrives. Because the shared queue is thread-safe, programs structured this way automatically synchronize much cross-thread data communication.

The global interpreter lock (GIL)

Finally, as we'll learn in more detail later in this section, Python's implementation of threads means that only one thread is ever really running its Python language code in the Python virtual machine at any point in time. Python threads are true operating system threads, but all threads must acquire a single shared lock when they are ready to run, and each thread may be swapped out after running for a short period of time (currently, after a set number of virtual machine instructions, though this implementation may change in Python 3.2).

Because of this structure, the Python language parts of Python threads cannot today be distributed across multiple CPUs on a multi-CPU computer. To leverage more than one CPU, you'll simply need to use process forking, not threads (the amount and complexity of code required for both are roughly the same). Moreover, the parts of a thread that perform long-running tasks implemented as C extensions can run truly independently if they release the GIL to allow the Python code of other threads to run while their task is in progress. Python code, however, cannot truly overlap in time.

The advantage of Python's implementation of threads is performance—when it was attempted, making the virtual machine truly thread-safe reportedly slowed all programs by a factor of two on Windows and by an even larger factor on Linux. Even nonthreaded programs ran at half speed.

Even though the GIL's multiplexing of Python language code makes Python threads less useful for leveraging capacity on multiple CPU machines, threads are still useful as programming tools to implement nonblocking operations, especially in GUIs. Moreover, the newer `multiprocessing` module we'll meet later offers another solution here, too—by providing a portable thread-like API that is implemented with processes, programs can both leverage the simplicity and programmability of threads and benefit from the scalability of independent processes across CPUs.

Despite what you may think after reading the preceding overview, threads are remarkably easy to use in Python. In fact, when a program is started it is already running a thread, usually called the “main thread” of the process. To start new, independent threads of execution within a process, Python code uses either the low-level `_thread` module to run a function call in a spawned thread, or the higher-level `threading` module

to manage threads with high-level class-based objects. Both modules also provide tools for synchronizing access to shared objects with locks.



This book presents both the `_thread` and `threading` modules, and its examples use both interchangeably. Some Python users would recommend that you always use `threading` rather than `_thread` in general. In fact, the latter was renamed from `thread` to `_thread` in 3.X to suggest such a lesser status for it. Personally, I think that is too extreme (and this is one reason this book sometimes uses `as thread` in imports to retain the original module name). Unless you need the more powerful tools in `threading`, the choice is largely arbitrary, and the `threading` module's extra requirements may be unwarranted.

The basic `thread` module does not impose OOP, and as you'll see in the examples of this section, is very straightforward to use. The `threading` module may be better for more complex tasks which require per-thread state retention or joins, but not all threaded programs require its extra tools, and many use threads in more limited scopes. In fact, this is roughly the same as comparing the `os.walk` call and visitor classes we'll meet in [Chapter 6](#)—both have valid audiences and use cases. The most general Python rule of thumb applies here as always: *keep it simple, unless it has to be complex*.

The `_thread` Module

Since the basic `_thread` module is a bit simpler than the more advanced `threading` module covered later in this section, let's look at some of its interfaces first. This module provides a *portable* interface to whatever threading system is available in your platform: its interfaces work the same on Windows, Solaris, SGI, and any system with an installed `pthreads` POSIX threads implementation (including Linux and others). Python scripts that use the Python `_thread` module work on all of these platforms without changing their source code.

Basic usage

Let's start off by experimenting with a script that demonstrates the main thread interfaces. The script in [Example 5-5](#) spawns threads until you reply with a `q` at the console; it's similar in spirit to (and a bit simpler than) the script in [Example 5-1](#), but it goes parallel with threads instead of process forks.

Example 5-5. PP4E\System\Threads\thread1.py

```
"spawn threads until you type 'q'"  
  
import _thread  
  
def child(tid):
```

```

print('Hello from thread', tid)

def parent():
    i = 0
    while True:
        i += 1
        _thread.start_new_thread(child, (i,))
        if input() == 'q': break

parent()

```

This script really contains only two thread-specific lines: the import of the `_thread` module and the thread creation call. To start a thread, we simply call the `_thread.start_new_thread` function, no matter what platform we're programming on.[‡] This call takes a function (or other callable) object and an arguments tuple and starts a new thread to execute a call to the passed function with the passed arguments. It's almost like Python's `function(*args)` call syntax, and similarly accepts an optional keyword arguments dictionary, too, but in this case the function call begins running in parallel with the rest of the program.

Operationally speaking, the `_thread.start_new_thread` call itself returns immediately with no useful value, and the thread it spawns silently exits when the function being run returns (the return value of the threaded function call is simply ignored). Moreover, if a function run in a thread raises an uncaught exception, a stack trace is printed and the thread exits, but the rest of the program continues. With the `_thread` module, the entire program exits silently on most platforms when the main thread does (though as we'll see later, the `threading` module may require special handling if child threads are still running).

In practice, though, it's almost trivial to use threads in a Python script. Let's run this program to launch a few threads; we can run it on both Unix-like platforms and Windows this time, because threads are more portable than process forks—here it is spawning threads on Windows:

```

C:\...\PP4E\System\Threads> python thread1.py
Hello from thread 1

Hello from thread 2

Hello from thread 3

Hello from thread 4
q

```

[‡] The `_thread` examples in this book now all use `start_new_thread`. This call is also available as `thread.start_new` for historical reasons, but this synonym may be removed in a future Python release. As of Python 3.1, both names are still available, but the `help` documentation for `start_new` claims that it is obsolete; in other words, you should probably prefer the other if you care about the future (and this book must!).

Each message here is printed from a new thread, which exits almost as soon as it is started.

Other ways to code threads with `_thread`

Although the preceding script runs a simple function, *any callable object* may be run in the thread, because all threads live in the same process. For instance, a thread can also run a lambda function or bound method of an object (the following code is part of file `thread-alts.py` in the book examples package):

```
import _thread # all 3 print 4294967296

def action(i): # function run in threads
    print(i ** 32)

class Power:
    def __init__(self, i):
        self.i = i
    def action(self): # bound method run in threads
        print(self.i ** 32)

_thread.start_new_thread(action, (2,)) # simple function
_thread.start_new_thread((lambda: action(2)), ()) # lambda function to defer

obj = Power(2)
_thread.start_new_thread(obj.action, ()) # bound method object
```

As we'll see in larger examples later in this book, *bound methods* are especially useful in this role—because they remember both the method function and instance object, they also give access to state information and class methods for use within and during the thread.

More fundamentally, because threads all run in the same process, bound methods run by threads reference the original in-process instance object, not a copy of it. Hence, any changes to its state made in a thread will be visible to all threads automatically. Moreover, since bound methods of a class instance pass for callables interchangeably with simple functions, using them in threads this way just works. And as we'll see later, the fact that they are normal objects also allows them to be stored freely on shared queues.

Running multiple threads

To really understand the power of threads running in parallel, though, we have to do something more long-lived in our threads, just as we did earlier for processes. Let's mutate the `fork-count` program of the prior section to use threads. The script in [Example 5-6](#) starts 5 copies of its `counter` function running in parallel threads.

Example 5-6. PP4E\System\Threads\thread-count.py

"""

thread basics: start 5 copies of a function running in parallel;

```

uses time.sleep so that the main thread doesn't die too early--
this kills all other threads on some platforms; stdout is shared:
thread outputs may be intermixed in this version arbitrarily.

"""

import _thread as thread, time

def counter(myId, count):
    for i in range(count):
        time.sleep(1)                      # simulate real work
        print('[%s] => %s' % (myId, i))

for i in range(5):                         # spawn 5 threads
    thread.start_new_thread(counter, (i, 5)) # each thread loops 5 times

time.sleep(6)
print('Main thread exiting.')              # don't exit too early

```

Each parallel copy of the `counter` function simply counts from zero up to four here and prints a message to standard output for each count.

Notice how this script sleeps for 6 seconds at the end. On Windows and Linux machines this has been tested on, the main thread shouldn't exit while any spawned threads are running if it cares about their work; if it does exit, all spawned threads are immediately terminated. This differs from processes, where spawned children live on when parents exit. Without the sleep here, the spawned threads would die almost immediately after they are started.

This may seem ad hoc, but it isn't required on all platforms, and programs are usually structured such that the main thread naturally lives as long as the threads it starts. For instance, a user interface may start an FTP download running in a thread, but the download lives a much shorter life than the user interface itself. Later in this section, we'll also see different ways to avoid this sleep using global locks and flags that let threads signal their completion.

Moreover, we'll later find that the `threading` module both provides a `join` method that lets us wait for spawned threads to finish explicitly, and refuses to allow a program to exit at all if any of its normal threads are still running (which may be useful in this case, but can require extra work to shut down in others). The `multiprocessing` module we'll meet later in this chapter also allows spawned children to outlive their parents, though this is largely an artifact of its process-based model.

Now, when [Example 5-6](#) is run on Windows 7 under Python 3.1, here is the output I get:

```

C:\...\PP4E\System\Threads> python thread-count.py
[1] => 0
[1] => 0
[0] => 0
[1] => 0
[0] => 0
[2] => 0
[3] => 0

```

```
[3] => 0  
  
[1] => 1  
[3] => 1  
[3] => 1  
[0] => 1[2] => 1  
[3] => 1  
[0] => 1[2] => 1  
[4] => 1  
  
[1] => 2  
[3] => 2[4] => 2  
[3] => 2[4] => 2  
[0] => 2  
[3] => 2[4] => 2  
[0] => 2  
[2] => 2  
[3] => 2[4] => 2  
[0] => 2  
[2] => 2  
  
...more output omitted...  
Main thread exiting.
```

If this looks odd, it's because it should. In fact, this demonstrates probably the most unusual aspect of threads. What's happening here is that the output of the 5 threads run in parallel is intermixed—because all the threaded function calls run in the same process, they all share the same standard output stream (in Python terms, there is just one `sys.stdout` file between them, which is where printed text is sent). The net effect is that their output can be combined and confused arbitrarily. In fact, this script's output can differ on each run. This jumbling of output grew even more pronounced in Python 3, presumably due to its new file output implementation.

More fundamentally, when multiple threads can access a shared resource like this, their access must be synchronized to avoid overlap in time—as explained in the next section.

Synchronizing access to shared objects and names

One of the nice things about threads is that they automatically come with a cross-task communications mechanism: objects and namespaces in a process that span the life of threads are shared by all spawned threads. For instance, because every thread runs in the same process, if one Python thread changes a global scope variable, the change can be seen by every other thread in the process, main or child. Similarly, threads can share and change mutable objects in the process's memory as long as they hold a reference to them (e.g., passed-in arguments). This serves as a simple way for a program's threads to pass information—exit flags, result objects, event indicators, and so on—back and forth to each other.

The downside to this scheme is that our threads must sometimes be careful to avoid changing global objects and names at the same time. If two threads may change a shared object at once, it's not impossible that one of the two changes will be lost (or worse,

will silently corrupt the state of the shared object completely): one thread may step on the work done so far by another whose operations are still in progress. The extent to which this becomes an issue varies per application, and sometimes it isn't an issue at all.

But even things that aren't obviously at risk may be at risk. Files and streams, for example, are shared by all threads in a program; if multiple threads write to one stream at the same time, the stream might wind up with interleaved, garbled data. [Example 5-6](#) of the prior section was a simple demonstration of this phenomenon in action, but it's indicative of the sorts of clashes in time that can occur when our programs go parallel. Even simple changes can go awry if they might happen concurrently. To be robust, threaded programs need to control access to shared global items like these so that only one thread uses them at once.

Luckily, Python's `_thread` module comes with its own easy-to-use tools for synchronizing access to objects shared among threads. These tools are based on the concept of a *lock*—to change a shared object, threads *acquire* a lock, make their changes, and then *release* the lock for other threads to grab. Python ensures that only one thread can hold a lock at any point in time; if others request it while it's held, they are blocked until the lock becomes available. Lock objects are allocated and processed with simple and portable calls in the `_thread` module that are automatically mapped to thread locking mechanisms on the underlying platform.

For instance, in [Example 5-7](#), a lock object created by `_thread.allocate_lock` is acquired and released by each thread around the `print` call that writes to the shared standard output stream.

Example 5-7. PP4E\System\Threads\thread-count-mutex.py

```
"""
synchronize access to stdout: because it is shared global,
thread outputs may be intermixed if not synchronized
"""

import _thread as thread, time

def counter(myId, count):                      # function run in threads
    for i in range(count):
        time.sleep(1)                            # simulate real work
        mutex.acquire()                         # print isn't interrupted now
        print('[%s] => %s' % (myId, i))
        mutex.release()

mutex = thread.allocate_lock()                  # make a global lock object
for i in range(5):                            # spawn 5 threads
    thread.start_new_thread(counter, (i, 5))   # each thread loops 5 times

time.sleep(6)
print('Main thread exiting.')                  # don't exit too early
```

Really, this script simply augments [Example 5-6](#) to synchronize prints with a thread lock. The net effect of the additional lock calls in this script is that no two threads will

ever execute a `print` call at the same point in time; the lock ensures mutually exclusive access to the `stdout` stream. Hence, the output of this script is similar to that of the original version, except that standard output text is never mangled by overlapping prints:

```
C:\...\PP4E\System\Threads> thread-count-mutex.py
[0] => 0
[1] => 0
[3] => 0
[2] => 0
[4] => 0
[0] => 1
[1] => 1
[3] => 1
[2] => 1
[4] => 1
[0] => 2
[1] => 2
[3] => 2
[4] => 2
[2] => 2
[0] => 3
[1] => 3
[3] => 3
[4] => 3
[2] => 3
[0] => 4
[1] => 4
[3] => 4
[4] => 4
[2] => 4
Main thread exiting.
```

Though somewhat platform-specific, the order in which the threads check in with their prints may still be arbitrary from run to run because they execute in parallel (getting work done in parallel is the whole point of threads, after all); but they no longer collide in time while printing their text. We'll see other cases where the lock idiom comes in to play later in this chapter—it's a core component of the multithreading model.

Waiting for spawned thread exits

Besides avoiding print collisions, thread module locks are surprisingly useful. They can form the basis of higher-level synchronization paradigms (e.g., semaphores) and can be used as general thread communication devices.[§] For instance, [Example 5-8](#) uses a global list of locks to know when all child threads have finished.

[§] They cannot, however, be used to directly synchronize *processes*. Since processes are more independent, they usually require locking mechanisms that are more long-lived and external to programs. [Chapter 4](#)'s `os.open` call with an open flag of `O_EXCL` allows scripts to lock and unlock files and so is ideal as a cross-process locking tool. See also the synchronization tools in the `multiprocessing` and `threading` modules and the IPC section later in this chapter for other general synchronization ideas.

Example 5-8. PP4E\System\Threads\thread-count-wait1.py

```
"""
uses mutexes to know when threads are done in parent/main thread,
instead of time.sleep; lock stdout to avoid comingled prints;
"""

import _thread as thread
stdoutmutex = thread.allocate_lock()
exitmutexes = [thread.allocate_lock() for i in range(10)]

def counter(myId, count):
    for i in range(count):
        stdoutmutex.acquire()
        print('[%s] => %s' % (myId, i))
        stdoutmutex.release()
    exitmutexes[myId].acquire()    # signal main thread

for i in range(10):
    thread.start_new_thread(counter, (i, 100))

for mutex in exitmutexes:
    while not mutex.locked(): pass
print('Main thread exiting.')
```

A lock’s `locked` method can be used to check its state. To make this work, the main thread makes one lock per child and tacks them onto a global `exitmutexes` list (remember, the threaded function shares global scope with the main thread). On exit, each thread acquires its lock on the list, and the main thread simply watches for all locks to be acquired. This is much more accurate than naïvely sleeping while child threads run in hopes that all will have exited after the sleep. Run this on your own to see its output—all 10 spawned threads count up to 100 (they run in arbitrarily interleaved order that can vary per run and platform, but their prints run atomically and do not comingle), before the main thread exits.

Depending on how your threads run, this could be even simpler: since threads share global memory anyhow, we can usually achieve the same effect with a simple global list of *integers* instead of locks. In [Example 5-9](#), the module’s namespace (scope) is shared by top-level code and the threaded function, as before. `exitmutexes` refers to the same list object in the main thread and all threads it spawns. Because of that, changes made in a thread are still noticed in the main thread without resorting to extra locks.

Example 5-9. PP4E\System\Threads\thread-count-wait2.py

```
"""
uses simple shared global data (not mutexes) to know when threads
are done in parent/main thread; threads share list but not its items,
assumes list won't move in memory once it has been created initially
"""

import _thread as thread
stdoutmutex = thread.allocate_lock()
```

```

exitmutexes = [False] * 10

def counter(myId, count):
    for i in range(count):
        stdoutmutex.acquire()
        print('[%s] => %s' % (myId, i))
        stdoutmutex.release()
    exitmutexes[myId] = True # signal main thread

for i in range(10):
    thread.start_new_thread(counter, (i, 100))

while False in exitmutexes: pass
print('Main thread exiting.')

```

The output of this script is similar to the prior—10 threads counting to 100 in parallel and synchronizing their prints along the way. In fact, both of the last two counting thread scripts produce roughly the same output as the original *thread_count.py*, albeit without `stdout` corruption and with larger counts and different random ordering of output lines. The main difference is that the main thread exits immediately after (and no sooner than!) the spawned child threads:

```

C:\...\PP4E\System\Threads> python thread-count-wait2.py
...more deleted...
[4] => 98
[6] => 98
[8] => 98
[5] => 98
[0] => 99
[7] => 98
[9] => 98
[1] => 99
[3] => 99
[2] => 99
[4] => 99
[6] => 99
[8] => 99
[5] => 99
[7] => 99
[9] => 99
Main thread exiting.

```

Coding alternatives: busy loops, arguments, and context managers

Notice how the main threads of both of the last two scripts fall into busy-wait loops at the end, which might become significant performance drains in tight applications. If so, simply add a `time.sleep` call in the wait loops to insert a pause between end tests and to free up the CPU for other tasks: this call pauses the calling thread only (in this case, the main one). You might also try experimenting with adding sleep calls to the `thread` function to simulate real work.

Passing in the lock to threaded functions as an *argument* instead of referencing it in the global scope might be more coherent, too. When passed in, all threads reference the same object, because they are all part of the same process. Really, the process's object memory is shared memory for threads, regardless of how objects in that shared memory are referenced (whether through global scope variables, passed argument names, object attributes, or another way).

And while we're at it, the `with` statement can be used to ensure thread operations around a nested block of code, much like its use to ensure file closure in the prior chapter. The thread lock's *context manager* acquires the lock on `with` statement entry and releases it on statement exit regardless of exception outcomes. The net effect is to save one line of code, but also to guarantee lock release when exceptions are possible. [Example 5-10](#) adds all these coding alternatives to our threaded counter script.

Example 5-10. PP4E\System\Threads\thread-count-wait3.py

```
"""
passed in mutex object shared by all threads instead of globals;
use with context manager statement for auto acquire/release;
sleep calls added to avoid busy loops and simulate real work
"""

import _thread as thread, time
stdoutmutex = thread.allocate_lock()
numthreads  = 5
exitmutexes = [thread.allocate_lock() for i in range(numthreads)]

def counter(myId, count, mutex):                      # shared object passed in
    for i in range(count):
        time.sleep(1 / (myId+1))                      # diff fractions of second
        with mutex:                                    # auto acquire/release: with
            print('[%s] => %s' % (myId, i))
        exitmutexes[myId].acquire()                   # global: signal main thread

for i in range(numthreads):
    thread.start_new_thread(counter, (i, 5, stdoutmutex))

while not all(mutex.locked() for mutex in exitmutexes): time.sleep(0.25)
print('Main thread exiting.')
```

When run, the different sleep times per thread make them run more independently:

```
C:\...\PP4E\System\Threads> thread-count-wait3.py
[4] => 0
[3] => 0
[2] => 0
[4] => 1
[1] => 0
[3] => 1
[4] => 2
[2] => 1
[3] => 2
[4] => 3
```

```
[4] => 4
[0] => 0
[1] => 1
[2] => 2
[3] => 3
[3] => 4
[2] => 3
[1] => 2
[2] => 4
[0] => 1
[1] => 3
[1] => 4
[0] => 2
[0] => 3
[0] => 4
Main thread exiting.
```

Of course, threads are for much more than counting. We'll put shared global data to more practical use in “[Adding a User Interface](#)” on page 867, where it will serve as completion signals from child processing threads transferring data over a network to a main thread controlling a tkinter GUI display, and again later in [Chapter 10](#)'s thread-tools and [Chapter 14](#)'s PyMailGUI to post results of email operations to a GUI (watch for “[Preview: GUIs and Threads](#)” on page 208 for more pointers on this topic). Global data shared among threads also turns out to be the basis of queues, which are discussed later in this chapter; each thread gets or puts data using the same shared queue object.

The threading Module

The Python standard library comes with two thread modules: `_thread`, the basic lower-level interface illustrated thus far, and `threading`, a higher-level interface based on objects and classes. The `threading` module internally uses the `_thread` module to implement objects that represent threads and common synchronization tools. It is loosely based on a subset of the Java language's threading model, but it differs in ways that only Java programmers would notice.^{||} [Example 5-11](#) morphs our counting threads example again to demonstrate this new module's interfaces.

Example 5-11. PP4E\System\Threads\thread-classes.py

```
"""
thread class instances with state and run() for thread's action;
uses higher-level Java-like threading module object join method (not
mutexes or shared global vars) to know when threads are done in main
parent thread; see library manual for more details on threading;
"""

import threading
```

^{||} But in case this means you, Python's lock and condition variables are distinct objects, not something inherent in all objects, and Python's `Thread` class doesn't have all the features of Java's. See Python's library manual for further details.

```

class Mythread(threading.Thread):           # subclass Thread object
    def __init__(self, myId, count, mutex):
        self.myId = myId
        self.count = count                  # per-thread state information
        self.mutex = mutex                 # shared objects, not globals
        threading.Thread.__init__(self)

    def run(self):                      # run provides thread logic
        for i in range(self.count):      # still sync stdout access
            with self.mutex:
                print('[%s] => %s' % (self.myId, i))

    stdoutmutex = threading.Lock()       # same as thread.allocate_lock()
threads = []
for i in range(10):
    thread = Mythread(i, 100, stdoutmutex) # make/start 10 threads
    thread.start()                     # starts run method in a thread
    threads.append(thread)

for thread in threads:
    thread.join()                     # wait for thread exits
print('Main thread exiting.')

```

The output of this script is the same as that shown for its ancestors earlier (again, threads may be randomly distributed in time, depending on your platform):

```

C:\...\PP4E\System\Threads> python thread-classes.py
...more deleted...
[4] => 98
[8] => 97
[9] => 97
[5] => 98
[3] => 99
[6] => 98
[7] => 98
[4] => 99
[8] => 98
[9] => 98
[5] => 99
[6] => 99
[7] => 99
[8] => 99
[9] => 99
Main thread exiting.

```

Using the `threading` module this way is largely a matter of specializing classes. Threads in this module are implemented with a `Thread` object, a Python class which we may customize per application by providing a `run` method that defines the thread's action. For example, this script subclasses `Thread` with its own `Mythread` class; the `run` method will be executed by the `Thread` framework in a new thread when we make a `Mythread` and call its `start` method.

In other words, this script simply provides methods expected by the `Thread` framework. The advantage of taking this more coding-intensive route is that we get both per-thread state information (the usual `instance` attribute namespace), and a set of additional thread-related tools from the framework “for free.” The `Thread.join` method used near the end of this script, for instance, waits until the thread exits (by default); we can use this method to prevent the main thread from exiting before its children, rather than using the `time.sleep` calls and global locks and variables we relied on in earlier threading examples.

The example script also uses `threading.Lock` to synchronize stream access as before (though this name is really just a synonym for `_thread.allocate_lock` in the current implementation). The `threading` module may provide the extra structure of classes, but it doesn’t remove the specter of concurrent updates in the multithreading model in general.

Other ways to code threads with `threading`

The `Thread` class can also be used to start a simple function, or any other type of callable object, without coding subclasses at all—if not redefined, the `Thread` class’s default `run` method simply calls whatever you pass to its constructor’s `target` argument, with any provided arguments passed to `args` (which defaults to `()` for none). This allows us to use `Thread` to run simple functions, too, though this call form is not noticeably simpler than the basic `_thread` module. For instance, the following code snippets sketch four different ways to spawn the same sort of thread (see `four-threads.py` in the examples tree; you can run all four in the same script, but would have to also synchronize prints to avoid overlap):

```
import threading, _thread
def action(i):
    print(i ** 32)

# subclass with state
class Mythread(threading.Thread):
    def __init__(self, i):
        self.i = i
        threading.Thread.__init__(self)
    def run(self):                      # redefine run for action
        print(self.i ** 32)
Mythread(2).start()                  # start invokes run()

# pass action in
thread = threading.Thread(target=(lambda: action(2)))      # run invokes target
thread.start()

# same but no lambda wrapper for state
threading.Thread(target=action, args=(2,)).start()          # callable plus its args

# basic thread module
_thread.start_new_thread(action, (2,))                      # all-function interface
```

As a rule of thumb, class-based threads may be better if your threads require per-thread state, or can leverage any of OOP’s many benefits in general. Your thread classes don’t necessarily have to subclass `Thread`, though. In fact, just as in the `_thread` module, the thread’s target in `threading` may be *any type of callable object*. When combined with techniques such as bound methods and nested scope references, the choice between coding techniques becomes even less clear-cut:

```
# a non-thread class with state, OOP
class Power:
    def __init__(self, i):
        self.i = i
    def action(self):
        print(self.i ** 32)

obj = Power(2)
threading.Thread(target=obj.action).start()          # thread runs bound method

# nested scope to retain state
def action(i):
    def power():
        print(i ** 32)
    return power

threading.Thread(target=action(2)).start()            # thread runs returned function

# both with basic thread module
_thread.start_new_thread(obj.action, ())           # thread runs a callable object
_thread.start_new_thread(action(2), ())
```

As usual, the threading APIs are as flexible as the Python language itself.

Synchronizing access to shared objects and names revisited

Earlier, we saw how print operations in threads need to be synchronized with locks to avoid overlap, because the output stream is shared by all threads. More formally, threads need to synchronize their changes to any item that may be shared across thread in a process—both objects and namespaces. Depending on a given program’s goals, this might include:

- Mutable object in memory (passed or otherwise referenced objects whose lifetimes span threads)
- Names in global scopes (changeable variables outside thread functions and classes)
- The contents of modules (each has just one shared copy in the system’s module table)

For instance, even simple global variables can require coordination if concurrent updates are possible, as in [Example 5-12](#).

```

Example 5-12. PP4E\System\Threads\thread-add-random.py
"prints different results on different runs on Windows 7"

import threading, time
count = 0

def adder():
    global count
    count = count + 1           # update a shared name in global scope
    time.sleep(0.5)             # threads share object memory and global names
    count = count + 1

threads = []
for i in range(100):
    thread = threading.Thread(target=adder, args=())
    thread.start()
    threads.append(thread)

for thread in threads: thread.join()
print(count)

```

Here, 100 threads are spawned to update the same global scope variable twice (with a sleep between updates to better interleave their operations). When run on Windows 7 with Python 3.1, different runs produce different results:

```

C:\...\PP4E\System\Threads> thread-add-random.py
189

C:\...\PP4E\System\Threads> thread-add-random.py
200

C:\...\PP4E\System\Threads> thread-add-random.py
194

C:\...\PP4E\System\Threads> thread-add-random.py
191

```

This happens because threads overlap arbitrarily in time: statements, even the simple assignment statements like those here, are not guaranteed to run to completion by themselves (that is, they are not atomic). As one thread updates the global, it may be using the partial result of another thread's work in progress. The net effect is this seemingly random behavior. To make this script work correctly, we need to again use thread locks to synchronize the updates—when [Example 5-13](#) is run, it always prints 200 as expected.

```

Example 5-13. PP4E\System\Threads\thread-add-synch.py
"prints 200 each time, because shared resource access synchronized"

import threading, time
count = 0

def adder(addlock):           # shared lock object passed in

```

```

global count
with addlock:
    count = count + 1           # auto acquire/release around stmt
    time.sleep(0.5)
with addlock:
    count = count + 1           # only 1 thread updating at once

addlock = threading.Lock()
threads = []
for i in range(100):
    thread = threading.Thread(target=adder, args=(addlock,))
    thread.start()
    threads.append(thread)

for thread in threads: thread.join()
print(count)

```

Although some basic operations in the Python language are atomic and need not be synchronized, you’re probably better off doing so for every potential concurrent update. Not only might the set of atomic operations change over time, but the internal implementation of threads in general can as well (and in fact, it may in Python 3.2, as described ahead).

Of course, this is an artificial example (spawning 100 threads to add twice isn’t exactly a real-world use case for threads!), but it illustrates the issues that threads must address for any sort of potentially concurrent updates to shared object or name. Luckily, for many or most realistic applications, the `queue` module of the next section can make thread synchronization an automatic artifact of program structure.

Before we move ahead, I should point out that besides `Thread` and `Lock`, the `threading` module also includes higher-level objects for synchronizing access to shared items (e.g., `Semaphore`, `Condition`, `Event`)—many more, in fact, than we have space to cover here; see the library manual for details. For more examples of threads and forks in general, see the remainder this chapter as well as the examples in the GUI and network scripting parts of this book. We will thread GUIs, for instance, to avoid blocking them, and we will thread and fork network servers to avoid denying service to clients.

We’ll also explore the `threading` module’s approach to program exits in the absence of `join` calls in conjunction with queues—our next topic.

The `queue` Module

You can synchronize your threads’ access to shared resources with locks, but you often don’t have to. As mentioned, realistically scaled threaded programs are often structured as a set of producer and consumer threads, which communicate by placing data on, and taking it off of, a shared queue. As long as the queue synchronizes access to itself, this automatically synchronizes the threads’ interactions.

The Python `queue` module implements this storage device. It provides a standard queue data structure—a first-in first-out (fifo) list of Python objects, in which items are added on one end and removed from the other. Like normal lists, the queues provided by this module may contain any type of Python object, including both simple types (strings, lists, dictionaries, and so on) and more exotic types (class instances, arbitrary callables like functions and bound methods, and more).

Unlike normal lists, though, the queue object is automatically controlled with thread lock acquire and release operations, such that only one thread can modify the queue at any given point in time. Because of this, programs that use a queue for their cross-thread communication will be thread-safe and can usually avoid dealing with locks of their own for data passed between threads.

Like the other tools in Python’s threading arsenal, queues are surprisingly simple to use. The script in [Example 5-14](#), for instance, spawns two consumer threads that watch for data to appear on the shared queue and four producer threads that place data on the queue periodically after a sleep interval (each of their sleep durations differs to simulate a real, long-running task). In other words, this program runs 7 threads (including the main one), 6 of which access the shared queue in parallel.

Example 5-14. PP4E\System\Threads\queuetest.py

```
"producer and consumer threads communicating with a shared queue"

numconsumers = 2                      # how many consumers to start
numproducers = 4                        # how many producers to start
nummessages  = 4                        # messages per producer to put

import _thread as thread, queue, time
safeprint = thread.allocate_lock()       # else prints may overlap
dataQueue = queue.Queue()               # shared global, infinite size

def producer(idnum):
    for msgnum in range(nummessages):
        time.sleep(idnum)
        dataQueue.put('[producer id=%d, count=%d]' % (idnum, msgnum))

def consumer(idnum):
    while True:
        time.sleep(0.1)
        try:
            data = dataQueue.get(block=False)
        except queue.Empty:
            pass
        else:
            with safeprint:
                print('consumer', idnum, 'got =>', data)

if __name__ == '__main__':
    for i in range(numconsumers):
        thread.start_new_thread(consumer, (i,))
    for i in range(numproducers):
```

```
    thread.start_new_thread(producer, (i,))
    time.sleep(((numproducers-1) * nummessages) + 1)
    print('Main thread exit.')
```

Before I show you this script’s output, I want to highlight a few points in its code.

Arguments versus globals

Notice how the queue is assigned to a global variable; because of that, it is shared by all of the spawned threads (all of them run in the same process and in the same global scope). Since these threads change an object instead of a variable name, it would work just as well to pass the queue object in to the threaded functions as an argument—the queue is a shared object in memory, regardless of how it is referenced (see *queueTest2.py* in the examples tree for a full version that does this):

```
dataQueue = queue.Queue()                      # shared object, infinite size

def producer(idnum, dataqueue):
    for msgnum in range(nummessages):
        time.sleep(idnum)
        dataqueue.put('[producer id=%d, count=%d]' % (idnum, msgnum))

def consumer(idnum, dataqueue): ...

if __name__ == '__main__':
    for i in range(numproducers):
        thread.start_new_thread(producer, (i, dataQueue))
    for i in range(numproducers):
        thread.start_new_thread(producer, (i, dataQueue))
```

Program exit with child threads

Also notice how this script exits when the main thread does, even though consumer threads are still running in their infinite loops. This works fine on Windows (and most other platforms)—with the basic `_thread` module, the program ends silently when the main thread does. This is why we’ve had to sleep in some examples to give threads time to do their work, but is also why we do not need to be concerned about exiting while consumer threads are still running here.

In the alternative `threading` module, though, the program will not exit if any spawned threads are running, unless they are set to be *daemon* threads. Specifically, the entire program exits when only daemon threads are left. Threads inherit a default initial daemonic value from the thread that creates them. The initial thread of a Python program is considered not daemonic, though alien threads created outside this module’s control are considered daemonic (including some threads created in C code). To override inherited defaults, a thread object’s `daemon` flag can be set manually. In other words, non-daemon threads prevent program exit, and programs by default do not exit until all `threading`-managed threads finish.

This is either a feature or nonfeature, depending on your program—it allows spawned worker threads to finish their tasks in the absence of `join` calls or sleeps, but it can prevent programs like the one in [Example 5-14](#) from shutting down when they wish. To make this example work with `threading`, use the following alternative code (see `queuetest3.py` in the examples tree for a complete version of this, as well as `thread-count-threading.py`, also in the tree, for a case where this refusal to exit can come in handy):

```
import threading, queue, time

def producer(idnum, dataqueue): ...

def consumer(idnum, dataqueue): ...

if __name__ == '__main__':
    for i in range(numconsumers):
        thread = threading.Thread(target=consumer, args=(i, dataQueue))
        thread.daemon = True # else cannot exit!
        thread.start()

    waitfor = []
    for i in range(numproducers):
        thread = threading.Thread(target=producer, args=(i, dataQueue))
        waitfor.append(thread)
        thread.start()

    for thread in waitfor: thread.join()    # or time.sleep() long enough here
    print('Main thread exit.')
```

We'll revisit the daemons and exits issue in [Chapter 10](#) while studying GUIs; as we'll see, it's no different in that context, except that the main thread is usually the GUI itself.

Running the script

Now, as coded in [Example 5-14](#), the following is the output of this example when run on my Windows machine. Notice that even though the queue automatically coordinates the communication of data between the threads, this script still must use a lock to manually synchronize access to the standard output stream; queues synchronize data passing, but some programs may still need to use locks for other purposes. As in prior examples, if the `safeprint` lock is not used, the printed lines from one consumer may be intermixed with those of another. It is not impossible that a consumer may be paused in the middle of a print operation:

```
C:\...\PP4E\System\Threads> queuetest.py
consumer 1 got => [producer id=0, count=0]
consumer 0 got => [producer id=0, count=1]
consumer 1 got => [producer id=0, count=2]
consumer 0 got => [producer id=0, count=3]
consumer 1 got => [producer id=1, count=0]
consumer 1 got => [producer id=2, count=0]
consumer 0 got => [producer id=1, count=1]
consumer 1 got => [producer id=3, count=0]
consumer 0 got => [producer id=1, count=2]
```

```
consumer 1 got => [producer id=2, count=1]
consumer 1 got => [producer id=1, count=3]
consumer 1 got => [producer id=3, count=1]
consumer 0 got => [producer id=2, count=2]
consumer 1 got => [producer id=2, count=3]
consumer 1 got => [producer id=3, count=2]
consumer 1 got => [producer id=3, count=3]
Main thread exit.
```

Try adjusting the parameters at the top of this script to experiment with different scenarios. A single consumer, for instance, would simulate a GUI’s main thread. Here is the output of a single-consumer run—producers still add to the queue in fairly random fashion, because threads run in parallel with each other and with the consumer:

```
C:\...\PP4E\System\Threads> queuetest.py
consumer 0 got => [producer id=0, count=0]
consumer 0 got => [producer id=0, count=1]
consumer 0 got => [producer id=0, count=2]
consumer 0 got => [producer id=0, count=3]
consumer 0 got => [producer id=1, count=0]
consumer 0 got => [producer id=2, count=0]
consumer 0 got => [producer id=1, count=1]
consumer 0 got => [producer id=3, count=0]
consumer 0 got => [producer id=1, count=2]
consumer 0 got => [producer id=2, count=1]
consumer 0 got => [producer id=1, count=3]
consumer 0 got => [producer id=3, count=1]
consumer 0 got => [producer id=2, count=2]
consumer 0 got => [producer id=2, count=3]
consumer 0 got => [producer id=3, count=2]
consumer 0 got => [producer id=3, count=3]
Main thread exit.
```

In addition to the basics used in our script, queues may be fixed or infinite in size, and get and put calls may or may not block; see the Python library manual for more details on queue interface options. Since we just simulated a typical GUI structure, though, let’s explore the notion a bit further.

Preview: GUIs and Threads

We will return to threads and queues and see additional thread and queue examples when we study GUIs later in this book. The PyMailGUI example in [Chapter 14](#), for instance, will make extensive use of thread and queue tools introduced here and developed further in [Chapter 10](#), and [Chapter 9](#) will discuss threading in the context of the tkinter GUI toolkit once we’ve had a chance to study it. Although we can’t get into code at this point, threads are usually an integral part of most nontrivial GUIs. In fact, the activity model of many GUIs is a combination of threads, a queue, and a timer-based loop.

Here’s why. In the context of a GUI, any operation that can block or take a long time to complete must be spawned off to run in parallel so that the GUI (the main thread)

remains active and continues responding to its users. Although such tasks can be run as processes, the efficiency and shared-state model of threads make them ideal for this role. Moreover, since most GUI toolkits do not allow multiple threads to update the GUI in parallel, updates are best restricted to the main thread.

Because only the main thread should generally update the display, GUI programs typically take the form of a main GUI thread and one or more long-running producer threads—one for each long-running task being performed. To synchronize their points of interface, all of the threads share data on a global queue: non-GUI threads post results, and the GUI thread consumes them.

More specifically:

- The *main thread* handles all GUI updates and runs a timer-based loop that wakes up periodically to check for new data on the queue to be displayed on-screen. In Python’s tkinter toolkit, for instance, the widget `after(msecs, func, *args)` method can be used to schedule queue-check events. Because such events are dispatched by the GUI’s event processor, all GUI updates occur only in this main thread (and often must, due to the lack of thread safety in GUI toolkits).
- The *child threads* don’t do anything GUI-related. They just produce data and put it on the queue to be picked up by the main thread. Alternatively, child threads can place a callback function on the queue, to be picked up and run by the main thread. It’s not generally sufficient to simply pass in a GUI update callback function from the main thread to the child thread and run it from there; the function in shared memory will still be executed in the child thread, and potentially in parallel with other threads.

Since threads are much more responsive than a timer event loop in the GUI, this scheme both avoids blocking the GUI (producer threads run in parallel with the GUI), and avoids missing incoming events (producer threads run independent of the GUI event loop and as fast as they can). The main GUI thread will display the queued results as quickly as it can, in the context of a slower GUI event loop.

Also keep in mind that regardless of the thread safety of a GUI toolkit, threaded GUI programs must still adhere to the principles of threaded programs in general—access to shared resources may still need to be synchronized if it falls outside the scope of the producer/consumer shared queue model. If spawned threads might also update another shared state that is used by the main GUI thread, thread locks may also be required to avoid operation overlap. For instance, spawned threads that download and cache email probably cannot overlap with others that use or update the same cache. That is, queues may not be enough; unless you can restrict threads’ work to queuing their results, threaded GUIs still must address concurrent updates.

We’ll see how the threaded GUI model can be realized in code later in this book. For more on this subject, see especially the discussion of threaded tkinter GUIs in

[Chapter 9](#), the thread queue tools implemented in [Chapter 10](#), and the PyMailGUI example in [Chapter 14](#).

Later in this chapter, we'll also meet the `multiprocessing` module, whose process and queue support offers new options for implementing this GUI model using processes instead of threads; as such, they work around the limitations of the thread GIL, but may incur extra performance overheads that can vary per platform, and may not be directly usable at all in threading contexts (the direct shared and mutable object state of threads is not supported, though messaging is). For now, let's cover a few final thread fine points.

Thread Timers versus GUI Timers

Interestingly, the `threading` module exports a general timer function, which, like the `tkinter` widget `after` method, can be used to run another function after a timer has expired:

```
Timer(N.M, somefunc).start() # after N.M seconds run somefunc
```

Timer objects have a `start()` method to set the timer as well as a `cancel()` method to cancel the scheduled event, and they implement the wait state in a spawned thread. For example, the following prints a message after 5.5 seconds:

```
>>> import sys
>>> from threading import Timer
>>> t = Timer(5.5, lambda: print('Spam!'))    # spawned thread
>>> t.start()
>>> Spam!
```

This may be useful in a variety of contexts, but it doesn't quite apply to GUIs: because the time-delayed function call is run in a spawned thread, not in the main GUI thread, it should not generally perform GUI updates. Because the `tkinter` `after` method is run from the main thread's event processing loop instead, it runs in the main GUI thread and can freely update the GUI.

As a preview, for instance, the following displays a pop-up message window in 5.5 seconds in the main thread of a `tkinter` GUI (you might also have to run `win.mainloop()` in some interfaces):

```
>>> from tkinter import Tk
>>> from tkinter.messagebox import showinfo
>>> win = Tk()
>>> win.after(5500, lambda: showinfo('Popup', 'Spam!'))
```

The last call here schedules the function to be run once in the main GUI thread, but it does not pause the caller during the wait, and so does not block the GUI. It's equivalent to this simpler form:

```
>>> win.after(5500, showinfo, 'Popup', 'Spam')
```

Stay tuned for much more on `tkinter` in the next part of this book, and watch for the full story on its `after` timer events in [Chapter 9](#) and the roles of threads in GUIs in [Chapter 10](#).

More on the Global Interpreter Lock

Although it's a lower-level topic than you generally need to do useful thread work in Python, the implementation of Python's threads can have impacts on both performance and coding. This section summarizes implementation details and some of their ramifications.



Threads implementation in the upcoming Python 3.2: This section describes the current implementation of threads up to and including Python 3.1. At this writing, Python 3.2 is still in development, but one of its likely enhancements is a new version of the GIL that provides better performance, especially on some multicore CPUs. The new GIL implementation will still synchronize access to the PVM (Python language code is still multiplexed as before), but it will use a context switching scheme that is more efficient than the current N-bytecode-instruction approach.

Among other things, the current `sys.setcheckinterval` call will likely be replaced with a timer duration call in the new scheme. Specifically, the concept of a check interval for thread switches will be abandoned and replaced by an absolute time duration expressed in seconds. It's anticipated that this duration will default to 5 milliseconds, but it will be tunable through `sys.setswitchinterval`.

Moreover, there have been a variety of plans made to remove the GIL altogether (including goals of the Unladen Swallow project being conducted by Google employees), though none have managed to produce any fruit thus far. Since I cannot predict the future, please see Python release documents to follow this (well...) thread.

Strictly speaking, Python currently uses the *global interpreter lock* (GIL) mechanism introduced at the start of this section, which guarantees that one thread, at most, is running code within the Python interpreter at any given point in time. In addition, to make sure that each thread gets a chance to run, the interpreter automatically switches its attention between threads at regular intervals (in Python 3.1, by releasing and acquiring the lock after a number of bytecode instructions) as well as at the start of long-running operations (e.g., on some file input/output requests).

This scheme avoids problems that could arise if multiple threads were to update Python system data at the same time. For instance, if two threads were allowed to simultaneously change an object's reference count, the result might be unpredictable. This scheme can also have subtle consequences. In this chapter's threading examples, for instance, the `stdout` stream can be corrupted unless each thread's call to write text is synchronized with thread locks.

Moreover, even though the GIL prevents more than one Python thread from running at the same time, it is not enough to ensure thread safety in general, and it does not

address higher-level synchronization issues at all. For example, as we saw, when more than one thread might attempt to *update* the same variable at the same time, the threads should generally be given exclusive access to the object with locks. Otherwise, it's not impossible that thread switches will occur in the middle of an update statement's bytecode.

Locks are not strictly required for all shared object access, especially if a single thread updates an object inspected by other threads. As a rule of thumb, though, you should generally use locks to synchronize threads whenever update rendezvous are possible instead of relying on artifacts of the current thread implementation.

The thread switch interval

Some concurrent updates might work without locks if the thread-switch interval is set high enough to allow each thread to finish without being swapped out. The `sys.setcheckinterval(N)` call sets the frequency with which the interpreter checks for things like thread switches and signal handlers.

This interval defines the number of bytecode instructions before a switch. It does not need to be reset for most programs, but it can be used to tune thread performance. Setting higher values means switches happen less often: threads incur less overhead but they are less responsive to events. Setting lower values makes threads more responsive to events but increases thread switch overhead.

Atomic operations

Because of the way Python uses the GIL to synchronize threads' access to the virtual machine, whole statements are not generally thread-safe, but each bytecode instruction is. Because of this bytecode indivisibility, some Python language operations are thread-safe—also called *atomic*, because they run without interruption—and do not require the use of locks or queues to avoid concurrent update issues. For instance, as of this writing, `list.append`, fetches and some assignments for variables, list items, dictionary keys, and object attributes, and other operations were still atomic in standard C Python; others, such as `x = x+1` (and any operation in general that reads data, modifies it, and writes it back) were not.

As mentioned earlier, though, relying on these rules is a bit of a gamble, because they require a deep understanding of Python internals and may vary per release. Indeed, the set of atomic operations may be radically changed if a new free-threaded implementation ever appears. As a rule of thumb, it may be easier to use locks for all access to global and shared objects than to try to remember which types of access may or may not be safe across multiple threads.

C API thread considerations

Finally, if you plan to mix Python with C, also see the thread interfaces described in the Python/C API standard manual. In threaded programs, C extensions must release

and reacquire the GIL around long-running operations to let the Python language portions of other Python threads run during the wait. Specifically, the long-running C extension function should release the lock on entry and reacquire it on exit when resuming Python code.

Also note that even though the Python code in Python threads cannot truly overlap in time due to the GIL synchronization, the C-coded portions of threads can. Any number may be running in parallel, as long as they do work outside the scope of the Python virtual machine. In fact, C threads may overlap both with other C threads and with Python language threads run in the virtual machine. Because of this, splitting code off to C libraries is one way that Python applications can still take advantage of multi-CPU machines.

Still, it may often be easier to leverage such machines by simply writing Python programs that fork processes instead of starting threads. The complexity of process and thread code is similar. For more on C extensions and their threading requirements, see [Chapter 20](#). In short, Python includes C language tools (including a pair of GIL management macros) that can be used to wrap long-running operations in C-coded extensions and that allow other Python language threads to run in parallel.

A process-based alternative: `multiprocessing` (ahead)

By now, you should have a basic grasp of parallel processes and threads, and Python’s tools that support them. Later in this chapter, we’ll revisit both ideas to study the `multiprocessing` module—a standard library tool that seeks to combine the simplicity and portability of threads with the benefits of processes, by implementing a threading-like API that runs processes instead of threads. It seeks to address the portability issue of processes, as well as the multiple-CPU limitations imposed in threads by the GIL, but it cannot be used as a replacement for forking in some contexts, and it imposes some constraints that threads do not, which stem from its process-based model (for instance, mutable object state is not directly shared because objects are copied across process boundaries, and unpickleable objects such as bound methods cannot be as freely used).

Because the `multiprocessing` module also implements tools to simplify tasks such as inter-process communication and exit status, though, let’s first get a handle on Python’s support in those domains as well, and explore some more process and thread examples along the way.

Program Exits

As we’ve seen, unlike C, there is no “main” function in Python. When we run a program, we simply execute all of the code in the top-level file, from top to bottom (i.e., in the filename we listed in the command line, clicked in a file explorer, and so on). Scripts

normally exit when Python falls off the end of the file, but we may also call for program exit explicitly with tools in the `sys` and `os` modules.

sys Module Exits

For example, the built-in `sys.exit` function ends a program when called, and earlier than normal:

```
>>> sys.exit(N)           # exit with status N, else exits on end of script
```

Interestingly, this call really just raises the built-in `SystemExit` exception. Because of this, we can catch it as usual to intercept early exits and perform cleanup activities; if uncaught, the interpreter exits as usual. For instance:

```
C:\...\PP4E\System> python
>>> import sys
>>> try:
...     sys.exit()          # see also: os._exit, Tk().quit()
... except SystemExit:
...     print('ignoring exit')
...
ignoring exit
>>>
```

Programming tools such as debuggers can make use of this hook to avoid shutting down. In fact, explicitly raising the built-in `SystemExit` exception with a Python `raise` statement is equivalent to calling `sys.exit`. More realistically, a `try` block would catch the exit exception raised elsewhere in a program; the script in [Example 5-15](#), for instance, exits from within a processing function.

Example 5-15. PP4E\System\Exits\testexit_sys.py

```
def later():
    import sys
    print('Bye sys world')
    sys.exit(42)
    print('Never reached')

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

Running this program as a script causes it to exit before the interpreter falls off the end of the file. But because `sys.exit` raises a Python exception, importers of its function can trap and override its exit exception or specify a `finally` cleanup block to be run during program exit processing:

```
C:\...\PP4E\System\Exits> python testexit_sys.py
Bye sys world

C:\...\PP4E\System\Exits> python
>>> from testexit_sys import later
>>> try:
...     later()
... except SystemExit:
```

```

...     print('Ignored...')
...
Bye sys world
Ignored...
>>> try:
...     later()
... finally:
...     print('Cleanup')
...
Bye sys world
Cleanup

C:\...\PP4E\System\Exits>                                # interactive session process exits

```

os Module Exits

It's possible to exit Python in other ways, too. For instance, within a forked child process on Unix, we typically call the `os._exit` function rather than `sys.exit`; threads may exit with a `_thread.exit` call; and tkinter GUI applications often end by calling something named `Tk().quit()`. We'll meet the tkinter module later in this book; let's take a look at `os` exits here.

On `os._exit`, the calling process exits immediately instead of raising an exception that could be trapped and ignored. In fact, the process also exits without flushing output stream buffers or running cleanup handlers (defined by the `atexit` standard library module), so this generally should be used only by child processes after a fork, where overall program shutdown actions aren't desired. [Example 5-16](#) illustrates the basics.

Example 5-16. PP4E\System\Exits\testexit_os.py

```

def outahere():
    import os
    print('Bye os world')
    os._exit(99)
    print('Never reached')

if __name__ == '__main__': outahere()

```

Unlike `sys.exit`, `os._exit` is immune to both `try/except` and `try/finally` interception:

```

C:\...\PP4E\System\Exits> python testexit_os.py
Bye os world

C:\...\PP4E\System\Exits> python
>>> from testexit_os import outahere
>>> try:
...     outahere()
... except:
...     print('Ignored')
...
Bye os world                                # exits interactive process

C:\...\PP4E\System\Exits> python

```

```
>>> from testexit_os import outahere
>>> try:
...     outahere()
... finally:
...     print('Cleanup')
...
Bye os world                                # ditto
```

Shell Command Exit Status Codes

Both the `sys` and `os` exit calls we just met accept an argument that denotes the exit status code of the process (it's optional in the `sys` call but required by `os`). After exit, this code may be interrogated in shells and by programs that ran the script as a child process. On Linux, for example, we ask for the `$status` shell variable's value in order to fetch the last program's exit status; by convention, a nonzero status generally indicates that some sort of problem occurred:

```
[mark@linux]$ python testexit_sys.py
Bye sys world
[mark@linux]$ echo $status
42
[mark@linux]$ python testexit_os.py
Bye os world
[mark@linux]$ echo $status
99
```

In a chain of command-line programs, exit statuses could be checked along the way as a simple form of cross-program communication.

We can also grab hold of the exit status of a program run by another script. For instance, as introduced in Chapters 2 and 3, when launching shell commands, exit status is provided as:

- The return value of an `os.system` call
- The return value of the `close` method of an `os.popen` object (for historical reasons, `None` is returned if the exit status was 0, which means no error occurred)
- A variety of interfaces in the `subprocess` module (e.g., the `call` function's return value, a `Popen` object's `returnvalue` attribute and `wait` method result)

In addition, when running programs by forking processes, the exit status is available through the `os.wait` and `os.waitpid` calls in a parent process.

Exit status with `os.system` and `os.popen`

Let's look at the case of the shell commands first—the following, run on Linux, spawns Example 5-15, and Example 5-16 reads the output streams through pipes and fetches their exit status codes:

```
[mark@linux]$ python
>>> import os
>>> pipe = os.popen('python testexit_sys.py')
```

```

>>> pipe.read()
'Bye sys world\012'
>>> stat = pipe.close()           # returns exit code
>>> stat
10752
>>> hex(stat)
'0x2a00'
>>> stat >> 8                  # extract status from bitmask on Unix-likes
42

>>> pipe = os.popen('python testexit_os.py')
>>> stat = pipe.close()
>>> stat, stat >> 8
(25344, 99)

```

This code works the same under Cygwin Python on Windows. When using `os.popen` on such Unix-like platforms, for reasons we won't go into here, the exit status is actually packed into specific bit positions of the return value; it's really there, but we need to shift the result right by eight bits to see it. Commands run with `os.system` send their statuses back directly through the Python library call:

```

>>> stat = os.system('python testexit_sys.py')
Bye sys world
>>> stat, stat >> 8
(10752, 42)

>>> stat = os.system('python testexit_os.py')
Bye os world
>>> stat, stat >> 8
(25344, 99)

```

All of this code works under the standard version of Python for Windows, too, though exit status is not encoded in a bit mask (test `sys.platform` if your code must handle both formats):

```

C:\...\PP4E\System\Exits> python
>>> os.system('python testexit_sys.py')
Bye sys world
42
>>> os.system('python testexit_os.py')
Bye os world
99

>>> pipe = os.popen('python testexit_sys.py')
>>> pipe.read()
'Bye sys world\n'
>>> pipe.close()
42
>>>
>>> os.popen('python testexit_os.py').close()
99

```

Output stream buffering: A first look

Notice that the last test in the preceding code didn't attempt to read the command's output pipe. If we do, we may have to run the target script in *unbuffered* mode with the `-u` Python command-line flag or change the script to flush its output manually with `sys.stdout.flush`. Otherwise, the text printed to the standard output stream might not be flushed from its buffer when `os._exit` is called in this case for immediate shutdown. By default, standard output is fully buffered when connected to a pipe like this; it's only line-buffered when connected to a terminal:

```
>>> pipe = os.popen('python testexit_os.py')
>>> pipe.read()                                     # streams not flushed on exit
''

>>> pipe = os.popen('python -u testexit_os.py')      # force unbuffered streams
>>> pipe.read()
'Bye os world\n'
```

Confusingly, you can pass mode and buffering argument to specify line buffering in both `os.popen` and `subprocess.Popen`, but this won't help here—arguments passed to these tools pertain to the calling process's input end of the pipe, *not* to the spawned program's output stream:

```
>>> pipe = os.popen('python testexit_os.py', 'r', 1)  # line buffered only
>>> pipe.read()                                     # but my pipe, not program's!
''

>>> from subprocess import Popen, PIPE
>>> pipe = Popen('python testexit_os.py', bufsize=1, stdout=PIPE)  # for my pipe
>>> pipe.stdout.read()                                # doesn't help
b''
```

Really, buffering mode arguments in these tools pertain to output the caller writes to a command's standard input stream, not to output read from that command.

If required, the spawned script itself can also manually flush its output buffers periodically or before forced exits. More on buffering when we discuss the potential for *deadlocks* later in this chapter, and again in Chapters 10 and 12 where we'll see how it applies to sockets. Since we brought up `subprocess`, though, let's turn to its exit tools next.

Exit status with subprocess

The alternative `subprocess` module offers exit status in a variety of ways, as we saw in Chapters 2 and 3 (a `None` value in `returncode` indicates that the spawned program has not yet terminated):

```
C:\...\PP4E\System\Exits> python
>>> from subprocess import Popen, PIPE, call
>>> pipe = Popen('python testexit_sys.py', stdout=PIPE)
>>> pipe.stdout.read()
b'Bye sys world\r\n'
```

```

>>> pipe.wait()
42

>>> call('python testexit_sys.py')
Bye sys world
42

>>> pipe = Popen('python testexit_sys.py', stdout=PIPE)
>>> pipe.communicate()
(b'Bye sys world\r\n', None)
>>> pipe.returncode
42

```

The `subprocess` module works the same on Unix-like platforms like Cygwin, but unlike `os.popen`, the exit status is not encoded, and so it matches the Windows result (note that `shell=True` is needed to run this as is on Cygwin and Unix-like platforms, as we learned in [Chapter 2](#); on Windows this argument is required only to run commands built into the shell, like `dir`):

```

[C:\...\PP4E\System\Exits]$ python
>>> from subprocess import Popen, PIPE, call
>>> pipe = Popen('python testexit_sys.py', stdout=PIPE, shell=True)
>>> pipe.stdout.read()
b'Bye sys world\n'
>>> pipe.wait()
42

>>> call('python testexit_sys.py', shell=True)
Bye sys world
42

```

Process Exit Status and Shared State

Now, to learn how to obtain the exit status from forked processes, let's write a simple forking program: the script in [Example 5-17](#) forks child processes and prints child process exit statuses returned by `os.wait` calls in the parent until a “q” is typed at the console.

Example 5-17. PP4E\System\Exits\testexit_fork.py

```

"""
fork child processes to watch exit status with os.wait; fork works on Unix
and Cygwin but not standard Windows Python 3.1; note: spawned threads share
globals, but each forked process has its own copy of them (forks share file
descriptors)--exitstat is always the same here but will vary if for threads;
"""

import os
exitstat = 0

def child():                      # could os.exit a script here
    global exitstat                # change this process's global
    exitstat += 1                  # exit status to parent's wait

```

```

print('Hello from child', os.getpid(), exitstat)
os._exit(exitstat)
print('never reached')

def parent():
    while True:
        newpid = os.fork()           # start a new copy of process
        if newpid == 0:              # if in copy, run child logic
            child()                 # loop until 'q' console input
        else:
            pid, status = os.wait()
            print('Parent got', pid, status, (status >> 8))
            if input() == 'q': break

if __name__ == '__main__': parent()

```

Running this program on Linux, Unix, or Cygwin (remember, `fork` still doesn't work on standard Windows Python as I write the fourth edition of this book) produces the following sort of results:

```

[C:\...\PP4E\System\Exits]$ python testexit_fork.py
Hello from child 5828 1
Parent got 5828 256 1

Hello from child 9540 1
Parent got 9540 256 1

Hello from child 3152 1
Parent got 3152 256 1
q

```

If you study this output closely, you'll notice that the exit status (the last number printed) is always the same—the number 1. Because forked processes begin life as *copies* of the process that created them, they also have copies of global memory. Because of that, each forked child gets and changes its own `exitstat` global variable without changing any other process's copy of this variable. At the same time, forked processes copy and thus share file descriptors, which is why prints go to the same place.

Thread Exits and Shared State

In contrast, threads run in parallel within the *same* process and share global memory. Each thread in [Example 5-18](#) changes the single shared global variable, `exitstat`.

Example 5-18. PP4E\System\Exits\testexit_thread.py

```

"""
spawn threads to watch shared global memory change; threads normally exit
when the function they run returns, but _thread.exit() can be called to
exit calling thread; _thread.exit is the same as sys.exit and raising
SystemExit; threads communicate with possibly locked global vars; caveat:
may need to make print/input calls atomic on some platforms--shared stdout;
"""

```

```

import _thread as thread
exitstat = 0

def child():
    global exitstat
    exitstat += 1
    threadid = thread.get_ident()           # process global names
                                                # shared by all threads
    print('Hello from child', threadid, exitstat)
    thread.exit()
    print('never reached')

def parent():
    while True:
        thread.start_new_thread(child, ())
        if input() == 'q': break

if __name__ == '__main__': parent()

```

The following shows this script in action on Windows; unlike forks, threads run in the standard version of Python on Windows, too. Thread identifiers created by Python differ each time—they are arbitrary but unique among all currently active threads and so may be used as dictionary keys to keep per-thread information (a thread’s id may be reused after it exits on some platforms):

```

C:\...\PP4E\System\Exits> python testexit_thread.py
Hello from child 4908 1

Hello from child 4860 2

Hello from child 2752 3

Hello from child 8964 4
q

```

Notice how the value of this script’s global `exitstat` is changed by each thread, because threads share global memory within the process. In fact, this is often how threads communicate in general. Rather than exit status codes, threads assign module-level globals or change shared mutable objects in-place to signal conditions, and they use thread module locks and queues to synchronize access to shared items if needed. This script might need to synchronize, too, if it ever does something more realistic—for global counter changes, but even `print` and `input` may have to be synchronized if they overlap stream access badly on some platforms. For this simple demo, we forego locks by assuming threads won’t mix their operations oddly.

As we’ve learned, a thread normally exits silently when the function it runs returns, and the function return value is ignored. Optionally, the `_thread.exit` function can be called to terminate the calling thread explicitly and silently. This call works almost exactly like `sys.exit` (but takes no return status argument), and it works by raising a `SystemExit` exception in the calling thread. Because of that, a thread can also prematurely end by calling `sys.exit` or by directly raising `SystemExit`. Be sure not to call `os._exit` within a thread function, though—doing so can have odd results (the last time

I tried, it hung the entire process on my Linux system and killed every thread in the process on Windows!).

The alternative `threading` module for threads has no method equivalent to `_thread.exit()`, but since all that the latter does is raise a system-exit exception, doing the same in `threading` has the same effect—the thread exits immediately and silently, as in the following sort of code (see `testexit-threading.py` in the example tree for this code):

```
import threading, sys, time

def action():
    sys.exit()                      # or raise SystemExit()
    print('not reached')

threading.Thread(target=action).start()
time.sleep(2)
print('Main exit')
```

On a related note, keep in mind that threads and processes have default lifespan models, which we explored earlier. By way of review, when child threads are still running, the two thread modules' behavior differs—programs on most platforms exit when the parent thread does under `_thread`, but not normally under `threading` unless children are made daemons. When using processes, children normally outlive their parent. This different process behavior makes sense if you remember that threads are in-process function calls, but processes are more independent and autonomous.

When used well, exit status can be used to implement error detection and simple communication protocols in systems composed of command-line scripts. But having said that, I should underscore that most scripts do simply fall off the end of the source to exit, and most thread functions simply return; explicit exit calls are generally employed for exceptional conditions and in limited contexts only. More typically, programs communicate with richer tools than integer exit codes; the next section shows how.

Interprocess Communication

As we saw earlier, when scripts spawn *threads*—tasks that run in parallel within the program—they can naturally communicate by changing and inspecting names and objects in shared global memory. This includes both accessible variables and attributes, as well as referenced mutable objects. As we also saw, some care must be taken to use locks to synchronize access to shared items that can be updated concurrently. Still, threads offer a fairly straightforward communication model, and the `queue` module can make this nearly automatic for many programs.

Things aren't quite as simple when scripts start child processes and independent programs that do not share memory in general. If we limit the kinds of communications that can happen between programs, many options are available, most of which we've

already seen in this and the prior chapters. For example, the following simple mechanisms can all be interpreted as cross-program communication devices:

- Simple files
- Command-line arguments
- Program exit status codes
- Shell environment variables
- Standard stream redirections
- Stream pipes managed by `os.popen` and `subprocess`

For instance, sending command-line options and writing to input streams lets us pass in program execution parameters; reading program output streams and exit codes gives us a way to grab a result. Because shell environment variable settings are inherited by spawned programs, they provide another way to pass context in. And pipes made by `os.popen` or `subprocess` allow even more dynamic communication. Data can be sent between programs at arbitrary times, not only at program start and exit.

Beyond this set, there are other tools in the Python library for performing Inter-Process Communication (IPC). This includes sockets, shared memory, signals, anonymous and named pipes, and more. Some vary in portability, and all vary in complexity and utility. For instance:

- *Signals* allow programs to send simple notification events to other programs.
- *Anonymous pipes* allow threads and related processes that share file descriptors to pass data, but generally rely on the Unix-like forking model for processes, which is not universally portable.
- *Named pipes* are mapped to the system's filesystem—they allow completely unrelated programs to converse, but are not available in Python on all platforms.
- *Sockets* map to system-wide port numbers—they similarly let us transfer data between arbitrary programs running on the same computer, but also between programs located on remote networked machines, and offer a more portable option.

While some of these can be used as communication devices by threads, too, their full power becomes more evident when leveraged by separate processes which do not share memory at large.

In this section, we explore directly managed pipes (both anonymous and named), as well as signals. We also take a first look at sockets here, but largely as a preview; sockets can be used for IPC on a single machine, but because the larger socket story also involves their role in networking, we'll save most of their details until the Internet part of this book.

Other IPC tools are available to Python programmers (e.g., shared memory as provided by the `mmap` module) but are not covered here for lack of space; search the Python

manuals and website for more details on other IPC schemes if you’re looking for something more specific.

After this section, we’ll also study the `multiprocessing` module, which offers additional and portable IPC options as part of its general process-launching API, including shared memory, and pipes and queues of arbitrary pickled Python objects. For now, let’s study traditional approaches first.

Anonymous Pipes

Pipes, a cross-program communication device, are implemented by your operating system and made available in the Python standard library. Pipes are unidirectional channels that work something like a shared memory buffer, but with an interface resembling a simple file on each of two ends. In typical use, one program writes data on one end of the pipe, and another reads that data on the other end. Each program sees only its end of the pipes and processes it using normal Python file calls.

Pipes are much more within the operating system, though. For instance, calls to read a pipe will normally *block* the caller until data becomes available (i.e., is sent by the program on the other end) instead of returning an end-of-file indicator. Moreover, read calls on a pipe always return the oldest data written to the pipe, resulting in a *first-in-first-out* model—the first data written is the first to be read. Because of such properties, pipes are also a way to synchronize the execution of independent programs.

Pipes come in two flavors—*anonymous* and *named*. Named pipes (often called fifos) are represented by a file on your computer. Because named pipes are really external files, the communicating processes need not be related at all; in fact, they can be independently started programs.

By contrast, anonymous pipes exist only within processes and are typically used in conjunction with process *forks* as a way to link parent and spawned child processes within an application. Parent and child converse over shared pipe file descriptors, which are inherited by spawned processes. Because threads run in the same process and share all global memory in general, anonymous pipes apply to them as well.

Anonymous pipe basics

Since they are more traditional, let’s start with a look at anonymous pipes. To illustrate, the script in [Example 5-19](#) uses the `os.fork` call to make a copy of the calling process as usual (we met forks earlier in this chapter). After forking, the original parent process and its child copy speak through the two ends of a pipe created with `os.pipe` prior to the fork. The `os.pipe` call returns a tuple of two *file descriptors*—the low-level file identifiers we met in [Chapter 4](#)—representing the input and output sides of the pipe. Because forked child processes get *copies* of their parents’ file descriptors, writing to the pipe’s output descriptor in the child sends data back to the parent on the pipe created before the child was spawned.

Example 5-19. PP4E\System\Processes\pipe1.py

```
import os, time

def child(pipeout):
    zzz = 0
    while True:
        time.sleep(zzz)
        msg = ('Spam %03d' % zzz).encode()
        os.write(pipeout, msg)
        zzz = (zzz+1) % 5

def parent():
    pipein, pipeout = os.pipe()
    if os.fork() == 0:
        child(pipeout)
    else:
        while True:
            line = os.read(pipein, 32)      # blocks until data sent
            print('Parent %d got [%s] at %s' % (os.getpid(), line, time.time()))

parent()
```

If you run this program on Linux, Cygwin, or another Unix-like platform (`pipe` is available on standard Windows Python, but `fork` is not), the parent process waits for the child to send data on the pipe each time it calls `os.read`. It's almost as if the child and parent act as client and server here—the parent starts the child and waits for it to initiate communication.[#] To simulate differing task durations, the child keeps the parent waiting one second longer between messages with `time.sleep` calls, until the delay has reached four seconds. When the `zzz` delay counter hits 005, it rolls back down to 000 and starts again:

```
[C:\...\PP4E\System\Processes]$ python pipe1.py
Parent 6716 got [b'Spam 000'] at 1267996104.53
Parent 6716 got [b'Spam 001'] at 1267996105.54
Parent 6716 got [b'Spam 002'] at 1267996107.55
Parent 6716 got [b'Spam 003'] at 1267996110.56
Parent 6716 got [b'Spam 004'] at 1267996114.57
Parent 6716 got [b'Spam 000'] at 1267996114.57
Parent 6716 got [b'Spam 001'] at 1267996115.59
Parent 6716 got [b'Spam 002'] at 1267996117.6
Parent 6716 got [b'Spam 003'] at 1267996120.61
Parent 6716 got [b'Spam 004'] at 1267996124.62
Parent 6716 got [b'Spam 000'] at 1267996124.62
```

[#]We will clarify the notions of “client” and “server” in the Internet programming part of this book. There, we'll communicate with sockets (which we'll see later in this chapter) are roughly like bidirectional pipes for programs running both across networks and on the same machine), but the overall conversation model is similar. Named pipes (fifos), described ahead, are also a better match to the client/server model because they can be accessed by arbitrary, unrelated processes (no forks are required). But as we'll see, the socket port model is generally used by most Internet scripting protocols—email, for instance, is mostly just formatted strings shipped over sockets between programs on standard port numbers reserved for the email protocol.

```
Parent 6716 got [b'Spam 001'] at 1267996125.63
...etc.: Ctrl-C to exit...
```

Notice how the parent received a `bytes` string through the pipe. Raw pipes normally deal in binary byte strings when their descriptors are used directly this way with the descriptor-based file tools we met in [Chapter 4](#) (as we saw there, descriptor read and write tools in `os` always return and expect byte strings). That's why we also have to manually encode to `bytes` when writing in the child—the string formatting operation is not available on `bytes`. As the next section shows, it's also possible to wrap a pipe descriptor in a text-mode file object, much as we did in the file examples in [Chapter 4](#), but that object simply performs encoding and decoding automatically on transfers; it's still `bytes` in the pipe.

Wrapping pipe descriptors in file objects

If you look closely at the preceding output, you'll see that when the child's delay counter hits 004, the parent ends up reading two messages from the pipe at the same time; the child wrote two distinct messages, but on some platforms or configurations (other than that used here) they might be interleaved or processed close enough in time to be fetched as a single unit by the parent. Really, the parent blindly asks to read, at most, 32 bytes each time, but it gets back whatever text is available in the pipe, when it becomes available.

To distinguish messages better, we can mandate a separator character in the pipe. An end-of-line makes this easy, because we can wrap the pipe descriptor in a file object with `os.fdopen` and rely on the file object's `readline` method to scan up through the next `\n` separator in the pipe. This also lets us leverage the more powerful tools of the text-mode file object we met in [Chapter 4](#). [Example 5-20](#) implements this scheme for the parent's end of the pipe.

Example 5-20. PP4E\System\Processes\pipe2.py

```
# same as pipe1.py, but wrap pipe input in stdio file object
# to read by line, and close unused pipe fds in both processes

import os, time

def child(pipeout):
    zzz = 0
    while True:
        time.sleep(zzz)                      # make parent wait
        msg = ('Spam %03d\n' % zzz).encode()   # pipes are binary in 3.X
        os.write(pipeout, msg)                 # send to parent
        zzz = (zzz+1) % 5                     # roll to 0 at 5

def parent():
    pipein, pipeout = os.pipe()            # make 2-ended pipe
    if os.fork() == 0:                    # in child, write to pipe
        os.close(pipein)                 # close input side here
        child(pipeout)
```

```

else:                                # in parent, listen to pipe
    os.close(pipeout)                 # close output side here
    pipein = os.fdopen(pipein)         # make text mode input file object
    while True:
        line = pipein.readline()[:-1]   # blocks until data sent
        print('Parent %d got [%s] at %s' % (os.getpid(), line, time.time()))

parent()

```

This version has also been augmented to *close* the unused end of the pipe in each process (e.g., after the fork, the parent process closes its copy of the output side of the pipe written by the child); programs should close unused pipe ends in general. Running with this new version reliably returns a single child message to the parent each time it reads from the pipe, because they are separated with markers when written:

```
[C:\...\PP4E\System\Processes]$ python pipe2.py
Parent 8204 got [Spam 000] at 1267997789.33
Parent 8204 got [Spam 001] at 1267997790.03
Parent 8204 got [Spam 002] at 1267997792.05
Parent 8204 got [Spam 003] at 1267997795.06
Parent 8204 got [Spam 004] at 1267997799.07
Parent 8204 got [Spam 000] at 1267997799.07
Parent 8204 got [Spam 001] at 1267997800.08
Parent 8204 got [Spam 002] at 1267997802.09
Parent 8204 got [Spam 003] at 1267997805.1
Parent 8204 got [Spam 004] at 1267997809.11
Parent 8204 got [Spam 000] at 1267997809.11
Parent 8204 got [Spam 001] at 1267997810.13
...etc.: Ctrl-C to exit...
```

Notice that this version's reads also return a text data `str` object now, per the default `r` text mode for `os.fdopen`. As mentioned, pipes normally deal in binary byte strings when their descriptors are used directly with `os` file tools, but wrapping in text-mode files allows us to use `str` strings to represent text data instead of `bytes`. In this example, bytes are decoded to `str` when read by the parent; using `os.fdopen` and text mode in the child would allow us to avoid its manual encoding call, but the file object would encode the `str` data anyhow (though the encoding is trivial for ASCII bytes like those used here). As for simple files, the best mode for processing pipe data in is determined by its nature.

Anonymous pipes and threads

Although the `os.fork` call required by the prior section's examples isn't available on standard Windows Python, `os.pipe` is. Because threads all run in the same process and share file descriptors (and global memory in general), this makes anonymous pipes usable as a communication and synchronization device for threads, too. This is an arguably lower-level mechanism than queues or shared names and objects, but it provides an additional IPC option for threads. [Example 5-21](#), for instance, demonstrates the same type of pipe-based communication occurring between threads instead of processes.

Example 5-21. PP4E\System\Processes\pipe-thread.py

```
# anonymous pipes and threads, not processes; this version works on Windows

import os, time, threading

def child(pipeout):
    zzz = 0
    while True:
        time.sleep(zzz)                                # make parent wait
        msg = ('Spam %03d' % zzz).encode()             # pipes are binary bytes
        os.write(pipeout, msg)                          # send to parent
        zzz = (zzz+1) % 5                             # goto 0 after 4

def parent(pipein):
    while True:
        line = os.read(pipein, 32)                   # blocks until data sent
        print('Parent %d got [%s] at %s' % (os.getpid(), line, time.time()))

pipein, pipeout = os.pipe()
threading.Thread(target=child, args=(pipeout,)).start()
parent(pipein)
```

Since threads work on standard Windows Python, this script does too. The output is similar here, but the speakers are in-process threads, not processes (note that because of its simple-minded infinite loops, at least one of its threads may not die on a Ctrl-C—on Windows you may need to use Task Manager to kill the *python.exe* process running this script or close its window to exit):

```
C:\...\PP4E\System\Processes> pipe-thread.py
Parent 8876 got [b'Spam 000'] at 1268579215.71
Parent 8876 got [b'Spam 001'] at 1268579216.73
Parent 8876 got [b'Spam 002'] at 1268579218.74
Parent 8876 got [b'Spam 003'] at 1268579221.75
Parent 8876 got [b'Spam 004'] at 1268579225.76
Parent 8876 got [b'Spam 000'] at 1268579225.76
Parent 8876 got [b'Spam 001'] at 1268579226.77
Parent 8876 got [b'Spam 002'] at 1268579228.79
...etc.: Ctrl-C or Task Manager to exit...
```

Bidirectional IPC with anonymous pipes

Pipes normally let data flow in only one direction—one side is input, one is output. What if you need your programs to talk back and forth, though? For example, one program might send another a request for information and then wait for that information to be sent back. A single pipe can't generally handle such bidirectional conversations, but two pipes can. One pipe can be used to pass requests to a program and another can be used to ship replies back to the requestor.

This really does have real-world applications. For instance, I once added a GUI interface to a command-line debugger for a C-like programming language by connecting two processes with pipes this way. The GUI ran as a separate process that constructed and

sent commands to the non-GUI debugger's input stream pipe and parsed the results that showed up in the debugger's output stream pipe. In effect, the GUI acted like a programmer typing commands at a keyboard and a client to the debugger server. More generally, by spawning command-line programs with streams attached by pipes, systems can add new interfaces to legacy programs. In fact, we'll see a simple example of this sort of GUI program structure in [Chapter 10](#).

The module in [Example 5-22](#) demonstrates one way to apply this idea to link the input and output streams of two programs. Its `spawn` function forks a new child program and connects the input and output streams of the parent to the output and input streams of the child. That is:

- When the parent reads from its standard input, it is reading text sent to the child's standard output.
- When the parent writes to its standard output, it is sending data to the child's standard input.

The net effect is that the two independent programs communicate by speaking over their standard streams.

Example 5-22. PP4E\System\Processes\pipes.py

```
"""
spawn a child process/program, connect my stdin/stdout to child process's
stdout/stdin--my reads and writes map to output and input streams of the
spawned program; much like tying together streams with subprocess module;
"""

import os, sys

def spawn(prog, *args):
    stdInFd = sys.stdin.fileno()                      # pass progname, cmdline args
    stdOutFd = sys.stdout.fileno()                     # get descriptors for streams
                                                       # normally stdin=0, stdout=1

    parentStdIn, childStdout = os.pipe()               # make two IPC pipe channels
    childStdIn, parentStdout = os.pipe()              # pipe returns (inputfd, outoutfd)
                                                       # make a copy of this process
    pid = os.fork()
    if pid:
        os.close(childStdout)                         # in parent process after fork:
        os.close(childStdIn)                          # close child ends in parent
        os.dup2(parentStdIn, stdInFd)                 # my sys.stdin copy = pipe1[0]
        os.dup2(parentStdout, stdOutFd)               # my sys.stdout copy = pipe2[1]
    else:
        os.close(parentStdIn)                         # in child process after fork:
        os.close(parentStdout)                        # close parent ends in child
        os.dup2(childStdIn, stdInFd)                  # my sys.stdin copy = pipe2[0]
        os.dup2(childStdout, stdOutFd)                # my sys.stdout copy = pipe1[1]
    args = (prog,) + args
    os.execvp(prog, args)                            # new program in this process
    assert False, 'execvp failed!'

if __name__ == '__main__':
```

```

mypad = os.getpid()
spawn('python', 'pipes-testchild.py', 'spam')      # fork child program

print('Hello 1 from parent', mypid)                # to child's stdin
sys.stdout.flush()                                # subvert stdio buffering
reply = input()                                    # from child's stdout
sys.stderr.write('Parent got: "%s"\n' % reply)     # stderr not tied to pipe!

print('Hello 2 from parent', mypid)
sys.stdout.flush()
reply = sys.stdin.readline()
sys.stderr.write('Parent got: "%s"\n' % reply[:-1])

```

The `spawn` function in this module does not work on standard Windows Python (remember that `fork` isn't yet available there today). In fact, most of the calls in this module map straight to Unix system calls (and may be arbitrarily terrifying at first glance to non-Unix developers!). We've already met some of these (e.g., `os.fork`), but much of this code depends on Unix concepts we don't have time to address well in this text. But in simple terms, here is a brief summary of the system calls demonstrated in this code:

`os.fork`

Copies the calling process as usual and returns the child's process ID in the parent process only.

`os.execvp`

Overlays a new program in the calling process; it's just like the `os.execlp` used earlier but takes a *tuple* or *list* of command-line argument strings (collected with the `*args` form in the function header).

`os.pipe`

Returns a tuple of file descriptors representing the input and output ends of a pipe, as in earlier examples.

`os.close(fd)`

Closes the descriptor-based file `fd`.

`os.dup2(fd1, fd2)`

Copies all system information associated with the file named by the file descriptor `fd1` to the file named by `fd2`.

In terms of connecting standard streams, `os.dup2` is the real nitty-gritty here. For example, the call `os.dup2(parentStdin, stdInFd)` essentially assigns the parent process's `stdin` file to the input end of one of the two pipes created; all `stdin` reads will henceforth come from the pipe. By connecting the other end of this pipe to the child process's copy of the `stdout` stream file with `os.dup2(childStdout, stdOutFd)`, text written by the child to its `stdout` winds up being routed through the pipe to the parent's `stdin` stream. The effect is reminiscent of the way we tied together streams with the `subprocess` module in [Chapter 3](#), but this script is more low-level and less portable.

To test this utility, the self-test code at the end of the file spawns the program shown in [Example 5-23](#) in a child process and reads and writes standard streams to converse with it over two pipes.

Example 5-23. PP4E\System\Processes\pipes-testchild.py

```
import os, time, sys
mypid      = os.getpid()
parentpid  = os.getppid()
sys.stderr.write('Child %d of %d got arg: "%s"\n' %
                  (mypid, parentpid, sys.argv[1]))
for i in range(2):
    time.sleep(3)          # make parent process wait by sleeping here
    recv = input()           # stdin tied to pipe: comes from parent's stdout
    time.sleep(3)
    send = 'Child %d got: [%s]' % (mypid, recv)
    print(send)             # stdout tied to pipe: goes to parent's stdin
    sys.stdout.flush()       # make sure it's sent now or else process blocks
```

The following is our test in action on Cygwin (it's similar other Unix-like platforms like Linux); its output is not incredibly impressive to read, but it represents two programs running independently and shipping data back and forth through a pipe device managed by the operating system. This is even more like a client/server model (if you imagine the child as the server, responding to requests sent from the parent). The text in square brackets in this output went from the parent process to the child and back to the parent again, all through pipes connected to standard streams:

```
[C:\...\PP4E\System\Processes]$ python pipes.py
Child 9228 of 9096 got arg: "spam"
Parent got: "Child 9228 got: [Hello 1 from parent 9096]"
Parent got: "Child 9228 got: [Hello 2 from parent 9096]"
```

Output stream buffering revisited: Deadlocks and flushes

The two processes of the prior section's example engage in a simple dialog, but it's already enough to illustrate some of the dangers lurking in cross-program communications. First of all, notice that both programs need to write to `stderr` to display a message; their `stdout` streams are tied to the other program's input stream. Because processes share file descriptors, `stderr` is the same in both parent and child, so status messages show up in the same place.

More subtly, note that both parent and child call `sys.stdout.flush` after they print text to the output stream. Input requests on pipes normally block the caller if no data is available, but it seems that this shouldn't be a problem in our example because there are as many writes as there are reads on the other side of the pipe. By default, though, `sys.stdout` is *buffered* in this context, so the printed text may not actually be transmitted until some time in the future (when the output buffers fill up). In fact, if the flush calls are not made, both processes may get stuck on some platforms waiting for input from the other—input that is sitting in a buffer and is never flushed out over the pipe. They

wind up in a *deadlock* state, both blocked on `input` calls waiting for events that never occur.

Technically, by default `stdout` is just *line-buffered* when connected to a terminal, but it is *fully buffered* when connected to other devices such as files, sockets, and the pipes used here. This is why you see a script's printed text in a shell window immediately as it is produced, but not until the process exits or its buffer fills when its output stream is connected to something else.

This output buffering is really a function of the system libraries used to access pipes, not of the pipes themselves (pipes do queue up output data, but they never hide it from readers!). In fact, it appears to occur in this example only because we copy the pipe's information over to `sys.stdout`, a built-in file object that uses stream buffering by default. However, such anomalies can also occur when using other cross-process tools.

In general terms, if your programs engage in a two-way dialog like this, there are a variety of ways to avoid buffering-related deadlock problems:

- *Flushes*: As demonstrated in Examples 5-22 and 5-23, manually flushing output pipe streams by calling the file object `flush` method is an easy way to force buffers to be cleared. Use `sys.stdout.flush` for the output stream used by `print`.
- *Arguments*: As introduced earlier in this chapter, the `-u` Python command-line flag turns off full buffering for the `sys.stdout` stream in Python programs. Setting your `PYTHONUNBUFFERED` environment variable to a nonempty value is equivalent to passing this flag but applies to every program run.
- *Open modes*: It's possible to use pipes themselves in unbuffered mode. Either use low-level `os` module calls to read and write pipe descriptors directly, or pass a buffer size argument of `0` (for *unbuffered*) or `1` (for *line-buffered*) to `os.fdopen` to disable buffering in the file object used to wrap the descriptor. You can use `open` arguments the same way to control buffering for output to *fifo files* (described in the next section). Note that in Python 3.X, fully unbuffered mode is allowed only for binary mode files, not text.
- *Command pipes*: As mentioned earlier in this chapter, you can similarly specify buffering mode arguments for command-line pipes when they are created by `os.popen` and `subprocess.Popen`, but this pertains to the caller's end of the pipe, not those of the spawned program. Hence it cannot prevent delayed outputs from the latter, but can be used for text sent to another program's input pipe.
- *Sockets*: As we'll see later, the `socket.makefile` call accepts a similar buffering mode argument for sockets (described later in this chapter and book), but in Python 3.X this call requires buffering for text-mode access and appears to not support line-buffered mode (more on this on [Chapter 12](#)).
- *Tools*: For more complex tasks, we can also use higher-level tools that essentially fool a program into believing it is connected to a terminal. These address programs

not written in Python, for which neither manual flush calls nor `-u` are an option. See “[More on Stream Buffering: pty and Pexpect](#)” on page 233.

Thread can avoid blocking a main GUI, too, but really just delegate the problem (the spawned thread will still be deadlocked). Of the options listed, the first two—manual flushes and command-line arguments—are often the simplest solutions. In fact, because it is so useful, the second technique listed above merits a few more words. Try this: comment-out all the `sys.stdout.flush` calls in Examples 5-22 and 5-23 (the files `pipes.py` and `pipes-testchild.py`) and change the parent’s spawn call in `pipes.py` to this (i.e., add a `-u` command-line argument):

```
spawn('python', '-u', 'pipes-testchild.py', 'spam')
```

Then start the program with a command line like this: `python -u pipes.py`. It will work as it did with the manual `stdout` flush calls, because `stdout` will be operating in unbuffered mode in both parent and child.

We’ll revisit the effects of unbuffered output streams in [Chapter 10](#), where we’ll code a simple GUI that displays the output of a non-GUI program by reading it over both a nonblocking socket and a pipe in a thread. We’ll explore the topic again in more depth in [Chapter 12](#), where we will redirect standard streams to sockets in more general ways. Deadlock in general, though, is a bigger problem than we have space to address fully here. On the other hand, if you know enough that you want to do IPC in Python, you’re probably already a veteran of the deadlock wars.

Anonymous pipes allow related tasks to communicate but are not directly suited for independently launched programs. To allow the latter group to converse, we need to move on to the next section and explore devices that have broader visibility.

More on Stream Buffering: pty and Pexpect

On Unix-like platforms, you may also be able to use the Python `pty` standard library module to force another program’s standard output to be unbuffered, especially if it’s not a Python program and you cannot change its code.

Technically, default buffering for `stdout` in other programs is determined outside Python by whether the underlying file descriptor refers to a terminal. This occurs in the `stdio` file system library and cannot be controlled by the spawning program. In general, output to terminals is line buffered, and output to nonterminals (including files, pipes, and sockets) is fully buffered. This policy is used for efficiency. Files and streams created within a Python script follow the same defaults, but you can specify buffering policies in Python’s file creation tools.

The `pty` module essentially fools the spawned program into thinking it is connected to a terminal so that only one line is buffered for `stdout`. The net effect is that each newline flushes the prior line—typical of interactive programs, and what you need if you wish to grab each piece of the printed output as it is produced.

Note, however, that the `pty` module is not required for this role when spawning Python scripts with pipes: simply use the `-u` Python command-line flag, pass line-buffered mode

arguments to file creation tools, or manually call `sys.stdout.flush()` in the spawned program. The `pty` module is also not available on all Python platforms today (most notably, it runs on Cygwin but not the standard Windows Python).

The `Pexpect` package, a pure-Python equivalent of the Unix `expect` program, uses `pty` to provide additional functionality and to handle interactions that bypass standard streams (e.g., password inputs). See the Python library manual for more on `pty`, and search the Web for `Pexpect`.

Named Pipes (Fifos)

On some platforms, it is also possible to create a long-lived pipe that exists as a real named file in the filesystem. Such files are called named pipes (or, sometimes, *fifos*) because they behave just like the pipes created by the previous section’s programs. Because fifos are associated with a real file on your computer, though, they are external to any particular program—they do not rely on memory shared between tasks, and so they can be used as an IPC mechanism for threads, processes, and independently launched programs.

Once a named pipe file is created, clients open it by name and read and write data using normal file operations. Fifos are unidirectional streams. In typical operation, a server program reads data from the fifo, and one or more client programs write data to it. In addition, a set of two fifos can be used to implement bidirectional communication just as we did for anonymous pipes in the prior section.

Because fifos reside in the filesystem, they are longer-lived than in-process anonymous pipes and can be accessed by programs started independently. The unnamed, in-process pipe examples thus far depend on the fact that file descriptors (including pipes) are copied to child processes’ memory. That makes it difficult to use anonymous pipes to connect programs started independently. With fifos, pipes are accessed instead by a filename visible to all programs running on the computer, regardless of any parent/child process relationships. In fact, like normal files, fifos typically outlive the programs that access them. Unlike normal files, though, the operating system synchronizes fifo access, making them ideal for IPC.

Because of their distinctions, fifo pipes are better suited as general IPC mechanisms for independent client and server programs. For instance, a perpetually running server program may create and listen for requests on a fifo that can be accessed later by arbitrary clients not forked by the server. In a sense, fifos are an alternative to the socket port interface we’ll meet in the next section. Unlike sockets, though, fifos do not directly support remote network connections, are not available in standard Windows Python today, and are accessed using the standard file interface instead of the more unique socket port numbers and calls we’ll study later.

Named pipe basics

In Python, named pipe files are created with the `os.mkfifo` call, which is available today on Unix-like platforms, including Cygwin's Python on Windows, but is not currently available in standard Windows Python. This call creates only the external file, though; to send and receive data through a fifo, it must be opened and processed as if it were a standard file.

To illustrate, [Example 5-24](#) is a derivation of the `pipe2.py` script listed in [Example 5-20](#), but rewritten here to use fifos rather than anonymous pipes. Much like `pipe2.py`, this script opens the fifo using `os.open` in the child for low-level byte string access, but with the `open` built-in in the parent to treat the pipe as text; in general, either end may use either technique to treat the pipe's data as bytes or text.

Example 5-24. PP4E\System\Processes\pipefifo.py

```
"""
named pipes; os.mkfifo is not available on Windows (without Cygwin);
there is no reason to fork here, since fifo file pipes are external
to processes--shared fds in parent/child processes are irrelevant;
"""

import os, time, sys
fifoName = '/tmp/pipefifo'                                # must open same name

def child():
    pipeout = os.open(fifoName, os.O_WRONLY)      # open fifo pipe file as fd
    zzz = 0
    while True:
        time.sleep(zzz)
        msg = ('Spam %03d\n' % zzz).encode()      # binary as opened here
        os.write(pipeout, msg)
        zzz = (zzz+1) % 5

def parent():
    pipein = open(fifoName, 'r')                  # open fifo as text file object
    while True:
        line = pipein.readline()[:-1]            # blocks until data sent
        print('Parent %d got "%s" at %s' % (os.getpid(), line, time.time()))

if __name__ == '__main__':
    if not os.path.exists(fifoName):
        os.mkfifo(fifoName)                      # create a named pipe file
    if len(sys.argv) == 1:
        parent()                                # run as parent if no args
    else:
        child()
```

Because the fifo exists independently of both parent and child, there's no reason to fork here. The child may be started independently of the parent as long as it opens a fifo file by the same name. Here, for instance, on Cygwin the parent is started in one shell

window and then the child is started in another. Messages start appearing in the parent window only after the child is started and begins writing messages onto the fifo file:

```
[C:\...\PP4E\System\Processes] $ python pipefifo.py          # parent window
Parent 8324 got "Spam 000" at 1268003696.07
Parent 8324 got "Spam 001" at 1268003697.06
Parent 8324 got "Spam 002" at 1268003699.07
Parent 8324 got "Spam 003" at 1268003702.08
Parent 8324 got "Spam 004" at 1268003706.09
Parent 8324 got "Spam 000" at 1268003706.09
Parent 8324 got "Spam 001" at 1268003707.11
Parent 8324 got "Spam 002" at 1268003709.12
Parent 8324 got "Spam 003" at 1268003712.13
Parent 8324 got "Spam 004" at 1268003716.14
Parent 8324 got "Spam 000" at 1268003716.14
Parent 8324 got "Spam 001" at 1268003717.15
...etc: Ctrl-C to exit...

[C:\...\PP4E\System\Processes]$ file /tmp/pipefifo           # child window
/tmp/pipefifo: fifo (named pipe)

[C:\...\PP4E\System\Processes]$ python pipefifo.py -child
...Ctrl-C to exit...
```

Named pipe use cases

By mapping communication points to a file system entity accessible to all programs run on a machine, fifos can address a broad range of IPC goals on platforms where they are supported. For instance, although this section’s example runs independent programs, named pipes can also be used as an IPC device by both in-process threads and directly forked related processes, much as we saw for anonymous pipes earlier.

By also supporting unrelated programs, though, fifo files are more widely applicable to general client/server models. For example, named pipes can make the GUI and command-line debugger integration I described earlier for anonymous pipes even more flexible—by using fifo files to connect the GUI to the non-GUI debugger’s streams, the GUI could be started independently when needed.

Sockets provide similar functionality but also buy us both inherent network awareness and broader portability to Windows—as the next section explains.

Sockets: A First Look

Sockets, implemented by the Python `socket` module, are a more general IPC device than the pipes we’ve seen so far. Sockets let us transfer data between programs running on the same computer, as well as programs located on remote networked machines. When used as an IPC mechanism on the same machine, programs connect to sockets by a machine-global port number and transfer data. When used as a networking connection, programs provide both a machine name and port number to transfer data to a remotely-running program.

Socket basics

Although sockets are one of the most commonly used IPC tools, it's impossible to fully grasp their API without also seeing its role in networking. Because of that, we'll defer most of our socket coverage until we can explore their use in network scripting in [Chapter 12](#). This section provides a brief introduction and preview, so you can compare with the prior section's named pipes (a.k.a. fifos). In short:

- Like fifos, sockets are global across a machine; they do not require shared memory among threads or processes, and are thus applicable to independent programs.
- Unlike fifos, sockets are identified by port number, not filesystem path name; they employ a very different nonfile API, though they can be wrapped in a file-like object; and they are more portable: they work on nearly every Python platform, including standard Windows Python.

In addition, sockets support networking roles that go beyond both IPC and this chapter's scope. To illustrate the basics, though, [Example 5-25](#) launches a server and 5 clients in threads running in parallel on the same machine, to communicate over a socket—because all threads connect to the same port, the server consumes the data added by each of the clients.

Example 5-25. PP4E\System\Processes\socket_preview.py

```
"""
sockets for cross-task communication: start threads to communicate over sockets;
independent programs can too, because sockets are system-wide, much like fifos;
see the GUI and Internet parts of the book for more realistic socket use cases;
some socket servers may also need to talk to clients in threads or processes;
sockets pass byte strings, but can be pickled objects or encoded Unicode text;
caveat: prints in threads may need to be synchronized if their output overlaps;
"""

from socket import socket, AF_INET, SOCK_STREAM      # portable socket api

port = 50008                      # port number identifies socket on machine
host = 'localhost'                 # server and client run on same local machine here

def server():
    sock = socket(AF_INET, SOCK_STREAM)            # ip addresses tcp connection
    sock.bind(('', port))                         # bind to port on this machine
    sock.listen(5)                                # allow up to 5 pending clients
    while True:
        conn, addr = sock.accept()                # wait for client to connect
        data = conn.recv(1024)                     # read bytes data from this client
        reply = 'server got: [%s]' % data         # conn is a new connected socket
        conn.send(reply.encode())                  # send bytes reply back to client

def client(name):
    sock = socket(AF_INET, SOCK_STREAM)
    sock.connect((host, port))                  # connect to a socket port
    sock.send(name.encode())                   # send bytes data to listener
    reply = sock.recv(1024)                    # receive bytes data from listener
```

```

    sock.close()                                # up to 1024 bytes in message
    print('client got: [%s]' % reply)

if __name__ == '__main__':
    from threading import Thread
    sthread = Thread(target=server)
    sthread.daemon = True
    sthread.start()                           # don't wait for server thread
    for i in range(5):                      # do wait for children to exit
        Thread(target=client, args=(client%5 % i,)).start()

```

Study this script's code and comments to see how the socket objects' methods are used to transfer data. In a nutshell, with this type of socket the server accepts a client connection, which by default blocks until a client requests service, and returns a new socket connected to the client. Once connected, the client and server transfer byte strings by using send and receive calls instead of writes and reads, though as we'll see later in the book, sockets can be wrapped in file objects much as we did earlier for pipe descriptors. Also like pipe descriptors, unwrapped sockets deal in binary `bytes` strings, not text `str`; that's why string formatting results are manually encoded again here.

Here is this script's output on Windows:

```
C:\...\PP4E\System\Processes> socket_preview.py
client got: [b"server got: [b'client1']"]
client got: [b"server got: [b'client3']"]
client got: [b"server got: [b'client4']"]
client got: [b"server got: [b'client2']"]
client got: [b"server got: [b'client0']"]
```

This output isn't much to look at, but each line reflects data sent from client to server, and then back again: the server receives a bytes string from a connected client and echoes it back in a larger reply string. Because all threads run in parallel, the order in which the clients are served is random on this machine.

Sockets and independent programs

Although sockets work for threads, the shared memory model of threads often allows them to employ simpler communication devices such as shared names and objects and queues. Sockets tend to shine brighter when used for IPC by separate processes and independently launched programs. [Example 5-26](#), for instance, reuses the server and client functions of the prior example, but runs them in both processes and threads of independently launched programs.

Example 5-26. PP4E\System\Processes\socket-preview-progs.py

```
"""
same socket, but talk between independent programs too, not just threads;
server here runs in a process and serves both process and thread clients;
sockets are machine-global, much like fifos: don't require shared memory
"""


```

```
from socket_preview import server, client      # both use same port number
```

```

import sys, os
from threading import Thread

mode = int(sys.argv[1])
if mode == 1:                                # run server in this process
    server()
elif mode == 2:                                # run client in this process
    client('client:process=%s' % os.getpid())
else:                                         # run 5 client threads in process
    for i in range(5):
        Thread(target=client, args=('client:thread=%s' % i,)).start()

```

Let's run this script on Windows, too (again, this portability is a major advantage of sockets). First, start the server in a process as an independently launched program in its own window; this process runs perpetually waiting for clients to request connections (and as for our prior pipe example you may need to use Task Manager or a window close to kill the server process eventually):

```
C:\...\PP4E\System\Processes> socket-preview-progs.py 1
```

Now, in another window, run a few clients in both processes and thread, by launching them as independent programs—using 2 as the command-line argument runs a single client process, but 3 spawns five threads to converse with the server on parallel:

```
C:\...\PP4E\System\Processes> socket-preview-progs.py 2
client got: [b"server got: [b'client:process=7384']"]
```

```
C:\...\PP4E\System\Processes> socket-preview-progs.py 2
client got: [b"server got: [b'client:process=7604']"]
```

```
C:\...\PP4E\System\Processes> socket-preview-progs.py 3
client got: [b"server got: [b'client:thread=1']"]
client got: [b"server got: [b'client:thread=2']"]
client got: [b"server got: [b'client:thread=0']"]
client got: [b"server got: [b'client:thread=3']"]
client got: [b"server got: [b'client:thread=4']"]
```

```
C:\...\PP4E\System\Processes> socket-preview-progs.py 3
client got: [b"server got: [b'client:thread=3']"]
client got: [b"server got: [b'client:thread=1']"]
client got: [b"server got: [b'client:thread=2']"]
client got: [b"server got: [b'client:thread=4']"]
client got: [b"server got: [b'client:thread=0']"]
```

```
C:\...\PP4E\System\Processes> socket-preview-progs.py 2
client got: [b"server got: [b'client:process=6428']"]
```

Socket use cases

This section's examples illustrate the basic IPC role of sockets, but this only hints at their full utility. Despite their seemingly limited byte string nature, higher-order use cases for sockets are not difficult to imagine. With a little extra work, for instance:

- Arbitrary Python *objects* like lists and dictionaries (or at least copies of them) can be transferred over sockets, too, by shipping the serialized byte strings produced by Python’s `pickle` module introduced in [Chapter 1](#) and covered in full in [Chapter 17](#).
- As we’ll see in [Chapter 10](#), the printed output of a simple script can be *redirected* to a GUI window, by connecting the script’s output stream to a socket on which a GUI is listening in nonblocking mode.
- Programs that fetch arbitrary *text* off the Web might read it as byte strings over sockets, but manually decode it using encoding names embedded in content-type headers or tags in the data itself.
- In fact, *the entire Internet* can be seen as a socket use case—as we’ll see in [Chapter 12](#), at the bottom, email, FTP, and web pages are largely just formatted byte string messages shipped over sockets.

Plus any other context in which programs exchange data—sockets are a general, portable, and flexible tool. For instance, they would provide the same utility as fifos for the GUI/debugger example used earlier, but would also work in Python on Windows and would even allow the GUI to connect to a debugger running on a different computer altogether. As such, they are seen by many as a more powerful IPC tool.

Again, you should consider this section just a preview; because the grander socket story also entails networking concepts, we’ll defer a more in-depth look at the socket API until [Chapter 12](#). We’ll also see sockets again briefly in [Chapter 10](#) in the GUI stream redirection use case listed above, and we’ll explore a variety of additional socket use cases in the Internet part of this book. In [Part IV](#), for instance, we’ll use sockets to transfer entire files and write more robust socket servers that spawn threads or processes to converse with clients to avoid denying connections. For the purposes of this chapter, let’s move on to one last traditional IPC tool—the signal.

Signals

For lack of a better analogy, signals are a way to poke a stick at a process. Programs generate signals to trigger a handler for that signal in another process. The operating system pokes, too—some signals are generated on unusual system events and may kill the program if not handled. If this sounds a little like raising exceptions in Python, it should; signals are software-generated events and the cross-process analog of exceptions. Unlike exceptions, though, signals are identified by number, are not stacked, and are really an asynchronous event mechanism outside the scope of the Python interpreter controlled by the operating system.

In order to make signals available to scripts, Python provides a `signal` module that allows Python programs to register Python functions as handlers for signal events. This module is available on both Unix-like platforms and Windows (though the Windows version may define fewer kinds of signals to be caught). To illustrate the basic signal

interface, the script in [Example 5-27](#) installs a Python handler function for the signal number passed in as a command-line argument.

Example 5-27. PP4E\System\Processes\signal1.py

```
"""
catch signals in Python; pass signal number N as a command-line arg,
use a "kill -N pid" shell command to send this process a signal; most
signal handlers restored by Python after caught (see network scripting
chapter for SIGCHLD details); on Windows, signal module is available,
but it defines only a few signal types there, and os.kill is missing;
"""

import sys, signal, time
def now(): return time.ctime(time.time())      # current time string

def onSignal(signum, stackframe):             # python signal handler
    print('Got signal', signum, 'at', now())   # most handlers stay in effect

signum = int(sys.argv[1])
signal.signal(signum, onSignal)               # install signal handler
while True: signal.pause()                   # wait for signals (or: pass)
```

There are only two `signal` module calls at work here:

`signal.signal`

Takes a signal number and function object and installs that function to handle that signal number when it is raised. Python automatically restores most signal handlers when signals occur, so there is no need to recall this function within the signal handler itself to reregister the handler. That is, except for `SIGCHLD`, a signal handler remains installed until explicitly reset (e.g., by setting the handler to `SIG_DFL` to restore default behavior or to `SIG_IGN` to ignore the signal). `SIGCHLD` behavior is platform specific.

`signal.pause`

Makes the process sleep until the next signal is caught. A `time.sleep` call is similar but doesn't work with signals on my Linux box; it generates an interrupted system call error. A busy `while True: pass` loop here would pause the script, too, but may squander CPU resources.

Here is what this script looks like running on Cygwin on Windows (it works the same on other Unix-like platforms like Linux): a signal number to watch for (12) is passed in on the command line, and the program is made to run in the background with an & shell operator (available in most Unix-like shells):

```
[C:\...\PP4E\System\Processes]$ python signal1.py 12 &
[1] 8224

$ ps
  PID  PPID  PGID   WINPID  TTY  UID   STIME COMMAND
 I  8944     1  8944     8944  con 1004 18:09:54 /usr/bin/bash
 8224  7336  8224    10020  con 1004 18:26:47 /usr/local/bin/python
```

```

8380    7336    8380        428  con 1004 18:26:50 /usr/bin/ps

$ kill -12 8224
Got signal 12 at Sun Mar  7 18:27:28 2010

$ kill -12 8224
Got signal 12 at Sun Mar  7 18:27:30 2010

$ kill -9 8224
[1]+  Killed                  python signal1.py 12

```

Inputs and outputs can be a bit jumbled here because the process prints to the same screen used to type new shell commands. To send the program a signal, the `kill` shell command takes a signal number and a process ID to be signaled (8224); every time a new `kill` command sends a signal, the process replies with a message generated by a Python signal handler function. Signal 9 always kills the process altogether.

The `signal` module also exports a `signal.alarm` function for scheduling a `SIGALRM` signal to occur at some number of seconds in the future. To trigger and catch timeouts, set the alarm and install a `SIGALRM` handler as shown in [Example 5-28](#).

Example 5-28. PP4E\System\Processes\signal2.py

```

"""
set and catch alarm timeout signals in Python; time.sleep doesn't play
well with alarm (or signal in general in my Linux PC), so we call
signal.pause here to do nothing until a signal is received;
"""

import sys, signal, time
def now(): return time.asctime()

def onSignal(signum, stackframe):          # python signal handler
    print('Got alarm', signum, 'at', now())  # most handlers stay in effect

while True:
    print('Setting at', now())
    signal.signal(signal.SIGALRM, onSignal)   # install signal handler
    signal.alarm(5)                          # do signal in 5 seconds
    signal.pause()                           # wait for signals

```

Running this script on Cygwin on Windows causes its `onSignal` handler function to be invoked every five seconds:

```

[C:\...\PP4E\System\Processes]$ python signal2.py
Setting at Sun Mar  7 18:37:10 2010
Got alarm 14 at Sun Mar  7 18:37:15 2010
Setting at Sun Mar  7 18:37:15 2010
Got alarm 14 at Sun Mar  7 18:37:20 2010
Setting at Sun Mar  7 18:37:20 2010
Got alarm 14 at Sun Mar  7 18:37:25 2010
Setting at Sun Mar  7 18:37:25 2010
Got alarm 14 at Sun Mar  7 18:37:30 2010

```

Setting at Sun Mar 7 18:37:30 2010
...Ctrl-C to exit...

Generally speaking, signals must be used with cautions not made obvious by the examples we've just seen. For instance, some system calls don't react well to being interrupted by signals, and only the main thread can install signal handlers and respond to signals in a multithreaded program.

When used well, though, signals provide an event-based communication mechanism. They are less powerful than data streams such as pipes, but are sufficient in situations in which you just need to tell a program that something important has occurred and don't need to pass along any details about the event itself. Signals are sometimes also combined with other IPC tools. For example, an initial signal may inform a program that a client wishes to communicate over a named pipe—the equivalent of tapping someone's shoulder to get their attention before speaking. Most platforms reserve one or more `SIGUSR` signal numbers for user-defined events of this sort. Such an integration structure is sometimes an alternative to running a blocking input call in a spawned thread.

See also the `os.kill(pid, sig)` call for sending signals to known processes from within a Python script on Unix-like platforms, much like the `kill` shell command used earlier; the required process ID can be obtained from the `os.fork` call's child process ID return value or from other interfaces. Like `os.fork`, this call is also available in Cygwin Python, but not in standard Windows Python. Also watch for the discussion about using signal handlers to clean up “zombie” processes in [Chapter 12](#).

The `multiprocessing` Module

Now that you know about IPC alternatives and have had a chance to explore processes, threads, and both process nonportability and thread GIL limitations, it turns out that there is another alternative, which aims to provide just the best of both worlds. As mentioned earlier, Python's standard library `multiprocessing` module package allows scripts to spawn processes using an API very similar to the `threading` module.

This relatively new package works on both Unix and Windows, unlike low-level process forks. It supports a process spawning model which is largely platform-neutral, and provides tools for related goals, such as IPC, including locks, pipes, and queues. In addition, because it uses processes instead of threads to run code in parallel, it effectively works around the limitations of the thread GIL. Hence, `multiprocessing` allows the programmer to leverage the capacity of multiple processors for parallel tasks, while retaining much of the simplicity and portability of the threading model.

Why `multiprocessing`?

So why learn yet another parallel processing paradigm and toolkit, when we already have the threads, processes, and IPC tools like sockets, pipes, and thread queues that

we've already studied? Before we get into the details, I want to begin with a few words about why you may (or may not) care about this package. In more specific terms, although this module's performance may not compete with that of pure threads or process forks for some applications, this module offers a compelling solution for many:

- Compared to raw process forks, you gain cross-platform portability and powerful IPC tools.
- Compared to threads, you essentially trade some potential and platform-dependent extra task start-up time for the ability to run tasks in truly parallel fashion on multi-core or multi-CPU machines.

On the other hand, this module imposes some constraints and tradeoffs that threads do not:

- Since objects are copied across process boundaries, shared mutable state does not work as it does for threads—changes in one process are not generally noticed in the other. Really, freely shared state may be the most compelling reason to use threads; its absence in this module may prove limiting in some threading contexts.
- Because this module requires pickleability for both its processes on Windows, as well as some of its IPC tools in general, some coding paradigms are difficult or nonportable—especially if they use bound methods or pass unpickleable objects such as sockets to spawned processes.

For instance, common coding patterns with lambda that work for the `threading` module cannot be used as process target callables in this module on Windows, because they cannot be pickled. Similarly, because bound object methods are also not pickleable, a threaded program may require a more indirect design if it either runs bound methods in its threads or implements thread exit actions by posting arbitrary callables (possibly including bound methods) on shared queues. The in-process model of threads supports such direct lambda and bound method use, but the separate processes of `multiprocessing` do not.

In fact we'll write a thread manager for GUIs in [Chapter 10](#) that relies on queueing in-process callables this way to implement thread exit actions—the callables are queued by worker threads, and fetched and dispatched by the main thread. Because the threaded PyMailGUI program we'll code in [Chapter 14](#) both uses this manager to queue bound methods for thread exit actions and runs bound methods as the main action of a thread itself, it could not be directly translated to the separate process model implied by `multiprocessing`.

Without getting into too many details here, to use `multiprocessing`, PyMailGUI's actions might have to be coded as simple functions or complete process subclasses for pickleability. Worse, they may have to be implemented as simpler action identifiers dispatched in the main process, if they update either the GUI itself or object state in general—pickling results in an object copy in the receiving process, not a reference to the original, and forks on Unix essentially copy an entire process. Updating the state

of a mutable message cache copied by pickling it to pass to a new process, for example, has no effect on the original.

The pickleability constraints for process arguments on Windows can limit `multiprocessing`'s scope in other contexts as well. For instance, in [Chapter 12](#), we'll find that this module doesn't directly solve the lack of portability for the `os.fork` call for traditionally coded *socket servers* on Windows, because connected sockets are not pickled correctly when passed into a new process created by this module to converse with a client. In this context, threads provide a more portable and likely more efficient solution.

Applications that pass simpler types of messages, of course, may fare better. Message constraints are easier to accommodate when they are part of an initial process-based design. Moreover, other tools in this module, such as its managers and shared memory API, while narrowly focused and not as general as shared thread state, offer additional mutable state options for some programs.

Fundamentally, though, because `multiprocessing` is based on separate processes, it may be best geared for tasks which are relatively independent, do not share mutable object state freely, and can make do with the message passing and shared memory tools provided by this module. This includes many applications, but this module is not necessarily a direct replacement for every threaded program, and it is not an alternative to process forks in all contexts.

To truly understand both this module package's benefits, as well as its tradeoffs, let's turn to a first example and explore this package's implementation along the way.

The Basics: Processes and Locks

We don't have space to do full justice to this sophisticated module in this book; see its coverage in the Python library manual for the full story. But as a brief introduction, by design most of this module's interfaces mirror the `threading` and `queue` modules we've already met, so they should already seem familiar. For example, the `multiprocessing` module's `Process` class is intended to mimic the `threading` module's `Thread` class we met earlier—it allows us to launch a function call in parallel with the calling script; with this module, though, the function runs in a process instead of a thread. [Example 5-29](#) illustrates these basics in action:

Example 5-29. PP4E\System\Processes\multi1.py

```
"""
multiprocess basics: Process works like threading.Thread, but
runs function call in parallel in a process instead of a thread;
locks can be used to synchronize, e.g. prints on some platforms;
starts new interpreter on windows, forks a new process on unix;
"""


```

```
import os
from multiprocessing import Process, Lock
```

```

def whoami(label, lock):
    msg = '%s: name:%s, pid:%s'
    with lock:
        print(msg % (label, __name__, os.getpid()))

if __name__ == '__main__':
    lock = Lock()
    whoami('function call', lock)

    p = Process(target=whoami, args=('spawned child', lock))
    p.start()
    p.join()

    for i in range(5):
        Process(target=whoami, args=(('run process %s' % i), lock)).start()

    with lock:
        print('Main process exit.')

```

When run, this script first calls a function directly and in-process; then launches a call to that function in a new process and waits for it to exit; and finally spawns five function call processes in parallel in a loop—all using an API identical to that of the `threading.Thread` model we studied earlier in this chapter. Here’s this script’s output on Windows; notice how the five child processes spawned at the end of this script outlive their parent, as is the usual case for processes:

```

C:\...\PP4E\System\Processes> multi1.py
function call: name:_main_, pid:8752
spawned child: name:_main_, pid:9268
Main process exit.
run process 3: name:_main_, pid:9296
run process 1: name:_main_, pid:8792
run process 4: name:_main_, pid:2224
run process 2: name:_main_, pid:8716
run process 0: name:_main_, pid:6936

```

Just like the `threading.Thread` class we met earlier, the `multiprocessing.Process` object can either be passed a `target` with arguments (as done here) or subclassed to redefine its `run` action method. Its `start` method invokes its `run` method in a new process, and the default `run` simply calls the passed-in target. Also like `threading`, a `join` method waits for child process exit, and a `Lock` object is provided as one of a handful of process synchronization tools; it’s used here to ensure that prints don’t overlap among processes on platforms where this might matter (it may not on Windows).

Implementation and usage rules

Technically, to achieve its portability, this module currently works by selecting from platform-specific alternatives:

- On Unix, it forks a new child process and invokes the `Process` object’s `run` method in the new child.

- On Windows, it spawns a new interpreter by using Windows-specific process creation tools, passing the pickled `Process` object in to the new process over a pipe, and starting a “`python -c`” command line in the new process, which runs a special Python-coded function in this package that reads and unpickles the `Process` and invokes its `run` method.

We met pickling briefly in [Chapter 1](#), and we will study it further later in this book. The implementation is a bit more complex than this, and is prone to change over time, of course, but it’s really quite an amazing trick. While the portable API generally hides these details from your code, its basic structure can still have subtle impacts on the way you’re allowed to use it. For instance:

- On Windows, the main process’s logic should generally be nested under a `__name__ == __main__` test as done here when using this module, so it can be imported freely by a new interpreter without side effects. As we’ll learn in more detail in [Chapter 17](#), unpickling classes and functions requires an import of their enclosing module, and this is the root of this requirement.
- Moreover, when globals are accessed in child processes on Windows, their values may not be the same as that in the parent at `start` time, because their module will be imported into a new process.
- Also on Windows, all arguments to `Process` must be pickleable. Because this includes `target`, targets should be simple functions so they can be pickled; they cannot be bound or unbound object *methods* and cannot be functions created with a *lambda*. See `pickle` in Python’s library manual for more on pickleability rules; nearly every object type works, but callables like functions and classes must be importable—they are pickled by name only, and later imported to recreate bytecode. On Windows, objects with system state, such as connected sockets, won’t generally work as arguments to a process target either, because they are not pickleable.
- Similarly, instances of custom `Process` subclasses must be pickleable on Windows as well. This includes all their attribute values. Objects available in this package (e.g., `Lock` in [Example 5-29](#)) are pickleable, and so may be used as both `Process` constructor arguments and subclass attributes.
- IPC objects in this package that appear in later examples like `Pipe` and `Queue` accept only pickleable objects, because of their implementation (more on this in the next section).
- On Unix, although a child process can make use of a shared global item created in the parent, it’s better to pass the object as an argument to the child process’s constructor, both for portability to Windows and to avoid potential problems if such objects were garbage collected in the parent.

There are additional rules documented in the library manual. In general, though, if you stick to passing in shared objects to processes and using the synchronization and

communication tools provided by this package, your code will usually be portable and correct. Let's look next at a few of those tools in action.

IPC Tools: Pipes, Shared Memory, and Queues

While the processes created by this package can always communicate using general system-wide tools like the sockets and fifo files we met earlier, the `multiprocessing` module also provides portable message passing tools specifically geared to this purpose for the processes it spawns:

- Its `Pipe` object provides an anonymous pipe, which serves as a connection between two processes. When called, `Pipe` returns two `Connection` objects that represent the ends of the pipe. Pipes are bidirectional by default, and allow arbitrary pickleable Python objects to be sent and received. On Unix they are implemented internally today with either a connected socket pair or the `os.pipe` call we met earlier, and on Windows with named pipes specific to that platform. Much like the `Process` object described earlier, though, the `Pipe` object's portable API spares callers from such things.
- Its `Value` and `Array` objects implement shared process/thread-safe memory for communication between processes. These calls return scalar and array objects based in the `ctypes` module and created in shared memory, with access synchronized by default.
- Its `Queue` object serves as a FIFO list of Python objects, which allows multiple producers and consumers. A queue is essentially a pipe with extra locking mechanisms to coordinate more arbitrary accesses, and inherits the pickleability constraints of `Pipe`.

Because these devices are safe to use across multiple processes, they can often serve to synchronize points of communication and obviate lower-level tools like locks, much the same as the thread queues we met earlier. As usual, a pipe (or a pair of them) may be used to implement a request/reply model. Queues support more flexible models; in fact, a GUI that wishes to avoid the limitations of the GIL might use the `multiprocessing` module's `Process` and `Queue` to spawn long-running tasks that post results, rather than threads. As mentioned, although this may incur extra start-up overhead on some platforms, unlike threads today, tasks coded this way can be as truly parallel as the underlying platform allows.

One constraint worth noting here: this package's pipes (and by proxy, queues) *pickle* the objects passed through them, so that they can be reconstructed in the receiving process (as we've seen, on Windows the receiver process may be a fully independent Python interpreter). Because of that, they do not support unpickleable objects; as suggested earlier, this includes some callables like bound methods and lambda functions (see file `multi-badq.py` in the book examples package for a demonstration of code that violates this constraint). Objects with system state, such as sockets, may fail as well.

Most other Python object types, including classes and simple functions, work fine on pipes and queues.

Also keep in mind that because they are pickled, objects transferred this way are effectively *copied* in the receiving process; direct in-place changes to mutable objects' state won't be noticed in the sender. This makes sense if you remember that this package runs independent processes with their own memory spaces; state cannot be as freely shared as in threading, regardless of which IPC tools you use.

multiprocessing pipes

To demonstrate the IPC tools listed above, the next three examples implement three flavors of communication between parent and child processes. [Example 5-30](#) uses a simple shared pipe object to send and receive data between parent and child processes.

Example 5-30. PP4E\System\Processes\multi2.py

```
"""
Use multiprocessing anonymous pipes to communicate. Returns 2 connection
object representing ends of the pipe: objects are sent on one end and
received on the other, though pipes are bidirectional by default
"""

import os
from multiprocessing import Process, Pipe

def sender(pipe):
    """
    send object to parent on anonymous pipe
    """
    pipe.send(['spam'] + [42, 'eggs'])
    pipe.close()

def talker(pipe):
    """
    send and receive objects on a pipe
    """
    pipe.send(dict(name='Bob', spam=42))
    reply = pipe.recv()
    print('talker got:', reply)

if __name__ == '__main__':
    (parentEnd, childEnd) = Pipe()                                # spawn child with pipe
    Process(target=sender, args=(childEnd,)).start()              # receive from child
    print('parent got:', parentEnd.recv())                         # or auto-closed on gc
    parentEnd.close()

    (parentEnd, childEnd) = Pipe()                                # receieve from child
    child = Process(target=talker, args=(childEnd,))               # send to child
    child.start()
    print('parent got:', parentEnd.recv())
    parentEnd.send({x * 2 for x in 'spam'})
```

```
child.join()                                # wait for child exit
print('parent exit')
```

When run on Windows, here's this script's output—one child passes an object to the parent, and the other both sends and receives on the same pipe:

```
C:\...\PP4E\System\Processes> multi2.py
parent got: ['spam', 42, 'eggs']
parent got: {'name': 'Bob', 'spam': 42}
talker got: {'ss', 'aa', 'pp', 'mm'}
parent exit
```

This module's pipe objects make communication between two processes portable (and nearly trivial).

Shared memory and globals

Example 5-31 uses shared memory to serve as both inputs and outputs of spawned processes. To make this work portably, we must create objects defined by the package and pass them to `Process` constructors. The last test in this demo (“loop4”) probably represents the most common use case for shared memory—that of distributing computation work to multiple parallel processes.

Example 5-31. PP4E\System\Processes\multi3.py

```
"""
Use multiprocessing shared memory objects to communicate.
Passed objects are shared, but globals are not on Windows.
Last test here reflects common use case: distributing work.
"""

import os
from multiprocessing import Process, Value, Array

procs = 3
count = 0      # per-process globals, not shared

def showdata(label, val, arr):
    """
    print data values in this process
    """
    msg = '%-12s: pid:%4s, global:%s, value:%s, array:%s'
    print(msg % (label, os.getpid(), count, val.value, list(arr)))

def updater(val, arr):
    """
    communicate via shared memory
    """
    global count
    count += 1                      # global count not shared
    val.value += 1                   # passed in objects are
    for i in range(3): arr[i] += 1

if __name__ == '__main__':
```

```

scalar = Value('i', 0)           # shared memory: process/thread safe
vector = Array('d', procs)       # type codes from ctypes: int, double

# show start value in parent process
showdata('parent start', scalar, vector)

# spawn child, pass in shared memory
p = Process(target=showdata, args=('child ', scalar, vector))
p.start(); p.join()

# pass in shared memory updated in parent, wait for each to finish
# each child sees updates in parent so far for args (but not global)

print('\nloop1 (updates in parent, serial children)...')
for i in range(procs):
    count += 1
    scalar.value += 1
    vector[i] += 1
    p = Process(target=showdata, args=(('process %s' % i), scalar, vector))
    p.start(); p.join()

# same as prior, but allow children to run in parallel
# all see the last iteration's result because all share objects

print('\nloop2 (updates in parent, parallel children)...')
ps = []
for i in range(procs):
    count += 1
    scalar.value += 1
    vector[i] += 1
    p = Process(target=showdata, args=(('process %s' % i), scalar, vector))
    p.start()
    ps.append(p)
for p in ps: p.join()

# shared memory updated in spawned children, wait for each

print('\nloop3 (updates in serial children)...')
for i in range(procs):
    p = Process(target=updater, args=(scalar, vector))
    p.start()
    p.join()
showdata('parent temp', scalar, vector)

# same, but allow children to update in parallel

ps = []
print('\nloop4 (updates in parallel children)...')
for i in range(procs):
    p = Process(target=updater, args=(scalar, vector))
    p.start()
    ps.append(p)
for p in ps: p.join()
                                # global count=6 in parent only

```

```
# show final results here          # scalar=12: +6 parent, +6 in 6 children
showdata('parent end', scalar, vector) # array[i]=8: +2 parent, +6 in 6 children
```

The following is this script's output on Windows. Trace through this and the code to see how it runs; notice how the changed value of the global variable is not shared by the spawned processes on Windows, but passed-in `Value` and `Array` objects are. The final output line reflects changes made to shared memory in both the parent and spawned children—the array's final values are all 8.0, because they were incremented twice in the parent, and once in each of six spawned children; the scalar value similarly reflects changes made by both parent and child; but unlike for threads, the global is per-process data on Windows:

```
C:\...\PP4E\System\Processes> multi3.py
parent start: pid:6204, global:0, value:0, array:[0.0, 0.0, 0.0]
child      : pid:9660, global:0, value:0, array:[0.0, 0.0, 0.0]

loop1 (updates in parent, serial children)...
process 0   : pid:3900, global:0, value:1, array:[1.0, 0.0, 0.0]
process 1   : pid:5072, global:0, value:2, array:[1.0, 1.0, 0.0]
process 2   : pid:9472, global:0, value:3, array:[1.0, 1.0, 1.0]

loop2 (updates in parent, parallel children)...
process 1   : pid:9468, global:0, value:6, array:[2.0, 2.0, 2.0]
process 2   : pid:9036, global:0, value:6, array:[2.0, 2.0, 2.0]
process 0   : pid:9548, global:0, value:6, array:[2.0, 2.0, 2.0]

loop3 (updates in serial children)...
parent temp : pid:6204, global:6, value:9, array:[5.0, 5.0, 5.0]

loop4 (updates in parallel children)...
parent end  : pid:6204, global:6, value:12, array:[8.0, 8.0, 8.0]
```

If you imagine the last test here run with a much larger array and many more parallel children, you might begin to sense some of the power of this package for distributing work.

Queues and subclassing

Finally, besides basic spawning and IPC tools, the `multiprocessing` module also:

- Allows its `Process` class to be subclassed to provide structure and state retention (much like `threading.Thread`, but for processes).
- Implements a process-safe `Queue` object which may be shared by any number of processes for more general communication needs (much like `queue.Queue`, but for processes).

Queues support a more flexible multiple client/server model. [Example 5-32](#), for instance, spawns three producer threads to post to a shared queue and repeatedly polls for results to appear—in much the same fashion that a GUI might collect results in parallel with the display itself, though here the concurrency is achieved with processes instead of threads.

Example 5-32. PP4E\System\Processes\multi4.py

```
"""
Process class can also be subclassed just like threading.Thread;
Queue works like queue.Queue but for cross-process, not cross-thread
"""

import os, time, queue
from multiprocessing import Process, Queue          # process-safe shared queue
                                                    # queue is a pipe + locks/semas

class Counter(Process):
    label = '@'
    def __init__(self, start, queue):               # retain state for use in run
        self.state = start
        self.post = queue
        Process.__init__(self)

    def run(self):                                 # run in newprocess on start()
        for i in range(3):
            time.sleep(1)
            self.state += 1
            print(self.label, self.pid, self.state)   # self.pid is this child's pid
            self.post.put([self.pid, self.state])      # stdout file is shared by all
            print(self.label, self.pid, '-')

if __name__ == '__main__':
    print('start', os.getpid())
    expected = 9

    post = Queue()
    p = Counter(0, post)                          # start 3 processes sharing queue
    q = Counter(100, post)                         # children are producers
    r = Counter(1000, post)
    p.start(); q.start(); r.start()

    while expected:                               # parent consumes data on queue
        time.sleep(0.5)                          # this is essentially like a GUI,
        try:                                     # though GUIs often use threads
            data = post.get(block=False)
        except queue.Empty:
            print('no data...')
        else:
            print('posted:', data)
            expected -= 1

    p.join(); q.join(); r.join()                  # must get before join putter
    print('finish', os.getpid(), r.exitcode)     # exitcode is child exit status
```

Notice in this code how:

- The `time.sleep` calls in this code's producer simulate long-running tasks.
- All four processes share the same output stream; `print` calls go the same place and don't overlap badly on Windows (as we saw earlier, the `multiprocessing` module also has a shareable `Lock` object to synchronize access if required).

- The exit status of child process is available after they finish in their `exitcode` attribute.

When run, the output of the main consumer process traces its queue fetches, and the (indented) output of spawned child producer processes gives process IDs and state.

```
C:\...\PP4E\System\Processes> multi4.py
start 6296
no data...
no data...
    @ 8008 101
posted: [8008, 101]
    @ 6068 1
    @ 3760 1001
posted: [6068, 1]
    @ 8008 102
posted: [3760, 1001]
    @ 6068 2
    @ 3760 1002
posted: [8008, 102]
    @ 8008 103
    @ 8008 -
posted: [6068, 2]
    @ 6068 3
    @ 6068 -
    @ 3760 1003
    @ 3760 -
posted: [3760, 1002]
posted: [8008, 103]
posted: [6068, 3]
posted: [3760, 1003]
finish 6296 0
```

If you imagine the “@” lines here as results of long-running operations and the others as a main GUI thread, the wide relevance of this package may become more apparent.

Starting Independent Programs

As we learned earlier, independent programs generally communicate with system-global tools such as sockets and the fifo files we studied earlier. Although processes spawned by `multiprocessing` can leverage these tools, too, their closer relationship affords them the host of additional IPC communication devices provided by this module.

Like threads, `multiprocessing` is designed to run function calls in parallel, not to start entirely separate programs directly. Spawning functions might use tools like `os.system`, `os.popen`, and `subprocess` to start a program if such an operation might block the caller, but there’s otherwise often no point in starting a process that just starts a program (you might as well start the program and skip a step). In fact, on Windows, `multiprocessing` today uses the same process creation call as `subprocess`, so there’s little point in starting two processes to run one.

It is, however, possible to start new programs in the child processes spawned, using tools like the `os.exec*` calls we met earlier—by spawning a process portably with `multiprocessing` and overlaying it with a new program this way, we start a new independent program, and effectively work around the lack of the `os.fork` call in standard Windows Python.

This generally assumes that the new program doesn't require any resources passed in by the `Process` API, of course (once a new program starts, it erases that which was running), but it offers a portable equivalent to the `fork/exec` combination on Unix. Furthermore, programs started this way can still make use of more traditional IPC tools, such as sockets and fifos, we met earlier in this chapter. [Example 5-33](#) illustrates the technique.

Example 5-33. PP4E\System\Processes\multi5.py

```
"Use multiprocessing to start independent programs, os.fork or not"

import os
from multiprocessing import Process

def runprogram(arg):
    os.execlp('python', 'python', 'child.py', str(arg))

if __name__ == '__main__':
    for i in range(5):
        Process(target=runprogram, args=(i,)).start()
    print('parent exit')
```

This script starts 5 instances of the `child.py` script we wrote in [Example 5-4](#) as independent processes, without waiting for them to finish. Here's this script at work on Windows, after deleting a superfluous system prompt that shows up arbitrarily in the middle of its output (it runs the same on Cygwin, but the output is not interleaved there):

```
C:\...\PP4E\System\Processes> type child.py
import os, sys
print('Hello from child', os.getpid(), sys.argv[1])

C:\...\PP4E\System\Processes> multi5.py
parent exit
Hello from child 9844 2
Hello from child 8696 4
Hello from child 1840 0
Hello from child 6724 1
Hello from child 9368 3
```

This technique isn't possible with threads, because all threads run in the same process; overlaying it with a new program would kill all its threads. Though this is unlikely to be as fast as a `fork/exec` combination on Unix, it at least provides similar and portable functionality on Windows when required.

And Much More

Finally, `multiprocessing` provides many more tools than these examples deploy, including condition, event, and semaphore synchronization tools, and local and remote managers that implement servers for shared object. For instance, [Example 5-34](#) demonstrates its support for *pools*—spawned children that work in concert on a given task.

Example 5-34. PP4E\System\Processes\multi6.py

```
"Plus much more: process pools, managers, locks, condition,..."  
  
import os  
from multiprocessing import Pool  
  
def powers(x):  
    #print(os.getpid())           # enable to watch children  
    return 2 ** x  
  
if __name__ == '__main__':  
    workers = Pool(processes=5)  
  
    results = workers.map(powers, [2]*100)  
    print(results[:16])  
    print(results[-2:])  
  
    results = workers.map(powers, range(100))  
    print(results[:16])  
    print(results[-2:])
```

When run, Python arranges to delegate portions of the task to workers run in parallel:

```
C:\...\PP4E\System\Processes> multi6.py  
[4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4]  
[4, 4]  
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768]  
[316912650057057350374175801344, 633825300114114700748351602688]
```

And a little less...

To be fair, besides such additional features and tools, `multiprocessing` also comes with additional constraints beyond those we've already covered (pickleability, mutable state, and so on). For example, consider the following sort of code:

```
def action(arg1, arg2):  
    print(arg1, arg2)  
  
if __name__ == '__main__':  
    Process(target=action, args=('spam', 'eggs')).start()    # shell waits for child
```

This works as expected, but if we change the last line to the following it fails on Windows because *lambdas* are not pickleable (really, not importable):

```
Process(target=(lambda: action('spam', 'eggs'))).start()  # fails!-not pickleable
```

This precludes a common coding pattern that uses lambda to add data to calls, which we'll use often for callbacks in the GUI part of this book. Moreover, this differs from the `threading` module that is the model for this package—calls like the following which work for threads must be translated to a callable and arguments:

```
threading.Thread(target=(lambda: action(2, 4))).start() # but lambdas work here
```

Conversely, some behavior of the `threading` module is mimicked by `multiprocessing`, whether you wish it did or not. Because programs using this package wait for child processes to end by default, we must mark processes as `daemon` if we don't want to block the shell where the following sort of code is run (technically, parents attempt to terminateemonic children on exit, which means that the program can exit when only daemonic children remain, much like `threading`):

```
def action(arg1, arg2):
    print(arg1, arg2)
    time.sleep(5)           # normally prevents the parent from exiting

if __name__ == '__main__':
    p = Process(target=action, args=('spam', 'eggs'))
    p.daemon = True          # don't wait for it
    p.start()
```

There's more on some of these issues in the Python library manual; they are not showstoppers by any stretch, but special cases and potential pitfalls to some. We'll revisit the lambda and daemon issues in a more realistic context in [Chapter 8](#), where we'll use `multiprocessing` to launch GUI demos independently.

Why `multiprocessing`? The Conclusion

As this section's examples suggest, `multiprocessing` provides a powerful alternative which aims to combine the portability and much of the utility of threads with the fully parallel potential of processes and offers additional solutions to IPC, exit status, and other parallel processing goals.

Hopefully, this section has also given you a better understanding of this module's tradeoffs discussed at its beginning. In particular, its separate process model precludes the freely shared mutable state of threads, and bound methods and lambdas are prohibited by both the pickleability requirements of its IPC pipes and queues, as well as its process action implementation on Windows. Moreover, its requirement of pickleability for process arguments on Windows also precludes it as an option for conversing with clients in socket servers portably.

While not a replacement for threading in all applications, though, `multiprocessing` offers compelling solutions for many. Especially for parallel-programming tasks which can be designed to avoid its limitations, this module can offer both performance and portability that Python's more direct multitasking tools cannot.

Unfortunately, beyond this brief introduction, we don't have space for a more complete treatment of this module in this book. For more details, refer to the Python library manual. Here, we turn next to a handful of additional program launching tools and a wrap up of this chapter.

Other Ways to Start Programs

We've seen a variety of ways to launch programs in this book so far—from the `os.fork/exec` combination on Unix, to portable shell command-line launchers like `os.system`, `os.popen`, and `subprocess`, to the portable `multiprocessing` module options of the last section. There are still other ways to start programs in the Python standard library, some of which are more platform neutral or obscure than others. This section wraps up this chapter with a quick tour through this set.

The `os.spawn` Calls

The `os.spawnv` and `os.spawnve` calls were originally introduced to launch programs on Windows, much like a `fork/exec` call combination on Unix-like platforms. Today, these calls work on both Windows and Unix-like systems, and additional variants have been added to parrot `os.exec`.

In recent versions of Python, the portable `subprocess` module has started to supersede these calls. In fact, Python's library manual includes a note stating that this module has more powerful and equivalent tools and should be preferred to `os.spawn` calls. Moreover, the newer `multiprocessing` module can achieve similarly portable results today when combined with `os.exec` calls, as we saw earlier. Still, the `os.spawn` calls continue to work as advertised and may appear in Python code you encounter.

The `os.spawn` family of calls execute a program named by a command line in a new process, on both Windows and Unix-like systems. In basic operation, they are similar to the `fork/exec` call combination on Unix and can be used as alternatives to the `system` and `popen` calls we've already learned. In the following interaction, for instance, we start a Python program with a command line in two traditional ways (the second also reads its output):

```
C:\...\PP4E\System\Processes> python
>>> print(open('makewords.py').read())
print('spam')
print('eggs')
print('ham')

>>> import os
>>> os.system('python makewords.py')
spam
eggs
ham
0
```

```
>>> result = os.popen('python makewords.py').read()
>>> print(result)
spam
eggs
ham
```

The equivalent `os.spawn` calls achieve the same effect, with a slightly more complex call signature that provides more control over the way the program is launched:

```
>>> os.spawnv(os.P_WAIT, r'C:\Python31\python', ('python', 'makewords.py'))
spam
eggs
ham
0
>>> os.spawnl(os.P_NOWAIT, r'C:\Python31\python', 'python', 'makewords.py')
1820
>>> spam
eggs
ham
```

The `spawn` calls are also much like forking programs in Unix. They don't actually copy the calling process (so shared descriptor operations won't work), but they can be used to start a program running completely independent of the calling program, even on Windows. The script in [Example 5-35](#) makes the similarity to Unix programming patterns more obvious. It launches a program with a `fork/exec` combination on Unix-like platforms (including Cygwin), or an `os.spawnv` call on Windows.

Example 5-35. PP4E\System\Processes\spawnv.py

```
"""
start up 10 copies of child.py running in parallel;
use spawnv to launch a program on Windows (like fork+exec);
P_OVERLAY replaces, P_DETACH makes child stdout go nowhere;
or use portable subprocess or multiprocessing options today!
"""

import os, sys

for i in range(10):
    if sys.platform[:3] == 'win':
        pypath = sys.executable
        os.spawnv(os.P_NOWAIT, pypath, ('python', 'child.py', str(i)))
    else:
        pid = os.fork()
        if pid != 0:
            print('Process %d spawned' % pid)
        else:
            os.execlp('python', 'python', 'child.py', str(i))
print('Main process exiting.')
```

To make sense of these examples, you have to understand the arguments being passed to the spawn calls. In this script, we call `os.spawnv` with a process mode flag, the full directory path to the Python interpreter, and a tuple of strings representing the shell command line with which to start a new program. The path to the Python interpreter

executable program running a script is available as `sys.executable`. In general, the *process mode* flag is taken from these predefined values:

`os.P_NOWAIT` and `os.P_NOWAITO`

The `spawn` functions will return as soon as the new process has been created, with the process ID as the return value. Available on Unix and Windows.

`os.P_WAIT`

The `spawn` functions will not return until the new process has run to completion and will return the exit code of the process if the run is successful or “-signal” if a signal kills the process. Available on Unix and Windows.

`os.P_DETACH` and `os.P_OVERLAY`

`P_DETACH` is similar to `P_NOWAIT`, but the new process is detached from the console of the calling process. If `P_OVERLAY` is used, the current program will be replaced (much like `os.exec`). Available on Windows.

In fact, there are eight different calls in the `spawn` family, which all start a program but vary slightly in their call signatures. In their names, an “l” means you list arguments individually, “p” means the executable file is looked up on the system path, and “e” means a dictionary is passed in to provide the shelled environment of the spawned program: the `os.spawnve` call, for example, works the same way as `os.spawnv` but accepts an extra fourth dictionary argument to specify a different shell environment for the spawned program (which, by default, inherits all of the parent’s settings):

```
os.spawnl(mode, path, ...)
os.spawnle(mode, path, ..., env)
os.spawnlp(mode, file, ...)           # Unix only
os.spawnlpe(mode, file, ..., env)    # Unix only
os.spawnnv(mode, path, args)
os.spawnve(mode, path, args, env)
os.spawnvp(mode, file, args)         # Unix only
os.spawnvpe(mode, file, args, env)   # Unix only
```

Because these calls mimic the names and call signatures of the `os.exec` variants, see earlier in this chapter for more details on the differences between these call forms. Unlike the `os.exec` calls, only half of the `os.spawn` forms—those without system path checking (and hence without a “p” in their names)—are currently implemented on Windows. All the process mode flags are supported on Windows, but detach and overlay modes are not available on Unix. Because this sort of detail may be prone to change, to verify which are present, be sure to see the library manual or run a `dir` built-in function call on the `os` module after an import.

Here is the script in [Example 5-35](#) at work on Windows, spawning 10 independent copies of the `child.py` Python program we met earlier in this chapter:

```
C:\...\PP4E\System\Processes> type child.py
import os, sys
print('Hello from child', os.getpid(), sys.argv[1])

C:\...\PP4E\System\Processes> python spawnv.py
```

```
Hello from child -583587 0
Hello from child -558199 2
Hello from child -586755 1
Hello from child -562171 3
Main process exiting.
Hello from child -581867 6
Hello from child -588651 5
Hello from child -568247 4
Hello from child -563527 7
Hello from child -543163 9
Hello from child -587083 8
```

Notice that the copies print their output in random order, and the parent program exits before all children do; all of these programs are really running in parallel on Windows. Also observe that the child program's output shows up in the console box where `spawnv.py` was run; when using `P_NOWAIT`, standard output comes to the parent's console, but it seems to go nowhere when using `P_DETACH` (which is most likely a feature when spawning GUI programs).

But having shown you this call, I need to again point out that both the `subprocess` and `multiprocessing` modules offer more portable alternatives for spawning programs with command lines today. In fact, unless `os.spawn` calls provide unique behavior you can't live without (e.g., control of shell window pop ups on Windows), the platform-specific alternatives code of [Example 5-35](#) can be replaced altogether with the portable `multiprocessing` code in [Example 5-33](#).

The `os.startfile` call on Windows

Although `os.spawn` calls may be largely superfluous today, there are other tools that can still make a strong case for themselves. For instance, the `os.system` call can be used on Windows to launch a DOS `start` command, which opens (i.e., runs) a file independently based on its Windows filename associations, as though it were clicked. `os.startfile` makes this even simpler in recent Python releases, and it can avoid blocking its caller, unlike some other tools.

Using the DOS start command

To understand why, first you need to know how the DOS `start` command works in general. Roughly, a DOS command line of the form `start command` works as if `command` were typed in the Windows Run dialog box available in the Start button menu. If `command` is a filename, it is opened exactly as if its name was double-clicked in the Windows Explorer file selector GUI.

For instance, the following three DOS commands automatically start Internet Explorer, my registered image viewer program, and my sound media player program on the files named in the commands. Windows simply opens the file with whatever program is associated to handle filenames of that form. Moreover, all three of these programs run independently of the DOS console box where the command is typed:

```
C:\...\PP4E\System\Media> start lp4e-preface-preview.html  
C:\...\PP4E\System\Media> start ora-lp4e.jpg  
C:\...\PP4E\System\Media> start sousa.au
```

Because the `start` command can run any file and command line, there is no reason it cannot also be used to start an independently running Python program:

```
C:\...\PP4E\System\Processes> start child.py 1
```

This works because Python is registered to open names ending in `.py` when it is installed. The script `child.py` is launched independently of the DOS console window even though we didn't provide the name or path of the Python interpreter program. Because `child.py` simply prints a message and exits, though, the result isn't exactly satisfying: a new DOS window pops up to serve as the script's standard output, and it immediately goes away when the child exits. To do better, add an `input` call at the bottom of the program file to wait for a key press before exiting:

```
C:\...\PP4E\System\Processes> type child-wait.py  
import os, sys  
print('Hello from child', os.getpid(), sys.argv[1])  
input("Press <Enter>")      # don't flash on Windows  
  
C:\...\PP4E\System\Processes> start child-wait.py 2
```

Now the child's DOS window pops up and stays up after the `start` command has returned. Pressing the Enter key in the pop-up DOS window makes it go away.

Using `start` in Python scripts

Since we know that Python's `os.system` and `os.popen` can be called by a script to run *any* command line that can be typed at a DOS shell prompt, we can also start independently running programs from a Python script by simply running a DOS `start` command line. For instance:

```
C:\...\PP4E\System\Media> python  
>>> import os  
>>> cmd = 'start lp4e-preface-preview.html'          # start IE browser  
>>> os.system(cmd)                                # runs independent  
0
```

The Python `os.system` calls here `start` whatever web page browser is registered on your machine to open `.html` files (unless these programs are already running). The launched programs run completely independent of the Python session—when running a DOS `start` command, `os.system` does not wait for the spawned program to exit.

The `os.startfile` call

In fact, `start` is so useful that recent Python releases also include an `os.startfile` call, which is essentially the same as spawning a DOS start command with `os.system` and works as though the named file were double-clicked. The following calls, for instance, have a similar effect:

```
>>> os.startfile('lp-code-readme.txt')
>>> os.system('start lp-code-readme.txt')
```

Both pop up the text file in Notepad on my Windows computer. Unlike the second of these calls, though, `os.startfile` provides no option to wait for the application to close (the DOS `start` command's `/WAIT` option does) and no way to retrieve the application's exit status (returned from `os.system`).

On recent versions of Windows, the following has a similar effect, too, because the registry is used at the command line (though this form pauses until the file's viewer is closed—like using `start /WAIT`):

```
>>> os.system('lp-code-readme.txt')      # 'start' is optional today
```

This is a convenient way to open arbitrary document and media files, but keep in mind that the `os.startfile` call works only on Windows, because it uses the Windows registry to know how to open a file. In fact, there are even more obscure and nonportable ways to launch programs, including Windows-specific options in the PyWin32 package, which we'll finesse here. If you want to be more platform neutral, consider using one of the other many program launcher tools we've seen, such as `os.popen` or `os.spawnv`. Or better yet, write a module to hide the details—as the next and final section demonstrates.

A Portable Program-Launch Framework

With all of these different ways to start programs on different platforms, it can be difficult to remember what tools to use in a given situation. Moreover, some of these tools are called in ways that are complicated and thus easy to forget. Although modules like `subprocess` and `multiprocessing` offer fully portable options today, other tools sometimes provide more specific behavior that's better on a given platform; shell window pop ups on Windows, for example, are often better suppressed.

I write scripts that need to launch Python programs often enough that I eventually wrote a module to try to hide most of the underlying details. By encapsulating the details in this module, I'm free to change them to use new tools in the future without breaking code that relies on them. While I was at it, I made this module smart enough to automatically pick a "best" launch scheme based on the underlying platform. Laziness is the mother of many a useful module.

Example 5-36 collects in a single module many of the techniques we've met in this chapter. It implements an abstract superclass, `LaunchMode`, which defines what it means to start a Python program named by a shell command line, but it doesn't define how. Instead, its subclasses provide a `run` method that actually starts a Python program according to a given scheme and (optionally) define an `announce` method to display a program's name at startup time.

Example 5-36. PP4E\launchmodes.py

```
"""
#####
# launch Python programs with command lines and reusable launcher scheme classes;
# auto inserts "python" and/or path to Python executable at front of command line;
# some of this module may assume 'python' is on your system path (see Launcher.py);

subprocess module would work too, but os.popen() uses it internally, and the goal
is to start a program running independently here, not to connect to its streams;
multiprocessing module also is an option, but this is command-lines, not functions:
doesn't make sense to start a process which would just do one of the options here;

new in this edition: runs script filename path through normpath() to change any
/ to \ for Windows tools where required; fix is inherited by PyEdit and others;
on Windows, / is generally allowed for file opens, but not by all launcher tools;
#####
"""

import sys, os
pyfile = (sys.platform[:3] == 'win' and 'python.exe') or 'python'
pypath = sys.executable      # use sys in newer pys

def fixWindowsPath(cmdline):
    """
    change all / to \ in script filename path at front of cmdline;
    used only by classes which run tools that require this on Windows;
    on other platforms, this does not hurt (e.g., os.system on Unix);
    """
    splitline = cmdline.lstrip().split(' ')          # split on spaces
    fixedpath = os.path.normpath(splitline[0])        # fix forward slashes
    return ' '.join([fixedpath] + splitline[1:])      # put it back together

class LaunchMode:
    """
    on call to instance, announce label and run command;
    subclasses format command lines as required in run();
    command should begin with name of the Python script
    file to run, and not with "python" or its full path;
    """
    def __init__(self, label, command):
        self.what = label
        self.where = command
    def __call__(self):                      # on call, ex: button press callback
        self.announce(self.what)
        self.run(self.where)                 # subclasses must define run()
    def announce(self, text):                # subclasses may redefine announce()
```

```

        print(text)                      # methods instead of if/elif logic
def run(self, cmdline):
    assert False, 'run must be defined'

class System(LaunchMode):
"""
    run Python script named in shell command line
    caveat: may block caller, unless & added on Unix
"""

def run(self, cmdline):
    cmdline = fixWindowsPath(cmdline)
    os.system('%s %s' % (pypath, cmdline))

class Popen(LaunchMode):
"""
    run shell command line in a new process
    caveat: may block caller, since pipe closed too soon
"""

def run(self, cmdline):
    cmdline = fixWindowsPath(cmdline)
    os.popen(pypath + ' ' + cmdline)      # assume nothing to be read

class Fork(LaunchMode):
"""
    run command in explicitly created new process
    for Unix-like systems only, including cygwin
"""

def run(self, cmdline):
    assert hasattr(os, 'fork')
    cmdline = cmdline.split()            # convert string to list
    if os.fork() == 0:                  # start new child process
        os.execvp(pypath, [pyfile] + cmdline) # run new program in child

class Start(LaunchMode):
"""
    run command independent of caller
    for Windows only: uses filename associations
"""

def run(self, cmdline):
    assert sys.platform[:3] == 'win'
    cmdline = fixWindowsPath(cmdline)
    os.startfile(cmdline)

class StartArgs(LaunchMode):
"""
    for Windows only: args may require real start
    forward slashes are okay here
"""

def run(self, cmdline):
    assert sys.platform[:3] == 'win'
    os.system('start ' + cmdline)        # may create pop-up window

class Spawn(LaunchMode):
"""
    run python in new process independent of caller

```

```

for Windows or Unix; use P_NOWAIT for dos box;
forward slashes are okay here
"""
def run(self, cmdline):
    os.spawnv(os.P_DETACH, pypath, (pyfile, cmdline))

class Top_level(LaunchMode):
    """
    run in new window, same process
    tbd: requires GUI class info too
    """
    def run(self, cmdline):
        assert False, 'Sorry - mode not yet implemented'

    #
    # pick a "best" launcher for this platform
    # may need to specialize the choice elsewhere
    #

    if sys.platform[:3] == 'win':
        PortableLauncher = Spawn
    else:
        PortableLauncher = Fork

    class QuietPortableLauncher(PortableLauncher):
        def announce(self, text):
            pass

        def selftest():
            file = 'echo.py'
            input('default mode...')
            launcher = PortableLauncher(file, file)
            launcher()                                # no block

            input('system mode...')
            System(file, file)()                      # blocks

            if sys.platform[:3] == 'win':
                input('DOS start mode...')
                StartArgs(file, file)()                # no block

    if __name__ == '__main__': selftest()

```

Near the end of the file, the module picks a default class based on the `sys.platform` attribute: `PortableLauncher` is set to a class that uses `spawnv` on Windows and one that uses the `fork/exec` combination elsewhere; in recent Pythons, we could probably just use the `spawnv` scheme on most platforms, but the alternatives in this module are used in additional contexts. If you import this module and always use its `PortableLauncher` attribute, you can forget many of the platform-specific details enumerated in this chapter.

To run a Python program, simply import the `PortableLauncher` class, make an instance by passing a label and command line (without a leading “python” word), and then call

the instance object as though it were a function. The program is started by a *call* operation—by its `__call__` operator-overloading method, instead of a normally named method—so that the classes in this module can also be used to generate callback handlers in tkinter-based GUIs. As we'll see in the upcoming chapters, button-presses in tkinter invoke a callable object with no arguments; by registering a `PortableLauncher` instance to handle the press event, we can automatically start a new program from another program's GUI. A GUI might associate a launcher with a GUI's button press with code like this:

```
Button(root, text=name, command=PortableLauncher(name, commandLine))
```

When run standalone, this module's `selftest` function is invoked as usual. As coded, `System` blocks the caller until the program exits, but `PortableLauncher` (really, `Spawn` or `Fork`) and `Start` do not:

```
C:\...\PP4E> type echo.py
print('Spam')
input('press Enter')

C:\...\PP4E> python launchmodes.py
default mode...
echo.py
system mode...
echo.py
Spam
press Enter
DOS start mode...
echo.py
```

As more practical applications, this file is also used in [Chapter 8](#) to launch GUI dialog demos independently, and again in a number of [Chapter 10](#)'s examples, including PyDemos and PyGadgets—launcher scripts designed to run major examples in this book in a portable fashion, which live at the top of this book's examples distribution directory. Because these launcher scripts simply import `PortableLauncher` and register instances to respond to GUI events, they run on both Windows and Unix unchanged (tkinter's portability helps, too, of course). The PyGadgets script even customizes `PortableLauncher` to update a GUI label at start time:

```
class Launcher(launchmodes.PortableLauncher):      # use wrapped launcher class
    def announce(self, text):                      # customize to set GUI label
        Info.config(text=text)
```

We'll explore these two client scripts, and others, such as [Chapter 11](#)'s PyEdit after we start coding GUIs in [Part III](#). Partly because of its role in PyEdit, this edition extends this module to automatically replace forward slashes with *backward slashes* in the script's file path name. PyEdit uses forward slashes in some filenames because they are allowed in file opens on Windows, but some Windows launcher tools require the backslash form instead. Specifically, `system`, `popen`, and `startfile` in `os` require backslashes, but `spawnv` does not. PyEdit and others inherit the new pathname fix of `fixWindowsPath` here simply by importing and using this module's classes; PyEdit

eventually changed so as to make this fix irrelevant for its own use case (see [Chapter 11](#)), but other clients still acquire the fix for free.

Also notice how some of the classes in this example use the `sys.executable` path string to obtain the Python executable’s full path name. This is partly due to their role in user-friendly demo launchers. In prior versions that predated `sys.executable`, these classes instead called two functions exported by a module named *Launcher.py* to find a suitable Python executable, regardless of whether the user had added its directory to the system PATH variable’s setting.

This search is no longer required. Since I’ll describe this module’s other roles in the next chapter, and since this search has been largely precluded by Python’s perpetual pandering to programmers’ professional proclivities, I’ll postpone any pointless pedagogical presentation here. (Period.)

Other System Tools Coverage

That concludes our tour of Python system tools. In this and the prior three chapters, we’ve met most of the commonly used system tools in the Python library. Along the way, we’ve also learned how to use them to do useful things such as start programs, process directories, and so on. The next chapter wraps up this domain by using the tools we’ve just met to implement scripts that do useful and more realistic system-level work.

Still other system-related tools in Python appear later in this text. For instance:

- Sockets, used to communicate with other programs and networks and introduced briefly here, show up again in [Chapter 10](#) in a common GUI use case and are covered in full in [Chapter 12](#).
- Select calls, used to multiplex among tasks, are also introduced in [Chapter 12](#) as a way to implement servers.
- File locking with `os.open`, introduced in [Chapter 4](#), is discussed again in conjunction with later examples.
- Regular expressions, string pattern matching used by many text processing tools in the system administration domain, don’t appear until [Chapter 19](#).

Moreover, things like forks and threads are used extensively in the Internet scripting chapters: see the discussion of threaded GUIs in Chapters [9](#) and [10](#); the server implementations in [Chapter 12](#); the FTP client GUI in [Chapter 13](#); and the PyMailGUI program in [Chapter 14](#). Along the way, we’ll also meet higher-level Python modules, such as `socketserver`, which implement fork and thread-based socket server code for us. In fact, many of the last four chapters’ tools will pop up constantly in later examples in this book—about what one would expect of general-purpose portable libraries.

Last, but not necessarily least, I'd like to point out one more time that many additional tools in the Python library don't appear in this book at all. With hundreds of library modules, more appearing all the time, and even more in the third-party domain, Python book authors have to pick and choose their topics frugally! As always, be sure to browse the Python library manuals and Web early and often in your Python career.

Complete System Programs

“The Greps of Wrath”

This chapter wraps up our look at the system interfaces domain in Python by presenting a collection of larger Python scripts that do real systems work—comparing and copying directory trees, splitting files, searching files and directories, testing other programs, configuring launched programs’ shell environments, and so on. The examples here are Python system utility programs that illustrate typical tasks and techniques in this domain and focus on applying built-in tools, such as file and directory tree processing.

Although the main point of this case-study chapter is to give you a feel for realistic scripts in action, the size of these examples also gives us an opportunity to see Python’s support for development paradigms like object-oriented programming (OOP) and reuse at work. It’s really only in the context of nontrivial programs such as the ones we’ll meet here that such tools begin to bear tangible fruit. This chapter also emphasizes the “why” of system tools, not just the “how”; along the way, I’ll point out real-world needs met by the examples we’ll study, to help you put the details in context.

One note up front: this chapter moves quickly, and a few of its examples are largely listed just for independent study. Because all the scripts here are heavily documented and use Python system tools described in the preceding chapters, I won’t go through all the code in exhaustive detail. You should read the source code listings and experiment with these programs on your own computer to get a better feel for how to combine system interfaces to accomplish realistic tasks. All are available in source code form in the book’s examples distribution and most work on all major platforms.

I should also mention that most of these are programs I have really used, not examples written just for this book. They were coded over a period of years and perform widely differing tasks, so there is no obvious common thread to connect the dots here other than need. On the other hand, they help explain why system tools are useful in the first place, demonstrate larger development concepts that simpler examples cannot, and bear collective witness to the simplicity and portability of automating system tasks with Python. Once you’ve mastered the basics, you’ll wish you had done so sooner.

A Quick Game of “Find the Biggest Python File”

Quick: what’s the biggest Python source file on your computer? This was the query innocently posed by a student in one of my Python classes. Because I didn’t know either, it became an official exercise in subsequent classes, and it provides a good example of ways to apply Python system tools for a realistic purpose in this book. Really, the query is a bit vague, because its scope is unclear. Do we mean the largest Python file in a directory, in a full directory tree, in the standard library, on the module import search path, or on your entire hard drive? Different scopes imply different solutions.

Scanning the Standard Library Directory

For instance, [Example 6-1](#) is a first-cut solution that looks for the biggest Python file in one directory—a limited scope, but enough to get started.

Example 6-1. PP4E\System\Filetools\bigpy-dir.py

```
"""
Find the largest Python source file in a single directory.
Search Windows Python source lib, unless dir command-line arg.
"""

import os, glob, sys
dirname = r'C:\Python31\Lib' if len(sys.argv) == 1 else sys.argv[1]

allsizes = []
alppy = glob.glob(dirname + os.sep + '*.py')
for filename in alppy:
    filesize = os.path.getsize(filename)
    allsizes.append((filesize, filename))

allsizes.sort()
print(allsizes[:2])
print(allsizes[-2:])
```

This script uses the `glob` module to run through a directory’s files and detects the largest by storing sizes and names on a list that is sorted at the end—because size appears first in the list’s tuples, it will dominate the ascending value sort, and the largest percolates to the end of the list. We could instead keep track of the currently largest as we go, but the list scheme is more flexible. When run, this script scans the Python standard library’s source directory on Windows, unless you pass a different directory on the command line, and it prints both the two smallest and largest files it finds:

```
C:\...\PP4E\System\Filetools> bigpy-dir.py
[(0, 'C:\\Python31\\Lib\\build_class.py'), (56, 'C:\\Python31\\Lib\\struct.py')]
[(147086, 'C:\\Python31\\Lib\\turtle.py'), (211238, 'C:\\Python31\\Lib\\decimal.
py')]

C:\...\PP4E\System\Filetools> bigpy-dir.py .
[(21, './__init__.py'), (461, './bigpy-dir.py')]
```

```
[(1940, '..\\bigext-tree.py'), (2547, '..\\split.py')]

C:\...\PP4E\System\Filetools> bigpy-dir.py ..
[(21, '..\\__init__.py'), (29, '..\\testargv.py')]
[(541, '..\\testarg2.py'), (549, '..\\more.py')]
```

Scanning the Standard Library Tree

The prior section's solution works, but it's obviously a partial answer—Python files are usually located in more than one directory. Even within the standard library, there are many subdirectories for module packages, and they may be arbitrarily nested. We really need to traverse an entire directory tree. Moreover, the first output above is difficult to read; Python's `pprint` (for “pretty print”) module can help here. [Example 6-2](#) puts these extensions into code.

Example 6-2. PP4E\System\Filetools\bigpy-tree.py

```
"""
Find the largest Python source file in an entire directory tree.
Search the Python source lib, use pprint to display results nicely.
"""

import sys, os, pprint
trace = False
if sys.platform.startswith('win'):
    dirname = r'C:\Python31\Lib'                      # Windows
else:
    dirname = '/usr/lib/python'                       # Unix, Linux, Cygwin

allsizes = []
for (thisDir, subsHere, filesHere) in os.walk(dirname):
    if trace: print(thisDir)
    for filename in filesHere:
        if filename.endswith('.py'):
            if trace: print('...', filename)
            fullname = os.path.join(thisDir, filename)
            fullsize = os.path.getsize(fullname)
            allsizes.append((fullsize, fullname))

allsizes.sort()
pprint pprint(allsizes[:2])
pprint pprint(allsizes[-2:])
```

When run, this new version uses `os.walk` to search an entire tree of directories for the largest Python source file. Change this script's `trace` variable if you want to track its progress through the tree. As coded, it searches the Python standard library's source tree, tailored for Windows and Unix-like locations:

```
C:\...\PP4E\System\Filetools> bigpy-tree.py
[(0, 'C:\\Python31\\Lib\\build_class.py'),
 (0, 'C:\\Python31\\Lib\\email\\mime\\__init__.py')]
[(211238, 'C:\\Python31\\Lib\\decimal.py'),
 (380582, 'C:\\Python31\\Lib\\pydoc_data\\topics.py')]
```

Scanning the Module Search Path

Sure enough—the prior section’s script found smallest and largest files in subdirectories. While searching Python’s entire standard library tree this way is more inclusive, it’s still incomplete: there may be additional modules installed elsewhere on your computer, which are accessible from the module import search path but outside Python’s source tree. To be more exhaustive, we could instead essentially perform the same tree search, but for every directory on the module import search path. [Example 6-3](#) adds this extension to include every importable Python-coded module on your computer—located both on the path directly and nested in package directory trees.

Example 6-3. PP4E\System\Filetools\bigpy-path.py

```
"""
Find the largest Python source file on the module import search path.
Skip already-visited directories, normalize path and case so they will
match properly, and include line counts in pprinted result. It's not
enough to use os.environ['PYTHONPATH']: this is a subset of sys.path.
"""

import sys, os, pprint
trace = 0 # 1=dirs, 2+=files

visited = {}
allsizes = []
for srcdir in sys.path:
    for (thisDir, subsHere, filesHere) in os.walk(srcdir):
        if trace > 0: print(thisDir)
        thisDir = os.path.normpath(thisDir)
        fixcase = os.path.normcase(thisDir)
        if fixcase in visited:
            continue
        else:
            visited[fixcase] = True
        for filename in filesHere:
            if filename.endswith('.py'):
                if trace > 1: print('...', filename)
                pypath = os.path.join(thisDir, filename)
                try:
                    pysize = os.path.getsize(pypath)
                except os.error:
                    print('skipping', pypath, sys.exc_info()[0])
                else:
                    pylines = len(open(pypath, 'rb').readlines())
                    allsizes.append((pysize, pylines, pypath))

print('By size...')
allsizes.sort()
pprint.pprint(allsizes[:3])
pprint.pprint(allsizes[-3:])

print('By lines...')
allsizes.sort(key=lambda x: x[1])
```

```
pprint.pprint(allsizes[:3])
pprint.pprint(allsizes[-3:])
```

When run, this script marches down the module import path and, for each valid directory it contains, attempts to search the entire tree rooted there. In fact, it nests loops three deep—for items on the path, directories in the item’s tree, and files in the directory. Because the module path may contain directories named in arbitrary ways, along the way this script must take care to:

- Normalize directory paths—fixing up slashes and dots to map directories to a common form.
- Normalize directory name case—converting to lowercase on case-insensitive Windows, so that same names match by string equality, but leaving case unchanged on Unix, where it matters.
- Detect repeats to avoid visiting the same directory twice (the same directory might be reached from more than one entry on `sys.path`).
- Skip any file-like item in the tree for which `os.path.getsize` fails (by default `os.walk` itself silently ignores things it cannot treat as directories, both at the top of and within the tree).
- Avoid potential *Unicode decoding errors* in file content by opening files in binary mode in order to count their lines. Text mode requires decodable content, and some files in Python 3.1’s library tree cannot be decoded properly on Windows. Catching Unicode exceptions with a `try` statement would avoid program exits, too, but might skip candidate files.

This version also adds line counts; this might add significant run time to this script too, but it’s a useful metric to report. In fact, this version uses this value as a sort key to report the three largest and smallest files by line counts too—this may differ from results based upon raw file size. Here’s the script in action in Python 3.1 on my Windows 7 machine; since these results depend on platform, installed extensions, and path settings, your `sys.path` and largest and smallest files may vary:

```
C:\...\PP4E\System\Filetools> bigpy-path.py
By size...
[(0, 0, 'C:\\Python31\\lib\\build_class.py'),
 (0, 0, 'C:\\Python31\\lib\\email\\mime\\__init__.py'),
 (0, 0, 'C:\\Python31\\lib\\email\\test\\__init__.py')]
[(161613, 3754, 'C:\\Python31\\lib\\tkinter\\__init__.py'),
 (211238, 5768, 'C:\\Python31\\lib\\decimal.py'),
 (380582, 78, 'C:\\Python31\\lib\\pydoc_data\\topics.py')]
By lines...
[(0, 0, 'C:\\Python31\\lib\\build_class.py'),
 (0, 0, 'C:\\Python31\\lib\\email\\mime\\__init__.py'),
 (0, 0, 'C:\\Python31\\lib\\email\\test\\__init__.py')]
[(147086, 4132, 'C:\\Python31\\lib\\turtle.py'),
 (150069, 4268, 'C:\\Python31\\lib\\test\\test_descr.py'),
 (211238, 5768, 'C:\\Python31\\lib\\decimal.py')]
```

Again, change this script’s `trace` variable if you want to track its progress through the tree. As you can see, the results for largest files differ when viewed by size and lines—a disparity which we’ll probably have to hash out in our next requirements meeting.

Scanning the Entire Machine

Finally, although searching trees rooted in the module import path normally includes every Python source file you can import on your computer, it’s still not complete. Technically, this approach checks only modules; Python source files which are top-level scripts run directly do not need to be included in the module path. Moreover, the module search path may be manually changed by some scripts dynamically at runtime (for example, by direct `sys.path` updates in scripts that run on web servers) to include additional directories that [Example 6-3](#) won’t catch.

Ultimately, finding the largest source file on your computer requires searching your entire drive—a feat which our tree searcher in [Example 6-2](#) *almost* supports, if we generalize it to accept the root directory name as an argument and add some of the bells and whistles of the path searcher version (we really want to avoid visiting the same directory twice if we’re scanning an entire machine, and we might as well skip errors and check line-based sizes if we’re investing the time). [Example 6-4](#) implements such general tree scans, outfitted for the heavier lifting required for scanning drives.

Example 6-4. PP4E\System\Filetools\bigext-tree.py

```
"""
Find the largest file of a given type in an arbitrary directory tree.
Avoid repeat paths, catch errors, add tracing and line count size.
Also uses sets, file iterators and generator to avoid loading entire
file, and attempts to work around undecodable dir/file name prints.
"""

import os, pprint
from sys import argv, exc_info

trace = 1                                # 0=off, 1=dirs, 2=+files
dirname, extname = os.curdir, '.py'          # default is .py files in cwd
if len(argv) > 1: dirname = argv[1]          # ex: C:\, C:\Python31\Lib
if len(argv) > 2: extname = argv[2]          # ex: .pyw, .txt
if len(argv) > 3: trace   = int(argv[3])    # ex: ". .py 2"

def tryprint(arg):
    try:
        print(arg)                          # unprintable filename?
    except UnicodeEncodeError:
        print(arg.encode())                # try raw byte string

visited  = set()
allsizes = []
for (thisDir, subsHere, filesHere) in os.walk(dirname):
    if trace: tryprint(thisDir)
    thisDir = os.path.normpath(thisDir)
```

```

fixname = os.path.normcase(thisDir)
if fixname in visited:
    if trace: tryprint('skipping ' + thisDir)
else:
    visited.add(fixname)
    for filename in filesHere:
        if filename.endswith(extname):
            if trace > 1: tryprint('+++' + filename)
            fullname = os.path.join(thisDir, filename)
            try:
                bytesize = os.path.getsize(fullname)
                linesize = sum(+1 for line in open(fullname, 'rb'))
            except Exception:
                print('error', exc_info()[0])
            else:
                allsizes.append((bytesize, linesize, fullname))

for (title, key) in [('bytes', 0), ('lines', 1)]:
    print('\nBy %s...' % title)
    allsizes.sort(key=lambda x: x[key])
    pprint.pprint(allsizes[:3])
    pprint.pprint(allsizes[-3:])

```

Unlike the prior tree version, this one allows us to search in specific directories, and for specific extensions. The default is to simply search the current working directory for Python files:

```

C:\...\PP4E\System\Filetools> bigext-tree.py
.

By bytes...
[(21, 1, '..\\__init__.py'),
 (461, 17, '..\\bigpy-dir.py'),
 (818, 25, '..\\bigpy-tree.py')]
[(1696, 48, '..\\join.py'),
 (1940, 49, '..\\bigext-tree.py'),
 (2547, 57, '..\\split.py')]

By lines...
[(21, 1, '..\\__init__.py'),
 (461, 17, '..\\bigpy-dir.py'),
 (818, 25, '..\\bigpy-tree.py')]
[(1696, 48, '..\\join.py'),
 (1940, 49, '..\\bigext-tree.py'),
 (2547, 57, '..\\split.py')]

```

For more custom work, we can pass in a directory name, extension type, and trace level on the command-line now (trace level 0 disables tracing, and 1, the default, shows directories visited along the way):

```

C:\...\PP4E\System\Filetools> bigext-tree.py .. .py 0

By bytes...
[(21, 1, '..\\__init__.py'),
 (21, 1, '..\\Filetools\\__init__.py'),

```

```
(28, 1, '..\\Streams\\hello-out.py')]  
[(2278, 67, '..\\Processes\\multi2.py'),  
(2547, 57, '..\\Filetools\\split.py'),  
(4361, 105, '..\\Tester\\tester.py')]
```

By lines...

```
[(21, 1, '..\\__init__.py'),  
(21, 1, '..\\Filetools\\__init__.py'),  
(28, 1, '..\\Streams\\hello-out.py')]  
[(2547, 57, '..\\Filetools\\split.py'),  
(2278, 67, '..\\Processes\\multi2.py'),  
(4361, 105, '..\\Tester\\tester.py')]
```

This script also lets us scan for different file types; here it is picking out the smallest and largest text file from one level up (at the time I ran this script, at least):

```
C:\\...\\PP4E\\System\\Filetools> bigext-tree.py ... .txt 1
```

```
..  
..\\Environment  
..\\Filetools  
..\\Processes  
..\\Streams  
..\\Tester  
..\\Tester\\Args  
..\\Tester\\Errors  
..\\Tester\\Inputs  
..\\Tester\\Outputs  
..\\Tester\\Scripts  
..\\Tester\\xxold  
..\\Threads
```

By bytes...

```
[(4, 2, '..\\Streams\\input.txt'),  
(13, 1, '..\\Streams\\hello-in.txt'),  
(20, 4, '..\\Streams\\data.txt')]  
[(104, 4, '..\\Streams\\output.txt'),  
(172, 3, '..\\Tester\\xxold\\README.txt.txt'),  
(435, 4, '..\\Filetools\\temp.txt')]
```

By lines...

```
[(13, 1, '..\\Streams\\hello-in.txt'),  
(22, 1, '..\\spam.txt'),  
(4, 2, '..\\Streams\\input.txt')]  
[(20, 4, '..\\Streams\\data.txt'),  
(104, 4, '..\\Streams\\output.txt'),  
(435, 4, '..\\Filetools\\temp.txt')]
```

And now, to search your entire system, simply pass in your machine's root directory name (use / instead of C:\ on Unix-like machines), along with an optional file extension type (.py is just the default now). The winner is...(please, no wagering):

```
C:\\...\\PP4E\\dev\\Examples\\PP4E\\System\\Filetools> bigext-tree.py C:\\  
C:\\  
C:\\$Recycle.Bin  
C:\\$Recycle.Bin\\S-1-5-21-3951091421-2436271001-910485044-1004  
C:\\cygwin
```

```
C:\cygwin\bin  
C:\cygwin\cygdrive  
C:\cygwin\dev  
C:\cygwin\dev\mqueue  
C:\cygwin\dev\shm  
C:\cygwin\etc  
...MANY more lines omitted...
```

By bytes...

```
[(0, 0, 'C:\\cygwin\\\\...\\python31\\Python-3.1.1\\Lib\\build_class.py'),  
(0, 0, 'C:\\cygwin\\\\...\\python31\\Python-3.1.1\\Lib\\email\\mime\\__init__.py'),  
(0, 0, 'C:\\cygwin\\\\...\\python31\\Python-3.1.1\\Lib\\email\\test\\__init__.py')]  
[(380582, 78, 'C:\\Python31\\Lib\\pydoc_data\\topics.py'),  
(398157, 83, 'C:\\...\\Install\\Source\\Python-2.6\\Lib\\pydoc_topics.py'),  
(412434, 83, 'C:\\Python26\\Lib\\pydoc_topics.py')]
```

By lines...

```
[(0, 0, 'C:\\cygwin\\\\...\\python31\\Python-3.1.1\\Lib\\build_class.py'),  
(0, 0, 'C:\\cygwin\\\\...\\python31\\Python-3.1.1\\Lib\\email\\mime\\__init__.py'),  
(0, 0, 'C:\\cygwin\\\\...\\python31\\Python-3.1.1\\Lib\\email\\test\\__init__.py')]  
[(204107, 5589, 'C:\\...\\Install\\Source\\Python-3.0\\Lib\\decimal.py'),  
(205470, 5768, 'C:\\cygwin\\\\...\\python31\\Python-3.1.1\\Lib\\decimal.py'),  
(211238, 5768, 'C:\\Python31\\Lib\\decimal.py')]
```

The script’s trace logic is preset to allow you to monitor its directory progress. I’ve shortened some directory names to protect the innocent here (and to fit on this page). This command may take a *long time* to finish on your computer—on my sadly under-powered Windows 7 netbook, it took 11 minutes to scan a solid state drive with some 59G of data, 200K files, and 25K directories when the system was lightly loaded (8 minutes when not tracing directory names, but half an hour when many other applications were running). Nevertheless, it provides the most exhaustive solution to the original query of all our attempts.

This is also as complete a solution as we have space for in this book. For more fun, consider that you may need to scan more than one drive, and some Python source files may also appear in zip archives, both on the module path or not (`os.walk` silently ignores zip files in [Example 6-3](#)). They might also be named in other ways—with `.pyw` extensions to suppress shell pop ups on Windows, and with arbitrary extensions for some top-level scripts. In fact, top-level scripts might have no filename extension at all, even though they are Python source files. And while they’re generally not Python files, some importable modules may also appear in frozen binaries or be statically linked into the Python executable. In the interest of space, we’ll leave such higher resolution (and potentially intractable!) search extensions as suggested exercises.

Printing Unicode Filenames

One fine point before we move on: notice the seemingly superfluous exception handling in [Example 6-4](#)’s `tryprint` function. When I first tried to scan an entire drive as shown in the preceding section, this script died on a Unicode encoding error while trying to

print a directory name of a saved web page. Adding the exception handler skips the error entirely.

This demonstrates a subtle but pragmatically important issue: Python 3.X's Unicode orientation extends to filenames, even if they are just printed. As we learned in [Chapter 4](#), because filenames may contain arbitrary text, `os.listdir` returns filenames in two different ways—we get back decoded Unicode strings when we pass in a normal `str` argument, and still-encoded byte strings when we send a `bytes`:

```
>>> import os
>>> os.listdir('.')
['bigext-tree.py', 'bigpy-dir.py', 'bigpy-path.py', 'bigpy-tree.py']

>>> os.listdir(b'.')[4]
[b'bigext-tree.py', b'bigpy-dir.py', b'bigpy-path.py', b'bigpy-tree.py']
```

Both `os.walk` (used in the [Example 6-4](#) script) and `glob.glob` inherit this behavior for the directory and file names they return, because they work by calling `os.listdir` internally at each directory level. For all these calls, passing in a byte string argument suppresses Unicode decoding of file and directory names. Passing a normal string assumes that filenames are decodable per the file system's Unicode scheme.

The reason this potentially mattered to this section's example is that running the tree search version over an entire hard drive eventually reached an undecodable filename (an old saved web page with an odd name), which generated an exception when the `print` function tried to display it. Here's a simplified recreation of the error, run in a shell window (Command Prompt) on Windows:

```
>>> root = r'C:\py3000'
>>> for (dir, subs, files) in os.walk(root): print(dir)
...
C:\py3000
C:\py3000\FutureProofPython - PythonInfo Wiki_files
C:\py3000\Oakwinter_com Code » Porting setuptools to py3k_files
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "C:\Python31\lib\encodings\cp437.py", line 19, in encode
        return codecs.charmap_encode(input,self.errors,encoding_map)[0]
UnicodeEncodeError: 'charmap' codec can't encode character '\u2019' in position
45: character maps to <undefined>
```

One way out of this dilemma is to use `bytes` strings for the directory root name—this suppresses filename decoding in the `os.listdir` calls run by `os.walk`, and effectively limits the scope of later printing to raw bytes. Since printing does not have to deal with encodings, it works without error. Manually encoding to bytes prior to printing works too, but the results are slightly different:

```
>>> root.encode()
b'C:\\py3000'
>>> for (dir, subs, files) in os.walk(root.encode()): print(dir)
...
```

```

b'C:\\py3000'
b'C:\\py3000\\FutureProofPython - PythonInfo Wiki_files'
b'C:\\py3000\\Oakwinter_com  Code \\xbb Porting setuptools to py3k_files'
b'C:\\py3000\\What\\x92s New in Python 3_0 \\x97 Python Documentation'

>>> for (dir, subs, files) in os.walk(root): print(dir.encode())
...
b'C:\\py3000'
b'C:\\py3000\\FutureProofPython - PythonInfo Wiki_files'
b'C:\\py3000\\Oakwinter_com  Code \\xc2\\xbb Porting setuptools to py3k_files'
b'C:\\py3000\\What\\xe2\\x80\\x99s New in Python 3_0 \\xe2\\x80\\x94 Python Documentation'

```

Unfortunately, either approach means that all the directory names printed during the walk display as cryptic byte strings. To maintain the better readability of normal strings, I instead opted for the exception handler approach used in the script’s code. This avoids the issues entirely:

```

>>> for (dir, subs, files) in os.walk(root):
...     try:
...         print(dir)
...     except UnicodeEncodeError:
...         print(dir.encode())           # or simply punt if encode may fail too
...
C:\\py3000
C:\\py3000\\FutureProofPython - PythonInfo Wiki_files
C:\\py3000\\Oakwinter_com  Code » Porting setuptools to py3k_files
b'C:\\py3000\\What\\xe2\\x80\\x99s New in Python 3_0 \\xe2\\x80\\x94 Python Documentation'

```

Oddly, though, the error seems more related to printing than to Unicode encodings of filenames—because the filename did not fail until printed, it must have been decodable when its string was created initially. That’s why wrapping up the `print` in a `try` suffices; otherwise, the error would occur earlier.

Moreover, this error does not occur if the script’s output is redirected to a file, either at the shell level (`bigext-tree.py c:\\ > out`), or by the `print` call itself (`print(dir, file=F)`). In the latter case the output file must later be read back in binary mode, as text mode triggers the same error when printing the file’s content to the shell window (but again, not until printed). In fact, the exact same code that fails when run in a system shell Command Prompt on Windows works without error when run in the IDLE GUI on the same platform—the tkinter GUI used by IDLE handles display of characters that printing to standard output connected to a shell terminal window does not:

```

>>> import os                  # run in IDLE (a tkinter GUI), not system shell
>>> root = r'C:\\py3000'
>>> for (dir, subs, files) in os.walk(root): print(dir)

C:\\py3000
C:\\py3000\\FutureProofPython - PythonInfo Wiki_files
C:\\py3000\\Oakwinter_com  Code » Porting setuptools to py3k_files
C:\\py3000\\What's New in Python 3_0 – Python Documentation_files

```

In other words, the exception occurs only when printing to a shell window, and long after the file name string is created. This reflects an artifact of extra translations

performed by the Python printer, not of Unicode file names in general. Because we have no room for further exploration here, though, we'll have to be satisfied with the fact that our exception handler sidesteps the printing problem altogether. You should still be aware of the implications of Unicode filename decoding, though; on some platforms you may need to pass byte strings to `os.walk` in this script to prevent decoding errors as filenames are created.*

Since Unicode is still relatively new in 3.1, be sure to test for such errors on your computer and your Python. Also see also Python's manuals for more on the treatment of Unicode filenames, and the text *Learning Python* for more on Unicode in general. As noted earlier, our scripts also had to open text files in binary mode because some might contain undecodable *content* too. It might seem surprising that Unicode issues can crop up in basic printing like this too, but such is life in the brave new Unicode world. Many real-world scripts don't need to care much about Unicode, of course—including those we'll explore in the next section.

Splitting and Joining Files

Like most kids, mine spent a lot of time on the Internet when they were growing up. As far as I could tell, it was the thing to do. Among their generation, computer geeks and gurus seem to have been held in the same sort of esteem that my generation once held rock stars. When kids disappeared into their rooms, chances were good that they were hacking on computers, not mastering guitar riffs (well, real ones, at least). It may or may not be healthier than some of the diversions of my own misspent youth, but that's a topic for another kind of book.

Despite the rhetoric of techno-pundits about the Web's potential to empower an upcoming generation in ways unimaginable by their predecessors, my kids seemed to spend most of their time playing games. To fetch new ones in my house at the time, they had to download to a shared computer which had Internet access and transfer those games to their own computers to install. (Their own machines did not have Internet access until later, for reasons that most parents in the crowd could probably expand upon.)

The problem with this scheme is that game files are not small. They were usually much too big to fit on a floppy or memory stick of the time, and burning a CD or DVD took away valuable game-playing time. If all the machines in my house ran Linux, this would have been a nonissue. There are standard command-line programs on Unix for chopping a file into pieces small enough to fit on a transfer device (`split`), and others for

* For a related `print` issue, see [Chapter 14](#)'s workaround for program aborts when printing stack tracebacks to standard output from spawned programs. Unlike the problem described here, that issue does not appear to be related to Unicode characters that may be unprintable in shell windows but reflects another regression for standard output prints in general in Python 3.1, which may or may not be repaired by the time you read this text. See also the Python environment variable `PYTHONIOENCODING`, which can override the default encoding used for standard streams.

putting the pieces back together to re-create the original file (`cat`). Because we had all sorts of different machines in the house, though, we needed a more portable solution.[†]

Splitting Files Portably

Since all the computers in my house ran Python, a simple portable Python script came to the rescue. The Python program in [Example 6-5](#) distributes a single file's contents among a set of part files and stores those part files in a directory.

Example 6-5. PP4E\System\Filetools\split.py

```
#!/usr/bin/python
"""
#####
split a file into a set of parts; join.py puts them back together;
this is a customizable version of the standard Unix split command-line
utility; because it is written in Python, it also works on Windows and
can be easily modified; because it exports a function, its logic can
also be imported and reused in other applications;
#####
"""

import sys, os
kilobytes = 1024
megabytes = kilobytes * 1000
chunksize = int(1.4 * megabytes)                      # default: roughly a floppy

def split(fromfile, todir, chunksize=chunksize):
    if not os.path.exists(todir):                         # caller handles errors
        os.mkdir(todir)                                   # make dir, read/write parts
    else:
        for fname in os.listdir(todir):                   # delete any existing files
            os.remove(os.path.join(todir, fname))
    partnum = 0
    input = open(fromfile, 'rb')                          # binary: no decode, endline
    while True:                                         # eof=empty string from read
        chunk = input.read(chunksize)                   # get next part <= chunksize
        if not chunk: break
        partnum += 1
        filename = os.path.join(todir, ('part%04d' % partnum))
        fileobj = open(filename, 'wb')
        fileobj.write(chunk)
        fileobj.close()                                 # or simply open().write()
    input.close()
    assert partnum <= 9999                            # join sort fails if 5 digits
    return partnum
```

[†] I should note that this background story stems from the second edition of this book, written in 2000. Some ten years later, floppies have largely gone the way of the parallel port and the dinosaur. Moreover, burning a CD or DVD is no longer as painful as it once was; there are new options today such as large flash memory cards, wireless home networks, and simple email; and naturally, my home computers configuration isn't what it once was. For that matter, some of my kids are no longer kids (though they've retained some backward compatibility with their former selves).

```

if __name__ == '__main__':
    if len(sys.argv) == 2 and sys.argv[1] == '-help':
        print('Use: split.py [file-to-split target-dir [chunksize]]')
    else:
        if len(sys.argv) < 3:
            interactive = True
            fromfile = input('File to be split? ') # input if clicked
            todir    = input('Directory to store part files? ')
        else:
            interactive = False
            fromfile, todir = sys.argv[1:3] # args in cmdline
            if len(sys.argv) == 4: chunksize = int(sys.argv[3])
        absfrom, absto = map(os.path.abspath, [fromfile, todir])
        print('Splitting', absfrom, 'to', absto, 'by', chunksize)

    try:
        parts = split(fromfile, todir, chunksize)
    except:
        print('Error during split:')
        print(sys.exc_info()[0], sys.exc_info()[1])
    else:
        print('Split finished:', parts, 'parts are in', absto)
    if interactive: input('Press Enter key') # pause if clicked

```

By default, this script splits the input file into chunks that are roughly the size of a floppy disk—perfect for moving big files between the electronically isolated machines of the time. Most importantly, because this is all portable Python code, this script will run on just about any machine, even ones without their own file splitter. All it requires is an installed Python. Here it is at work splitting a Python 3.1 self-installer executable located in the current working directory on Windows (I've omitted a few `dir` output lines to save space here; use `ls -l` on Unix):

```

C:\temp> cd C:\temp

C:\temp> dir python-3.1.msi
...more...
06/27/2009  04:53 PM      13,814,272 python-3.1.msi
              1 File(s)   13,814,272 bytes
              0 Dir(s)  188,826,189,824 bytes free

C:\temp> python C:\...\PP4E\System\Filetools\split.py -help
Use: split.py [file-to-split target-dir [chunksize]]

C:\temp> python C:\...\P4E\System\Filetools\split.py python-3.1.msi pysplit
Splitting C:\temp\python-3.1.msi to C:\temp\pysplit by 1433600
Split finished: 10 parts are in C:\temp\pysplit

C:\temp> dir pysplit
...more...
02/21/2010  11:13 AM    <DIR>      .
02/21/2010  11:13 AM    <DIR>      ..
02/21/2010  11:13 AM          1,433,600 part0001
02/21/2010  11:13 AM          1,433,600 part0002

```

```
02/21/2010 11:13 AM      1,433,600 part0003
02/21/2010 11:13 AM      1,433,600 part0004
02/21/2010 11:13 AM      1,433,600 part0005
02/21/2010 11:13 AM      1,433,600 part0006
02/21/2010 11:13 AM      1,433,600 part0007
02/21/2010 11:13 AM      1,433,600 part0008
02/21/2010 11:13 AM      1,433,600 part0009
02/21/2010 11:13 AM      911,872 part0010
          10 File(s)   13,814,272 bytes
           2 Dir(s)  188,812,328,960 bytes free
```

Each of these generated part files represents one binary chunk of the file *python-3.1.msi*—a chunk small enough to fit comfortably on a floppy disk of the time. In fact, if you add the sizes of the generated part files given by the `ls` command, you’ll come up with exactly the same number of bytes as the original file’s size. Before we see how to put these files back together again, here are a few points to ponder as you study this script’s code:

Operation modes

This script is designed to input its parameters in either *interactive* or *command-line* mode; it checks the number of command-line arguments to find out the mode in which it is being used. In command-line mode, you list the file to be split and the output directory on the command line, and you can optionally override the default part file size with a third command-line argument.

In interactive mode, the script asks for a filename and output directory at the console window with `input` and pauses for a key press at the end before exiting. This mode is nice when the program file is started by clicking on its icon; on Windows, parameters are typed into a pop-up DOS box that doesn’t automatically disappear. The script also shows the absolute paths of its parameters (by running them through `os.path.abspath`) because they may not be obvious in interactive mode.

Binary file mode

This code is careful to open both input and output files in binary mode (`rb`, `wb`), because it needs to portably handle things like executables and audio files, not just text. In [Chapter 4](#), we learned that on Windows, text-mode files automatically map `\r\n` end-of-line sequences to `\n` on input and map `\n` to `\r\n` on output. For true binary data, we really don’t want any `\r` characters in the data to go away when read, and we don’t want any superfluous `\r` characters to be added on output. Binary-mode files suppress this `\r` mapping when the script is run on Windows and so avoid data corruption.

In Python 3.X, binary mode also means that file data is `bytes` objects in our script, not encoded `str` text, though we don’t need to do anything special—this script’s file processing code runs the same on Python 3.X as it did on 2.X. In fact, binary mode is required in 3.X for this program, because the target file’s data may not be encoded text at all; text mode requires that file content must be decodable in 3.X, and that might fail both for truly binary data and text files obtained from other

platforms. On output, binary mode accepts bytes and suppresses Unicode encoding and line-end translations.

Manually closing files

This script also goes out of its way to manually close its files. As we also saw in [Chapter 4](#), we can often get by with a single line: `open(partname, 'wb').write(chunk)`. This shorter form relies on the fact that the current Python implementation automatically closes files for you when file objects are reclaimed (i.e., when they are garbage collected, because there are no more references to the file object). In this one-liner, the file object would be reclaimed immediately, because the `open` result is temporary in an expression and is never referenced by a longer-lived name. Similarly, the `input` file is reclaimed when the `split` function exits.

However, it's not impossible that this automatic-close behavior may go away in the future. Moreover, the Jython Java-based Python implementation does not reclaim unreferenced objects as immediately as the standard Python. You should close manually if you care about the Java port, your script may potentially create many files in a short amount of time, and it may run on a machine that has a limit on the number of open files per program. Because the `split` function in this module is intended to be a general-purpose tool, it accommodates such worst-case scenarios. Also see [Chapter 4](#)'s mention of the file context manager and the `with` statement; this provides an alternative way to guarantee file closes.

Joining Files Portably

Back to moving big files around the house: after downloading a big game program file, you can run the previous splitter script by clicking on its name in Windows Explorer and typing filenames. After a split, simply copy each part file onto its own floppy (or other more modern medium), walk the files to the destination machine, and re-create the split output directory on the target computer by copying the part files. Finally, the script in [Example 6-6](#) is clicked or otherwise run to put the parts back together.

Example 6-6. PP4E\System\Filetools\join.py

```
#!/usr/bin/python
"""
#####
join all part files in a dir created by split.py, to re-create file.
This is roughly like a 'cat fromdir/* > tofile' command on unix, but is
more portable and configurable, and exports the join operation as a
reusable function. Relies on sort order of filenames: must be same
length. Could extend split/join to pop up Tkinter file selectors.
#####
import os, sys
readsize = 1024
```

```

def join(fromdir, tofile):
    output = open(tofile, 'wb')
    parts = os.listdir(fromdir)
    parts.sort()
    for filename in parts:
        filepath = os.path.join(fromdir, filename)
        fileobj = open(filepath, 'rb')
        while True:
            filebytes = fileobj.read(readsize)
            if not filebytes: break
            output.write(filebytes)
        fileobj.close()
    output.close()

if __name__ == '__main__':
    if len(sys.argv) == 2 and sys.argv[1] == '-help':
        print('Use: join.py [from-dir-name to-file-name]')
    else:
        if len(sys.argv) != 3:
            interactive = True
            fromdir = input('Directory containing part files? ')
            tofile = input('Name of file to be recreated? ')
        else:
            interactive = False
            fromdir, tofile = sys.argv[1:]
        absfrom, absto = map(os.path.abspath, [fromdir, tofile])
        print('Joining', absfrom, 'to make', absto)

    try:
        join(fromdir, tofile)
    except:
        print('Error joining files:')
        print(sys.exc_info()[0], sys.exc_info()[1])
    else:
        print('Join complete: see', absto)
    if interactive: input('Press Enter key') # pause if clicked

```

Here is a join in progress on Windows, combining the split files we made a moment ago; after running the `join` script, you still may need to run something like `zip`, `gzip`, or `tar` to unpack an archive file unless it's shipped as an executable, but at least the original downloaded file is set to go[‡]:

```
C:\temp> python C:\...\PP4E\System\Filetools\join.py -help
Use: join.py [from-dir-name to-file-name]
```

[‡] It turns out that the `zip`, `gzip`, and `tar` commands can all be replaced with pure Python code today, too. The `gzip` module in the Python standard library provides tools for reading and writing compressed `.gz` files, usually named with a `.gz` filename extension. It can serve as an all-Python equivalent of the standard `gzip` and `gunzip` command-line utility programs. This built-in module uses another module called `zlib` that implements `gzip`-compatible data compressions. In recent Python releases, the `zipfile` module can be imported to make and use ZIP format archives (`zip` is an archive and compression format, `gzip` is a compression scheme), and the `tarfile` module allows scripts to read and write tar archives. See the Python library manual for details.

```
C:\temp> python C:\...\PP4E\System\Filetools\join.py pysplit mypy31.msi
Joining C:\temp\pysplit to make C:\temp\mypy31.msi
Join complete: see C:\temp\mypy31.msi

C:\temp> dir *.msi
...more...
02/21/2010 11:21 AM      13,814,272 mypy31.msi
06/27/2009  04:53 PM      13,814,272 python-3.1.msi
              2 File(s)   27,628,544 bytes
              0 Dir(s)  188,798,611,456 bytes free

C:\temp> fc /b mypy31.msi python-3.1.msi
Comparing files mypy31.msi and PYTHON-3.1.MSI
FC: no differences encountered
```

The join script simply uses `os.listdir` to collect all the part files in a directory created by `split`, and sorts the filename list to put the parts back together in the correct order. We get back an exact byte-for-byte copy of the original file (proved by the DOS `fc` command in the code; use `cmp` on Unix).

Some of this process is still manual, of course (I never did figure out how to script the “walk the floppies to your bedroom” step), but the `split` and `join` scripts make it both quick and simple to move big files around. Because this script is also portable Python code, it runs on any platform to which we cared to move split files. For instance, my home computers ran both Windows and Linux at the time; since this script runs on either platform, the gamers were covered. Before we move on, here are a couple of implementation details worth underscoring in the `join` script’s code:

Reading by blocks or files

First of all, notice that this script deals with files in binary mode but also reads each part file in blocks of 1 KB each. In fact, the `readsize` setting here (the size of each block read from an input part file) has no relation to `chunksize` in `split.py` (the total size of each output part file). As we learned in [Chapter 4](#), this script could instead read each part file all at once: `output.write(open(filepath, 'rb').read())`. The downside to this scheme is that it really does load all of a file into memory at once. For example, reading a 1.4 MB part file into memory all at once with the file object `read` method generates a 1.4 MB string in memory to hold the file’s bytes. Since `split` allows users to specify even larger chunk sizes, the `join` script plans for the worst and reads in terms of limited-size blocks. To be completely robust, the `split` script could read its input data in smaller chunks too, but this hasn’t become a concern in practice (recall that as your program runs, Python automatically reclaims strings that are no longer referenced, so this isn’t as wasteful as it might seem).

Sorting filenames

If you study this script’s code closely, you may also notice that the `join` scheme it uses relies completely on the sort order of filenames in the parts directory. Because it simply calls the `list` `sort` method on the `filenames` list returned by `os.listdir`, it implicitly requires that filenames have the same length and format when created

by `split`. To satisfy this requirement, the splitter uses zero-padding notation in a string formatting expression ('`part%04d`') to make sure that filenames all have the same number of digits at the end (four). When sorted, the leading zero characters in small numbers guarantee that part files are ordered for joining correctly.

Alternatively, we could strip off digits in filenames, convert them with `int`, and sort numerically, by using the list `sort` method's `keys` argument, but that would still imply that all filenames must start with the same type of substring, and so doesn't quite remove the file-naming dependency between the `split` and `join` scripts. Because these scripts are designed to be two steps of the same process, though, some dependencies between them seem reasonable.

Usage Variations

Finally, let's run a few more experiments with these Python system utilities to demonstrate other usage modes. When run without full command-line arguments, both `split` and `join` are smart enough to input their parameters *interactively*. Here they are chopping and gluing the Python self-installer file on Windows again, with parameters typed in the DOS console window:

```
C:\temp> python C:\...\PP4E\System\Filetools\split.py
File to be split? python-3.1.msi
Directory to store part files? splitout
Splitting C:\temp\python-3.1.msi to C:\temp\splitout by 1433600
Split finished: 10 parts are in C:\temp\splitout
Press Enter key

C:\temp> python C:\...\PP4E\System\Filetools\join.py
Directory containing part files? splitout
Name of file to be recreated? newpy31.msi
Joining C:\temp\splitout to make C:\temp\newpy31.msi
Join complete: see C:\temp\newpy31.msi
Press Enter key

C:\temp> fc /B python-3.1.msi newpy31.msi
Comparing files python-3.1.msi and NEWPY31.MSI
FC: no differences encountered
```

When these program files are *double-clicked* in a Windows file explorer GUI, they work the same way (there are usually no command-line arguments when they are launched this way). In this mode, absolute path displays help clarify where files really are. Remember, the current working directory is the script's home directory when clicked like this, so a simple name actually maps to a source code directory; type a full path to make the split files show up somewhere else:

[in a pop-up DOS console box when split.py is clicked]

```
File to be split? c:\temp\python-3.1.msi
Directory to store part files? c:\temp\parts
Splitting c:\temp\python-3.1.msi to c:\temp\parts by 1433600
Split finished: 10 parts are in c:\temp\parts
Press Enter key
```

```
[in a pop-up DOS console box when join.py is clicked]
Directory containing part files? c:\temp\parts
Name of file to be recreated? c:\temp\morepy31.msi
Joining c:\temp\parts to make c:\temp\morepy31.msi
Join complete: see c:\temp\morepy31.msi
Press Enter key
```

Because these scripts package their core logic in functions, though, it's just as easy to reuse their code by *importing* and calling from another Python component (make sure your module import search path includes the directory containing the PP4E root first; the first abbreviated line here is one way to do so):

```
C:\temp> set PYTHONPATH=C:\...\dev\Examples
C:\temp> python
>>> from PP4E.System.Filetools.split import split
>>> from PP4E.System.Filetools.join import join
>>>
>>> numparts = split('python-3.1.msi', 'calldir')
>>> numparts
10
>>> join('calldir', 'callpy31.msi')
>>>
>>> import os
>>> os.system('fc /B python-3.1.msi callpy31.msi')
Comparing files python-3.1.msi and CALLPY31.msi
FC: no differences encountered
0
```

A word about performance: all the `split` and `join` tests shown so far process a 13 MB file, but they take less than one second of real wall-clock time to finish on my Windows 7 2GHz Atom processor laptop computer—plenty fast for just about any use I could imagine. Both scripts run just as fast for other reasonable *part file sizes*, too; here is the splitter chopping up the file into 4MB and 500KB parts:

```
C:\temp> C:\...\PP4E\System\Filetools\split.py python-3.1.msi tempsplit 4000000
Splitting C:\temp\python-3.1.msi to C:\temp\tempsplit by 4000000
Split finished: 4 parts are in C:\temp\tempsplit

C:\temp> dir tempsplit
...more...
Directory of C:\temp\tempsplit

02/21/2010  01:27 PM    <DIR>          .
02/21/2010  01:27 PM    <DIR>          ..
02/21/2010  01:27 PM        4,000,000 part0001
02/21/2010  01:27 PM        4,000,000 part0002
02/21/2010  01:27 PM        4,000,000 part0003
02/21/2010  01:27 PM        1,814,272 part0004
              4 File(s)   13,814,272 bytes
              2 Dir(s)  188,671,983,616 bytes free
```

```
C:\temp> C:\...\PP4E\System\Filetools\split.py python-3.1.msi tempsplit 500000
Splitting C:\temp\python-3.1.msi to C:\temp\tempsplit by 500000
Split finished: 28 parts are in C:\temp\tempsplit

C:\temp> dir tempsplit
...more...
Directory of C:\temp\tempsplit

02/21/2010  01:27 PM    <DIR>      .
02/21/2010  01:27 PM    <DIR>      ..
02/21/2010  01:27 PM            500,000 part0001
02/21/2010  01:27 PM            500,000 part0002
02/21/2010  01:27 PM            500,000 part0003
02/21/2010  01:27 PM            500,000 part0004
02/21/2010  01:27 PM            500,000 part0005
...more lines omitted...
02/21/2010  01:27 PM            500,000 part0024
02/21/2010  01:27 PM            500,000 part0025
02/21/2010  01:27 PM            500,000 part0026
02/21/2010  01:27 PM            500,000 part0027
02/21/2010  01:27 PM            314,272 part0028
      28 File(s)     13,814,272 bytes
      2 Dir(s)   188,671,946,752 bytes free
```

The split can take noticeably longer to finish, but only if the part file's size is set small enough to generate thousands of part files—splitting into 1,382 parts works but runs slower (though some machines today are quick enough that you might not notice):

```
C:\temp> C:\...\PP4E\System\Filetools\split.py python-3.1.msi tempsplit 10000
Splitting C:\temp\python-3.1.msi to C:\temp\tempsplit by 10000
Split finished: 1382 parts are in C:\temp\tempsplit

C:\temp> C:\...\PP4E\System\Filetools\join.py tempsplit manypy31.msi
Joining C:\temp\tempsplit to make C:\temp\manypy31.msi
Join complete: see C:\temp\manypy31.msi

C:\temp> fc /B python-3.1.msi manypy31.msi
Comparing files python-3.1.msi and MANYPY31.MSI
FC: no differences encountered

C:\temp> dir tempsplit
...more...
Directory of C:\temp\tempsplit

02/21/2010  01:40 PM    <DIR>      .
02/21/2010  01:40 PM    <DIR>      ..
02/21/2010  01:39 PM            10,000 part0001
02/21/2010  01:39 PM            10,000 part0002
02/21/2010  01:39 PM            10,000 part0003
02/21/2010  01:39 PM            10,000 part0004
02/21/2010  01:39 PM            10,000 part0005
```

```
...over 1,000 lines deleted...
02/21/2010 01:40 PM      10,000 part1378
02/21/2010 01:40 PM      10,000 part1379
02/21/2010 01:40 PM      10,000 part1380
02/21/2010 01:40 PM      10,000 part1381
02/21/2010 01:40 PM      4,272 part1382
1382 File(s)   13,814,272 bytes
2 Dir(s)  188,651,008,000 bytes free
```

Finally, the splitter is also smart enough to create the output directory if it doesn't yet exist and to clear out any old files there if it does exist—the following, for example, leaves only new files in the output directory. Because the joiner combines whatever files exist in the output directory, this is a nice ergonomic touch. If the output directory was not cleared before each split, it would be too easy to forget that a prior run's files are still there. Given that target audience for these scripts, they needed to be as forgiving as possible; your user base may vary (though you often shouldn't assume so).

```
C:\temp> C:\...\PP4E\System\Filetools\split.py python-3.1.msi tempsplit 5000000
Splitting C:\temp\python-3.1.msi to C:\temp\tempsplit by 5000000
Split finished: 3 parts are in C:\temp\tempsplit
```

```
C:\temp> dir tempsplit
...more...
Directory of C:\temp\tempsplit

02/21/2010 01:47 PM    <DIR>      .
02/21/2010 01:47 PM    <DIR>      ..
02/21/2010 01:47 PM      5,000,000 part0001
02/21/2010 01:47 PM      5,000,000 part0002
02/21/2010 01:47 PM      3,814,272 part0003
3 File(s)   13,814,272 bytes
2 Dir(s)  188,654,452,736 bytes free
```

Of course, the dilemma that these scripts address might today be more easily addressed by simply buying a bigger memory stick or giving kids their own Internet access. Still, once you catch the scripting bug, you'll find the ease and flexibility of Python to be powerful and enabling tools, especially for writing custom automation scripts like these. When used well, Python may well become your Swiss Army knife of computing.

Generating Redirection Web Pages

Moving is rarely painless, even in cyberspace. Changing your website's Internet address can lead to all sorts of confusion. You need to ask known contacts to use the new address and hope that others will eventually stumble onto it themselves. But if you rely on the Internet, moves are bound to generate at least as much confusion as an address change in the real world.

Unfortunately, such site relocations are often unavoidable. Both Internet Service Providers (ISPs) and server machines can come and go over the years. Moreover, some ISPs

let their service fall to intolerably low levels; if you are unlucky enough to have signed up with such an ISP, there is not much recourse but to change providers, and that often implies a change of web addresses.[§]

Imagine, though, that you are an O'Reilly author and have published your website's address in multiple books sold widely all over the world. What do you do when your ISP's service level requires a site change? Notifying each of the hundreds of thousands of readers out there isn't exactly a practical solution.

Probably the best you can do is to leave forwarding instructions at the old site for some reasonably long period of time—the virtual equivalent of a "We've Moved" sign in a storefront window. On the Web, such a sign can also send visitors to the new site automatically: simply leave a page at the old site containing a hyperlink to the page's address at the new site, along with timed auto-relocation specifications. With such *forward-link files* in place, visitors to the old addresses will be only one click or a few seconds away from reaching the new ones.

That sounds simple enough. But because visitors might try to directly access the address of *any* file at your old site, you generally need to leave one forward-link file for every old file—HTML pages, images, and so on. Unless your prior server supports auto-redirection (and mine did not), this represents a dilemma. If you happen to enjoy doing lots of mindless typing, you could create each forward-link file by hand. But given that my home site contained over 100 HTML files at the time I wrote this paragraph, the prospect of running one editor session per file was more than enough motivation for an automated solution.

Page Template File

Here's what I came up with. First of all, I create a general *page template* text file, shown in [Example 6-7](#), to describe how all the forward-link files should look, with parts to be filled in later.

Example 6-7. PP4E\System\Filetools\template.html

```
<HTML>
<head>
<META HTTP-EQUIV="Refresh" CONTENT="10; URL=http://$server$/home$/file$">
<title>Site Redirection Page: $file$</title>
</head>
<BODY>

<H1>This page has moved</H1>
<P>This page now lives at this address:
```

[§] It happens. In fact, most people who spend any substantial amount of time in cyberspace could probably tell a horror story or two. Mine goes like this: a number of years ago, I had an account with an ISP that went completely offline for a few weeks in response to a security breach by an ex-employee. Worse, not only was personal email disabled, but queued up messages were permanently lost. If your livelihood depends on email and the Web as much as mine does, you'll appreciate the havoc such an outage can wreak.

```

<P><A HREF="http://\$server\$/home\$/file\$>
http://$server$/home$/file$</A>

<P>Please click on the new address to jump to this page, and
update any links accordingly. You will be redirected shortly.
</P>

<HR>
</BODY></HTML>

```

To fully understand this template, you have to know something about HTML, a web page description language that we'll explore in [Part IV](#). But for the purposes of this example, you can ignore most of this file and focus on just the parts surrounded by dollar signs: the strings `$server$`, `$home$`, and `$file$` are targets to be replaced with real values by global text substitutions. They represent items that vary per site relocation and file.

Page Generator Script

Now, given a page template file, the Python script in [Example 6-8](#) generates all the required forward-link files automatically.

Example 6-8. PP4E\System\Filetools\site-forward.py

```

"""
#####
# Create forward-link pages for relocating a web site.
# Generates one page for every existing site html file; upload the generated
# files to your old web site. See ftplib later in the book for ways to run
# uploads in scripts either after or during page file creation.
#####

import os
servername = 'learning-python.com'      # where site is relocating to
homedir = 'books'                      # where site will be rooted
sitefilesdir = r'C:\temp\public_html'   # where site files live locally
uploaddir = r'C:\temp\isp-forward'     # where to store forward files
templatename = 'template.html'          # template for generated pages

try:
    os.mkdir(uploaddir)                # make upload dir if needed
except OSError: pass

template = open(templatename).read()      # load or import template text
sitefiles = os.listdir(sitefilesdir)      # filenames, no directory prefix

count = 0
for filename in sitefiles:
    if filename.endswith('.html') or filename.endswith('.htm'):
        fwdname = os.path.join(uploaddir, filename)
        print('creating', filename, 'as', fwdname)

```

```

filetext = template.replace('$server$', servername)    # insert text
filetext = filetext.replace('$home$', homedir)        # and write
filetext = filetext.replace('$file$', filename)        # file varies
open(fwdname, 'w').write(filetext)
count += 1

print('Last file =>\n', filetext, sep='')
print('Done:', count, 'forward files created.')

```

Notice that the template’s text is loaded by reading a *file*; it would work just as well to code it as an imported Python string variable (e.g., a triple-quoted string in a module file). Also observe that all configuration options are assignments at the top of the *script*, not command-line arguments; since they change so seldom, it’s convenient to type them just once in the script itself.

But the main thing worth noticing here is that this script doesn’t care what the template file looks like at all; it simply performs global substitutions blindly in its text, with a different filename value for each generated file. In fact, we can change the template file any way we like without having to touch the script. Though a fairly simple technique, such a division of labor can be used in all sorts of contexts—generating “makefiles,” form letters, HTML replies from CGI scripts on web servers, and so on. In terms of library tools, the generator script:

- Uses `os.listdir` to step through all the filenames in the site’s directory (`glob.glob` would work too, but may require stripping directory prefixes from file names)
- Uses the string object’s `replace` method to perform global search-and-replace operations that fill in the \$-delimited targets in the template file’s text, and `endswith` to skip non-HTML files (e.g., images—most browsers won’t know what to do with HTML text in a “.jpg” file)
- Uses `os.path.join` and built-in file objects to write the resulting text out to a forward-link file of the same name in an output directory

The end result is a mirror image of the original website directory, containing only forward-link files generated from the page template. As an added bonus, the generator script can be run on just about any Python platform—I can run it on my Windows laptop (where I’m writing this book), as well as on a Linux server (where my <http://learning-python.com> domain is hosted). Here it is in action on Windows:

```

C:\...\PP4E\System\Filetools> python site-forward.py
creating about-lp.html as C:\temp\isp-forward\about-lp.html
creating about-lp1e.html as C:\temp\isp-forward\about-lp1e.html
creating about-lp2e.html as C:\temp\isp-forward\about-lp2e.html
creating about-lp3e.html as C:\temp\isp-forward\about-lp3e.html
creating about-lp4e.html as C:\temp\isp-forward\about-lp4e.html
...many more lines deleted...
creating training.html as C:\temp\isp-forward\training.html
creating whatsnew.html as C:\temp\isp-forward\whatsnew.html

```

```

creating whatsold.html as C:\temp\isp-forward\whatsold.html
creating xlate-lp.html as C:\temp\isp-forward\xlate-lp.html
creating zopeoutline.htm as C:\temp\isp-forward\zopeoutline.htm
Last file =>
<HTML>
<head>
<META HTTP-EQUIV="Refresh" CONTENT="10; URL=http://learning-python.com/books/zop
eoutline.htm">
<title>Site Redirection Page: zopeoutline.htm</title>
</head>
<BODY>

<H1>This page has moved</H1>
<P>This page now lives at this address:

<P><A HREF="http://learning-python.com/books/zopeoutline.htm">
http://learning-python.com/books/zopeoutline.htm</A>

<P>Please click on the new address to jump to this page, and
update any links accordingly. You will be redirected shortly.
</P>

<HR>
</BODY></HTML>
Done: 124 forward files created.

```

To verify this script's output, double-click on any of the output files to see what they look like in a web browser (or run a `start` command in a DOS console on Windows—e.g., `start isp-forward\about-lp4e.html`). Figure 6-1 shows what one generated page looks like on my machine.

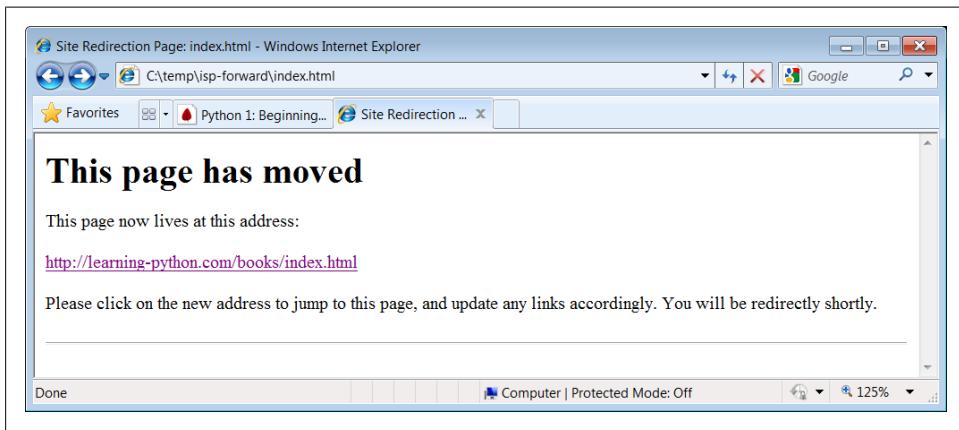


Figure 6-1. Site-forward output file page

To complete the process, you still need to install the forward links: upload all the generated files in the output directory to your old site's web directory. If that's too much to do by hand, too, be sure to see the FTP site upload scripts in [Chapter 13](#) for

an automatic way to do that step with Python as well (*PP4E\Internet\Ftp\upload-flat.py* will do the job). Once you've started scripting in earnest, you'll be amazed at how much manual labor Python can automate. The next section provides another prime example.

A Regression Test Script

Mistakes happen. As we've seen, Python provides interfaces to a variety of system services, along with tools for adding others. [Example 6-9](#) shows some of the more commonly used system tools in action. It implements a simple *regression test* system for Python scripts—it runs each in a directory of Python scripts with provided input and command-line arguments, and compares the output of each run to the prior run's results. As such, this script can be used as an automated testing system to catch errors introduced by changes in program source files; in a big system, you might not know when a fix is really a bug in disguise.

Example 6-9. PP4E\System\Tester\tester.py

```
"""
#####
# Test a directory of Python scripts, passing command-line arguments,
# piping in stdin, and capturing stdout, stderr, and exit status to
# detect failures and regressions from prior run outputs. The subprocess
# module spawns and controls streams (much like os.popen3 in Python 2.X),
# and is cross-platform. Streams are always binary bytes in subprocess.
# Test inputs, args, outputs, and errors map to files in subdirectories.

This is a command-line script, using command-line arguments for
optional test directory name, and force-generation flag. While we
could package it as a callable function, the fact that its results
are messages and output files makes a call/return model less useful.

Suggested enhancement: could be extended to allow multiple sets
of command-line arguments and/or inputs per test script, to run a
script multiple times (glob for multiple ".in*" files in Inputs?).
Might also seem simpler to store all test files in same directory
with different extensions, but this could grow large over time.
Could also save both stderr and stdout to Errors on failures, but
I prefer to have expected/actual output in Outputs on regressions.
#####
"""

import os, sys, glob, time
from subprocess import Popen, PIPE

# configuration args
testdir = sys.argv[1] if len(sys.argv) > 1 else os.curdir
forcegen = len(sys.argv) > 2
print('Start tester:', time.asctime())
print('in', os.path.abspath(testdir))
```

```

def verbose(*args):
    print('-'*80)
    for arg in args: print(arg)
def quiet(*args): pass
trace = quiet

# glob scripts to be tested
testpatt = os.path.join(testdir, 'Scripts', '*.py')
testfiles = glob.glob(testpatt)
testfiles.sort()
trace(os.getcwd(), *testfiles)

numfail = 0
for testpath in testfiles:                      # run all tests in dir
    testname = os.path.basename(testpath)         # strip directory path

    # get input and args
    infile = testname.replace('.py', '.in')
    inpath = os.path.join(testdir, 'Inputs', infile)
    indata = open(inpath, 'rb').read() if os.path.exists(inpath) else b''

    argfile = testname.replace('.py', '.args')
    argpath = os.path.join(testdir, 'Args', argfile)
    argdata = open(argpath).read() if os.path.exists(argpath) else ''

    # locate output and error, scrub prior results
    outfile = testname.replace('.py', '.out')
    outpath = os.path.join(testdir, 'Outputs', outfile)
    outpathbad = outpath + '.bad'
    if os.path.exists(outpathbad): os.remove(outpathbad)

    errfile = testname.replace('.py', '.err')
    errpath = os.path.join(testdir, 'Errors', errfile)
    if os.path.exists(errpath): os.remove(errpath)

    # run test with redirected streams
    pypath = sys.executable
    command = '%s %s %s' % (pypath, testpath, argdata)
    trace(command, indata)

    process = Popen(command, shell=True, stdin=PIPE, stdout=PIPE, stderr=PIPE)
    process.stdin.write(indata)
    process.stdin.close()
    outdata = process.stdout.read()                         # data are bytes
    errdata = process.stderr.read()                        # requires binary files
    exitstatus = process.wait()
    trace(outdata, errdata, exitstatus)

    # analyze results
    if exitstatus != 0:
        print('ERROR status:', testname, exitstatus)      # status and/or stderr
    if errdata:
        print('ERROR stream:', testname, errpath)        # save error text
        open(errpath, 'wb').write(errdata)

```

```

if exitstatus or errdata:                                # consider both failure
    numfail += 1
    open(outpathbad, 'wb').write(outdata)
    # can get status+stderr
    # save output to view

elif not os.path.exists(outpath) or forcegen:
    print('generating:', outpath)                      # create first output
    open(outpath, 'wb').write(outdata)

else:
    priorout = open(outpath, 'rb').read()              # or compare to prior
    if priorout == outdata:
        print('passed:', testname)
    else:
        numfail += 1
        print('FAILED output:', testname, outpathbad)
        open(outpathbad, 'wb').write(outdata)

print('Finished:', time.asctime())
print('%s tests were run, %s tests failed.' % (len(testfiles), numfail))

```

We've seen the tools used by this script earlier in this part of the book—`subprocess`, `os.path`, `glob`, `files`, and the like. This example largely just pulls these tools together to solve a useful purpose. Its core operation is comparing new outputs to old, in order to spot changes (“regressions”). Along the way, it also manages command-line arguments, error messages, status codes, and files.

This script is also larger than most we've seen so far, but it's a realistic and representative system administration tool (in fact, it's derived from a similar tool I actually used in the past to detect changes in a compiler). Probably the best way to understand how it works is to demonstrate what it does. The next section steps through a testing session to be read in conjunction with studying the test script's code.

Running the Test Driver

Much of the magic behind the test driver script in [Example 6-9](#) has to do with its directory structure. When you run it for the first time in a test directory (or force it to start from scratch there by passing a second command-line argument), it:

- Collects scripts to be run in the `Scripts` subdirectory
- Fetches any associated script input and command-line arguments from the `Inputs` and `Args` subdirectories
- Generates initial `stdout` output files for tests that exit normally in the `Outputs` subdirectory
- Reports tests that fail either by exit status code or by error messages appearing in `stderr`

On all failures, the script also saves any `stderr` error message text, as well as any `stdout` data generated up to the point of failure; standard error text is saved to a file in the `Errors` subdirectory, and standard output of failed tests is saved with a special

“.bad” filename extension in `Outputs` (saving this normally in the `Outputs` subdirectory would trigger a failure when the test is later fixed!). Here’s a first run:

```
C:\...\PP4E\System\Tester> python tester.py . 1
Start tester: Mon Feb 22 22:13:38 2010
in C:\Users\mark\Stuff\Books\4E\PP4E\dev\Examples\PP4E\System\Tester
generating: .\Outputs\test-basic-args.out
generating: .\Outputs\test-basic-stdout.out
generating: .\Outputs\test-basic-streams.out
generating: .\Outputs\test-basic-this.out
ERROR status: test-errors-runtime.py 1
ERROR stream: test-errors-runtime.py .\Errors\test-errors-runtime.err
ERROR status: test-errors-syntax.py 1
ERROR stream: test-errors-syntax.py .\Errors\test-errors-syntax.err
ERROR status: test-status-bad.py 42
generating: .\Outputs\test-status-good.out
Finished: Mon Feb 22 22:13:41 2010
8 tests were run, 3 tests failed.
```

To run each script, the tester configures any preset command-line arguments provided, pipes in fetched canned input (if any), and captures the script’s standard output and error streams, along with its exit status code. When I ran this example, there were 8 test scripts, along with a variety of inputs and outputs. Since the directory and file naming structures are the key to this example, here is a listing of the test directory I used—the `Scripts` directory is primary, because that’s where tests to be run are collected:

```
C:\...\PP4E\System\Tester> dir /B
Args
Errors
Inputs
Outputs
Scripts
tester.py
xxold

C:\...\PP4E\System\Tester> dir /B Scripts
test-basic-args.py
test-basic-stdout.py
test-basic-streams.py
test-basic-this.py
test-errors-runtime.py
test-errors-syntax.py
test-status-bad.py
test-status-good.py
```

The other subdirectories contain any required inputs and any generated outputs associated with scripts to be tested:

```
C:\...\PP4E\System\Tester> dir /B Args
test-basic-args.args
test-status-good.args
```

```
C:\...\PP4E\System\Tester> dir /B Inputs
test-basic-args.in
test-basic-streams.in
```

```
C:\...\PP4E\System\Tester> dir /B Outputs
test-basic-args.out
test-basic-stdout.out
test-basic-streams.out
test-basic-this.out
test-errors-runtime.out.bad
test-errors-syntax.out.bad
test-status-bad.out.bad
test-status-good.out
```

```
C:\...\PP4E\System\Tester> dir /B Errors
test-errors-runtime.err
test-errors-syntax.err
```

I won't list all these files here (as you can see, there are many, and all are available in the book examples distribution package), but to give you the general flavor, here are the files associated with the test script *test-basic-args.py*:

```
C:\...\PP4E\System\Tester> type Scripts\test-basic-args.py
# test args, streams
import sys, os
print(os.getcwd())                      # to Outputs
print(sys.path[0])

print('[argv]')
for arg in sys.argv:                   # from Args
    print(arg)                         # to Outputs

print('[interaction]')                  # to Outputs
text = input('Enter text:')            # from Inputs
rept = sys.stdin.readline()           # from Inputs
sys.stdout.write(text * int(rept))    # to Outputs
```

```
C:\...\PP4E\System\Tester> type Args\test-basic-args.args
-command -line --stuff
```

```
C:\...\PP4E\System\Tester> type Inputs\test-basic-args.in
Eggs
10
```

```
C:\...\PP4E\System\Tester> type Outputs\test-basic-args.out
C:\Users\mark\Stuff\Books\4E\PP4E\dev\Examples\PP4E\System\Tester
C:\Users\mark\Stuff\Books\4E\PP4E\dev\Examples\PP4E\System\Tester\Scripts
[argv]
.\Scripts\test-basic-args.py
-command
-line
--stuff
[interaction]
Enter text:EggsEggsEggsEggsEggsEggsEggsEggsEggsEggsEggs
```

And here are two files related to one of the detected errors—the first is its captured *stderr*, and the second is its *stdout* generated up to the point where the error occurred; these are for human (or other tools) inspection, and are automatically removed the next time the tester script runs:

```
C:\...\PP4E\System\Tester> type Errors\test-errors-runtime.err
Traceback (most recent call last):
  File ".\Scripts\test-errors-runtime.py", line 3, in <module>
    print(1 / 0)
ZeroDivisionError: int division or modulo by zero

C:\...\PP4E\System\Tester> type Outputs\test-errors-runtime.out.bad
starting
```

Now, when run again without making any changes to the tests, the test driver script compares saved prior outputs to new ones and detects no regressions; failures designated by exit status and *stderr* messages are still reported as before, but there are no deviations from other tests' saved expected output:

```
C:\...\PP4E\System\Tester> python tester.py
Start tester: Mon Feb 22 22:26:41 2010
in C:\Users\mark\Stuff\Books\4E\PP4E\dev\Examples\PP4E\System\Tester
passed: test-basic-args.py
passed: test-basic-stdout.py
passed: test-basic-streams.py
passed: test-basic-this.py
ERROR status: test-errors-runtime.py 1
ERROR stream: test-errors-runtime.py .\Errors\test-errors-runtime.err
ERROR status: test-errors-syntax.py 1
ERROR stream: test-errors-syntax.py .\Errors\test-errors-syntax.err
ERROR status: test-status-bad.py 42
passed: test-status-good.py
Finished: Mon Feb 22 22:26:43 2010
8 tests were run, 3 tests failed.
```

But when I make a change in one of the test scripts that will produce different output (I changed a loop counter to print fewer lines), the regression is caught and reported; the new and different output of the script is reported as a failure, and saved in *Outputs* as a “.bad” for later viewing:

```
C:\...\PP4E\System\Tester> python tester.py
Start tester: Mon Feb 22 22:28:35 2010
in C:\Users\mark\Stuff\Books\4E\PP4E\dev\Examples\PP4E\System\Tester
passed: test-basic-args.py
FAILED output: test-basic-stdout.py .\Outputs\test-basic-stdout.out.bad
passed: test-basic-streams.py
passed: test-basic-this.py
ERROR status: test-errors-runtime.py 1
ERROR stream: test-errors-runtime.py .\Errors\test-errors-runtime.err
ERROR status: test-errors-syntax.py 1
ERROR stream: test-errors-syntax.py .\Errors\test-errors-syntax.err
ERROR status: test-status-bad.py 42
passed: test-status-good.py
Finished: Mon Feb 22 22:28:38 2010
```

```
8 tests were run, 4 tests failed.
```

```
C:\...\PP4E\System\Tester> type Outputs\test-basic-stdout.out.bad
begin
Spam!
Spam!Spam!
Spam!Spam!Spam!
Spam!Spam!Spam!Spam!
end
```

One last usage note: if you change the `trace` variable in this script to be `verbose`, you'll get much more output designed to help you trace the programs operation (but probably too much for real testing runs):

```
C:\...\PP4E\System\Tester> tester.py
Start tester: Mon Feb 22 22:34:51 2010
in C:\Users\mark\Stuff\Books\4E\PP4E\dev\Examples\PP4E\System\Tester
-----
C:\Users\mark\Stuff\Books\4E\PP4E\dev\Examples\PP4E\System\Tester
.\Scripts\test-basic-args.py
.\Scripts\test-basic-stdout.py
.\Scripts\test-basic-streams.py
.\Scripts\test-basic-this.py
.\Scripts\test-errors-runtime.py
.\Scripts\test-errors-syntax.py
.\Scripts\test-status-bad.py
.\Scripts\test-status-good.py
-----
C:\Python31\python.exe .\Scripts\test-basic-args.py -command -line --stuff
b'Eggs\r\n10\r\n'
-----
b'C:\\\\Users\\\\mark\\\\Stuff\\\\Books\\\\4E\\\\PP4E\\\\dev\\\\Examples\\\\PP4E\\\\System\\\\Tester\\\\
\\nC:\\\\Users\\\\mark\\\\Stuff\\\\Books\\\\4E\\\\PP4E\\\\dev\\\\Examples\\\\PP4E\\\\System\\\\Tester\\\\
Scripts\\\\n[argv]\\r\\n.\\\\Scripts\\\\test-basic-args.py\\r\\n-command\\r\\n-line\\r\\n--st
uff\\r\\n[interaction]\\r\\nEnter text:EggsEggsEggsEggsEggsEggsEggsEggsEggsEggs'
b''
0
passed: test-basic-args.py
...more lines deleted...
```

Study the test driver's code for more details. Naturally, there is much more to the general testing story than we have space for here. For example, in-process tests don't need to spawn programs and can generally make do with importing modules and testing them in `try` exception handler statements. There is also ample room for expansion and customization in our testing script (see its docstring for starters). Moreover, Python comes with two testing frameworks, `doctest` and `unittest` (a.k.a. PyUnit), which provide techniques and structures for coding regression and unit tests:

`unittest`

An object-oriented framework that specifies test cases, expected results, and test suites. Subclasses provide test methods and use inherited assertion calls to specify expected results.

doctest

Parses out and reruns tests from an interactive session log that is pasted into a module’s docstrings. The logs give test calls and expected results; doctest essentially reruns the interactive session.

See the Python library manual, the PyPI website, and your favorite Web search engine for additional testing toolkits in both Python itself and the third-party domain.

For automated testing of Python command-line scripts that run as independent programs and tap into standard script execution context, though, our tester does the job. Because the test driver is fully independent of the scripts it tests, we can drop in new test cases without having to update the driver’s code. And because it is written in Python, it’s quick and easy to change as our testing needs evolve. As we’ll see again in the next section, this “scriptability” that Python provides can be a decided advantage for real tasks.

Testing Gone Bad?

Once we learn about sending email from Python scripts in [Chapter 13](#), you might also want to augment this script to automatically send out email when regularly run tests fail (e.g., when run from a `cron` job on Unix). That way, you don’t even need to remember to check results. Of course, you could go further still.

One company I worked for added sound effects to compiler test scripts; you got an audible round of applause if no regressions were found and an entirely different noise otherwise. (See `playfile.py` at the end of this chapter for hints.)

Another company in my development past ran a nightly test script that automatically isolated the source code file check-in that triggered a test regression and sent a nasty email to the guilty party (and his or her supervisor). Nobody expects the Spanish Inquisition!

Copying Directory Trees

My CD writer sometimes does weird things. In fact, copies of files with odd names can be totally botched on the CD, even though other files show up in one piece. That’s not necessarily a showstopper; if just a few files are trashed in a big CD backup copy, I can always copy the offending files elsewhere one at a time. Unfortunately, drag-and-drop copies on some versions of Windows don’t play nicely with such a CD: the copy operation stops and exits the moment the first bad file is encountered. You get only as many files as were copied up to the error, but no more.

In fact, this is not limited to CD copies. I’ve run into similar problems when trying to back up my laptop’s hard drive to another drive—the drag-and-drop copy stops with an error as soon as it reaches a file with a name that is too long or odd to copy (common

in saved web pages). The last 30 minutes spent copying is wasted time; frustrating, to say the least!

There may be some magical Windows setting to work around this feature, but I gave up hunting for one as soon as I realized that it would be easier to code a copier in Python. The `cpall.py` script in [Example 6-10](#) is one way to do it. With this script, I control what happens when bad files are found—I can skip over them with Python exception handlers, for instance. Moreover, this tool works with the same interface and effect on other platforms. It seems to me, at least, that a few minutes spent writing a portable and reusable Python script to meet a need is a better investment than looking for solutions that work on only one platform (if at all).

Example 6-10. PP4E\System\Filetools\cpall.py

```
"""
#####
Usage: "python cpall.py dirFrom dirTo".
Recursive copy of a directory tree. Works like a "cp -r dirFrom/* dirTo"
Unix command, and assumes that dirFrom and dirTo are both directories.
Was written to get around fatal error messages under Windows drag-and-drop
copies (the first bad file ends the entire copy operation immediately),
but also allows for coding more customized copy operations in Python.
#####
"""

import os, sys
maxfileload = 1000000
blksize = 1024 * 500

def copyfile(pathFrom, pathTo, maxfileload=maxfileload):
    """
    Copy one file pathFrom to pathTo, byte for byte;
    uses binary file modes to supress Unicode decode and endline transform
    """
    if os.path.getsize(pathFrom) <= maxfileload:
        bytesFrom = open(pathFrom, 'rb').read()    # read small file all at once
        open(pathTo, 'wb').write(bytesFrom)
    else:
        fileFrom = open(pathFrom, 'rb')            # read big files in chunks
        fileTo   = open(pathTo,   'wb')            # need b mode for both
        while True:
            bytesFrom = fileFrom.read(blksize)    # get one block, less at end
            if not bytesFrom: break               # empty after last chunk
            fileTo.write(bytesFrom)

def copytree(dirFrom, dirTo, verbose=0):
    """
    Copy contents of dirFrom and below to dirTo, return (files, dirs) counts;
    may need to use bytes for dirnames if undecodable on other platforms;
    may need to do more file type checking on Unix: skip links, fifos, etc.
    """
    fcount = dcount = 0
    for filename in os.listdir(dirFrom):           # for files/dirs here
```

```

pathFrom = os.path.join(dirFrom, filename)
pathTo   = os.path.join(dirTo,   filename)      # extend both paths
if not os.path.isdir(pathFrom):                 # copy simple files
    try:
        if verbose > 1: print('copying', pathFrom, 'to', pathTo)
        copyfile(pathFrom, pathTo)
        fcount += 1
    except:
        print('Error copying', pathFrom, 'to', pathTo, '--skipped')
        print(sys.exc_info()[0], sys.exc_info()[1])
else:
    if verbose: print('copying dir', pathFrom, 'to', pathTo)
    try:
        os.mkdir(pathTo)                         # make new subdir
        below = copytree(pathFrom, pathTo)        # recur into subdirs
        fcount += below[0]                        # add subdir counts
        dcount += below[1]
        dcount += 1
    except:
        print('Error creating', pathTo, '--skipped')
        print(sys.exc_info()[0], sys.exc_info()[1])
return (fcount, dcount)

def getargs():
"""
Get and verify directory name arguments, returns default None on errors
"""
try:
    dirFrom, dirTo = sys.argv[1:]
except:
    print('Usage error: cpall.py dirFrom dirTo')
else:
    if not os.path.isdir(dirFrom):
        print('Error: dirFrom is not a directory')
    elif not os.path.exists(dirTo):
        os.mkdir(dirTo)
        print('Note: dirTo was created')
        return (dirFrom, dirTo)
    else:
        print('Warning: dirTo already exists')
        if hasattr(os.path, 'samefile'):
            same = os.path.samefile(dirFrom, dirTo)
        else:
            same = os.path.abspath(dirFrom) == os.path.abspath(dirTo)
        if same:
            print('Error: dirFrom same as dirTo')
        else:
            return (dirFrom, dirTo)

if __name__ == '__main__':
    import time
    dirstuple = getargs()
    if dirstuple:
        print('Copying...')
        start = time.clock()

```

```
fcount, dcount = copytree(*dirstuple)
print('Copied', fcount, 'files,', dcount, 'directories', end=' ')
print('in', time.clock() - start, 'seconds')
```

This script implements its own recursive tree traversal logic and keeps track of both the “from” and “to” directory paths as it goes. At every level, it copies over simple files, creates directories in the “to” path, and recurs into subdirectories with “from” and “to” paths extended by one level. There are other ways to code this task (e.g., we might change the working directory along the way with `os.chdir` calls or there is probably an `os.walk` solution which replaces from and to path prefixes as it walks), but extending paths on recursive descent works well in this script.

Notice this script’s reusable `copyfile` function—just in case there are multigigabyte files in the tree to be copied, it uses a file’s size to decide whether it should be read all at once or in chunks (remember, the file `read` method without arguments actually loads the entire file into an in-memory string). We choose fairly large file and block sizes, because the more we read at once in Python, the faster our scripts will typically run. This is more efficient than it may sound; strings left behind by prior reads will be garbage collected and reused as we go. We’re using binary file modes here again, too, to suppress the Unicode encodings and end-of-line translations of text files—trees may contain arbitrary kinds of files.

Also notice that this script creates the “to” directory if needed, but it assumes that the directory is empty when a copy starts up; for accuracy, be sure to remove the target directory before copying a new tree to its name, or old files may linger in the target tree (we could automatically remove the target first, but this may not always be desired). This script also tries to determine if the source and target are the same; on Unix-like platforms with oddities such as links, `os.path.samefile` does a more accurate job than comparing absolute file names (different file names may be the same file).

Here is a copy of a big book examples tree (I use the tree from the prior edition throughout this chapter) in action on Windows; pass in the name of the “from” and “to” directories to kick off the process, redirect the output to a file if there are too many error messages to read all at once (e.g., `> output.txt`), and run an `rm -r` or `rmdir /S` shell command (or similar platform-specific tool) to delete the target directory first if needed:

```
C:\...\PP4E\System\Filetools> rmdir /S copytemp
copytemp, Are you sure (Y/N)? y

C:\...\PP4E\System\Filetools> cpall.py C:\temp\PP3E\Examples copytemp
Note: dirTo was created
Copying...
Copied 1430 files, 185 directories in 10.4470980971 seconds

C:\...\PP4E\System\Filetools> fc /B copytemp\PP3E\Launcher.py
                                C:\temp\PP3E\Examples\PP3E\Launcher.py
Comparing files COPYTEMP\PP3E\Launcher.py and C:\TEMP\PP3E\EXAMPLES\PP3E\LAUNCHER.PY
FC: no differences encountered
```

You can use the copy function’s `verbose` argument to trace the process if you wish. At the time I wrote this edition in 2010, this test run copied a tree of 1,430 files and 185 directories in 10 seconds on my woefully underpowered netbook machine (the built-in `time.clock` call is used to query the system time in seconds); it may run arbitrarily faster or slower for you. Still, this is at least as fast as the best drag-and-drop I’ve timed on this machine.

So how does this script work around bad files on a CD backup? The secret is that it catches and ignores file *exceptions*, and it keeps walking. To copy all the files that are good on a CD, I simply run a command line such as this one:

```
C:\...\PP4E\System\Filetools> python cpall.py G:\Examples C:\PP3E\Examples
```

Because the CD is addressed as “G:” on my Windows machine, this is the command-line equivalent of drag-and-drop copying from an item in the CD’s top-level folder, except that the Python script will recover from errors on the CD and get the rest. On copy errors, it prints a message to standard output and continues; for big copies, you’ll probably want to redirect the script’s output to a file for later inspection.

In general, `cpall` can be passed any absolute directory path on your machine, even those that indicate devices such as CDs. To make this go on Linux, try a root directory such as `/dev/cdrom` or something similar to address your CD drive. Once you’ve copied a tree this way, you still might want to verify; to see how, let’s move on to the next example.

Comparing Directory Trees

Engineers can be a paranoid sort (but you didn’t hear that from me). At least I am. It comes from decades of seeing things go terribly wrong, I suppose. When I create a CD backup of my hard drive, for instance, there’s still something a bit too magical about the process to trust the CD writer program to do the right thing. Maybe I should, but it’s tough to have a lot of faith in tools that occasionally trash files and seem to crash my Windows machine every third Tuesday of the month. When push comes to shove, it’s nice to be able to verify that data copied to a backup CD is the same as the original—or at least to spot deviations from the original—as soon as possible. If a backup is ever needed, it will be *really* needed.

Because data CDs are accessible as simple directory trees in the file system, we are once again in the realm of tree walkers—to verify a backup CD, we simply need to walk its top-level directory. If our script is general enough, we will also be able to use it to verify other copy operations as well—e.g., downloaded tar files, hard-drive backups, and so on. In fact, the combination of the `cpall` script of the prior section and a general tree comparison would provide a portable and scriptable way to copy and verify data sets.

We’ve already studied generic directory tree walkers, but they won’t help us here directly: we need to walk *two* directories in parallel and inspect common files along the way. Moreover, walking either one of the two directories won’t allow us to spot files

and directories that exist only in the other. Something more custom and recursive seems in order here.

Finding Directory Differences

Before we start coding, the first thing we need to clarify is what it means to compare two directory trees. If both trees have exactly the same branch structure and depth, this problem reduces to comparing corresponding files in each tree. In general, though, the trees can have arbitrarily different shapes, depths, and so on.

More generally, the contents of a directory in one tree may have more or fewer entries than the corresponding directory in the other tree. If those differing contents are filenames, there is no corresponding file to compare with; if they are directory names, there is no corresponding branch to descend through. In fact, the only way to detect files and directories that appear in one tree but not the other is to detect differences in each level's directory.

In other words, a tree comparison algorithm will also have to perform *directory* comparisons along the way. Because this is a nested and simpler operation, let's start by coding and debugging a single-directory comparison of filenames in [Example 6-11](#).

Example 6-11. PP4E\System\Filetools\dirdiff.py

```
"""
#####
Usage: python dirdiff.py dir1-path dir2-path
Compare two directories to find files that exist in one but not the other.
This version uses the os.listdir function and list difference. Note that
this script checks only filenames, not file contents--see diffall.py for an
extension that does the latter by comparing .read() results.
#####
"""

import os, sys

def reportdiffs(unique1, unique2, dir1, dir2):
    """
    Generate diffs report for one dir: part of comparedirs output
    """
    if not (unique1 or unique2):
        print('Directory lists are identical')
    else:
        if unique1:
            print('Files unique to', dir1)
            for file in unique1:
                print('...', file)
        if unique2:
            print('Files unique to', dir2)
            for file in unique2:
                print('...', file)

def difference(seq1, seq2):
```

```

"""
Return all items in seq1 only;
a set(seq1) - set(seq2) would work too, but sets are randomly
ordered, so any platform-dependent directory order would be lost
"""
return [item for item in seq1 if item not in seq2]

def comparedirs(dir1, dir2, files1=None, files2=None):
    """
    Compare directory contents, but not actual files;
    may need bytes listdir arg for undecodable filenames on some platforms
    """
    print('Comparing', dir1, 'to', dir2)
    files1 = os.listdir(dir1) if files1 is None else files1
    files2 = os.listdir(dir2) if files2 is None else files2
    unique1 = difference(files1, files2)
    unique2 = difference(files2, files1)
    reportdiffs(unique1, unique2, dir1, dir2)
    return not (unique1 or unique2)           # true if no diffs

def getargs():
    "Args for command-line mode"
    try:
        dir1, dir2 = sys.argv[1:]           # 2 command-line args
    except:
        print('Usage: dirdiff.py dir1 dir2')
        sys.exit(1)
    else:
        return (dir1, dir2)

if __name__ == '__main__':
    dir1, dir2 = getargs()
    comparedirs(dir1, dir2)

```

Given listings of names in two directories, this script simply picks out unique names in the first and unique names in the second, and reports any unique names found as differences (that is, files in one directory but not the other). Its `comparedirs` function returns a true result if no differences were found, which is useful for detecting differences in callers.

Let's run this script on a few directories; differences are detected and reported as names unique in either passed-in directory pathname. Notice that this is only a *structural* comparison that just checks names in listings, not file contents (we'll add the latter in a moment):

```

C:\...\PP4E\System\Filetools> dirdiff.py C:\temp\PP3E\Examples copytemp
Comparing C:\temp\PP3E\Examples to copytemp
Directory lists are identical

C:\...\PP4E\System\Filetools> dirdiff.py C:\temp\PP3E\Examples\PP3E\System ..
Comparing C:\temp\PP3E\Examples\PP3E\System to ..
Files unique to C:\temp\PP3E\Examples\PP3E\System
... App

```

```
... Exits
... Media
... moreplus.py
Files unique to ..
... more.pyc
... spam.txt
... Tester
... __init__.pyc
```

The `unique` function is the heart of this script: it performs a simple list difference operation. When applied to directories, `unique` items represent tree differences, and `common` items are names of files or subdirectories that merit further comparisons or traversals. In fact, in Python 2.4 and later, we could also use the built-in `set` object type if we don't care about the order in the results—because sets are not sequences, they would not maintain any original and possibly platform-specific left-to-right order of the directory listings provided by `os.listdir`. For that reason (and to avoid requiring users to upgrade), we'll keep using our own comprehension-based function instead of sets.

Finding Tree Differences

We've just coded a directory comparison tool that picks out unique files and directories. Now all we need is a tree walker that applies `dirdiff` at each level to report unique items, explicitly compares the contents of files in common, and descends through directories in common. [Example 6-12](#) fits the bill.

Example 6-12. PP4E\System\Filetools\diffall.py

```
"""
#####
Usage: "python diffall.py dir1 dir2".
Recursive directory tree comparison: report unique files that exist in only
dir1 or dir2, report files of the same name in dir1 and dir2 with differing
contents, report instances of same name but different type in dir1 and dir2,
and do the same for all subdirectories of the same names in and below dir1
and dir2. A summary of diffs appears at end of output, but search redirected
output for "DIFF" and "unique" strings for further details. New: (3E) limit
reads to 1M for large files, (3E) catch same name=file/dir, (4E) avoid extra
os.listdir() calls in dirdiff.comparedirs() by passing results here along.
#####
"""

import os, dirdiff
blocksize = 1024 * 1024 # up to 1M per read

def intersect(seq1, seq2):
    """
    Return all items in both seq1 and seq2;
    a set(seq1) & set(seq2) woud work too, but sets are randomly
    ordered, so any platform-dependent directory order would be lost
    """
    return [item for item in seq1 if item in seq2]
```

```

def comparetrees(dir1, dir2, diffs, verbose=False):
    """
    Compare all subdirectories and files in two directory trees;
    uses binary files to prevent Unicode decoding and endline transforms,
    as trees might contain arbitrary binary files as well as arbitrary text;
    may need bytes listdir arg for undecodable filenames on some platforms
    """
    # compare file name lists
    print('-' * 20)
    names1 = os.listdir(dir1)
    names2 = os.listdir(dir2)
    if not dirdiff.comparedirs(dir1, dir2, names1, names2):
        diffs.append('unique files at %s - %s' % (dir1, dir2))

    print('Comparing contents')
    common = intersect(names1, names2)
    missed = common[:]

    # compare contents of files in common
    for name in common:
        path1 = os.path.join(dir1, name)
        path2 = os.path.join(dir2, name)
        if os.path.isfile(path1) and os.path.isfile(path2):
            missed.remove(name)
            file1 = open(path1, 'rb')
            file2 = open(path2, 'rb')
            while True:
                bytes1 = file1.read(blocksize)
                bytes2 = file2.read(blocksize)
                if (not bytes1) and (not bytes2):
                    if verbose: print(name, 'matches')
                    break
                if bytes1 != bytes2:
                    diffs.append('files differ at %s - %s' % (path1, path2))
                    print(name, 'DIFFERS')
                    break

    # recur to compare directories in common
    for name in common:
        path1 = os.path.join(dir1, name)
        path2 = os.path.join(dir2, name)
        if os.path.isdir(path1) and os.path.isdir(path2):
            missed.remove(name)
            comparetrees(path1, path2, diffs, verbose)

    # same name but not both files or dirs?
    for name in missed:
        diffs.append('files missed at %s - %s: %s' % (dir1, dir2, name))
        print(name, 'DIFFERS')

if __name__ == '__main__':
    dir1, dir2 = dirdiff.getargs()
    diffs = []

```

```

comparetrees(dir1, dir2, diffs, True)      # changes diffs in-place
print('=' * 40)                          # walk, report diffs list
if not diffs:
    print('No diffs found.')
else:
    print('Diffs found:', len(diffs))
    for diff in diffs: print('-', diff)

```

At each directory in the tree, this script simply runs the `dirdiff` tool to detect unique names, and then compares names in common by intersecting directory lists. It uses recursive function calls to traverse the tree and visits subdirectories only after comparing all the files at each level so that the output is more coherent to read (the trace output for subdirectories appears after that for files; it is not intermixed).

Notice the `misses` list, added in the third edition of this book; it's very unlikely, but not impossible, that the same name might be a file in one directory and a subdirectory in the other. Also notice the `blocksize` variable; much like the tree copy script we saw earlier, instead of blindly reading entire files into memory all at once, we limit each read to grab up to 1 MB at a time, just in case any files in the directories are too big to be loaded into available memory. Without this limit, I ran into `MemoryError` exceptions on some machines with a prior version of this script that read both files all at once, like this:

```

bytes1 = open(path1, 'rb').read()
bytes2 = open(path2, 'rb').read()
if bytes1 == bytes2: ...

```

This code was simpler, but is less practical for very large files that can't fit into your available memory space (consider CD and DVD image files, for example). In the new version's loop, the file reads return what is left when there is less than 1 MB present or remaining and return empty strings at end-of-file. Files match if all blocks read are the same, and they reach end-of-file at the same time.

We're also dealing in binary files and byte strings again to suppress Unicode decoding and end-line translations for file content, because trees may contain arbitrary binary and text files. The usual note about changing this to pass byte strings to `os.listdir` on platforms where filenames may generate Unicode decoding errors applies here as well (e.g. pass `dir1.encode()`). On some platforms, you may also want to detect and skip certain kinds of special files in order to be fully general, but these were not in my trees, so they are not in my script.

One minor change for the fourth edition of this book: `os.listdir` results are now gathered just once per subdirectory and passed along, to avoid extra calls in `dirdiff`—not a huge win, but every cycle counts on the pitifully underpowered netbook I used when writing this edition.

Running the Script

Since we've already studied the tree-walking tools this script employs, let's jump right into a few example runs. When run on identical trees, status messages scroll during the traversal, and a `No diffs found.` message appears at the end:

```
C:\...\PP4E\System\Filetools> diffall.py C:\temp\PP3E\Examples copytemp > diffs.txt
C:\...\PP4E\System\Filetools> type diffs.txt | more
-----
Comparing C:\temp\PP3E\Examples to copytemp
Directory lists are identical
Comparing contents
README-root.txt matches
-----
Comparing C:\temp\PP3E\Examples\PP3E to copytemp\PP3E
Directory lists are identical
Comparing contents
echoEnvironment.pyw matches
LaunchBrowser.pyw matches
Launcher.py matches
Launcher.pyc matches
...over 2,000 more lines omitted...
-----
Comparing C:\temp\PP3E\Examples\PP3E\TempParts to copytemp\PP3E\TempParts
Directory lists are identical
Comparing contents
109_0237.JPG matches
lawnlake1-jan-03.jpg matches
part-001.txt matches
part-002.html matches
=====
No diffs found.
```

I usually run this with the `verbose` flag passed in as `True`, and redirect output to a file (for big trees, it produces too much output to scroll through comfortably); use `False` to watch fewer status messages fly by. To show how differences are reported, we need to generate a few; for simplicity, I'll manually change a few files scattered about one of the trees, but you could also run a global search-and-replace script like the one we'll write later in this chapter. While we're at it, let's remove a few common files so that directory uniqueness differences show up on the scope, too; the last two removal commands in the following will generate one difference in the same directory in different trees:

```
C:\...\PP4E\System\Filetools> notepad copytemp\PP3E\README-PP3E.txt
C:\...\PP4E\System\Filetools> notepad copytemp\PP3E\System\Filetools\commands.py
C:\...\PP4E\System\Filetools> notepad C:\temp\PP3E\Examples\PP3E\__init__.py
C:\...\PP4E\System\Filetools> del copytemp\PP3E\System\Filetools\cpall_visitor.py
C:\...\PP4E\System\Filetools> del copytemp\PP3E\Launcher.py
C:\...\PP4E\System\Filetools> del C:\temp\PP3E\Examples\PP3E\PyGadgets.py
```

Now, rerun the comparison walker to pick out differences and redirect its output report to a file for easy inspection. The following lists just the parts of the output report that

identify differences. In typical use, I inspect the summary at the bottom of the report first, and then search for the strings "DIFF" and "unique" in the report's text if I need more information about the differences summarized; this interface could be much more user-friendly, of course, but it does the job for me:

```
C:\...\PP4E\System\Filetools> diffall.py C:\temp\PP3E\Examples copytemp > diff2.txt
C:\...\PP4E\System\Filetools> notepad diff2.txt
-----
Comparing C:\temp\PP3E\Examples to copytemp
Directory lists are identical
Comparing contents
README-root.txt matches
-----
Comparing C:\temp\PP3E\Examples\PP3E to copytemp\PP3E
Files unique to C:\temp\PP3E\Examples\PP3E
... Launcher.py
Files unique to copytemp\PP3E
... PyGadgets.py
Comparing contents
echoEnvironment.pyw matches
LaunchBrowser.pyw matches
Launcher.pyc matches
...more omitted...
PyGadgets_bar.pyw matches
README-PP3E.txt DIFFERS
todos.py matches
tounix.py matches
__init__.py DIFFERS
__init__.pyc matches
-----
Comparing C:\temp\PP3E\Examples\PP3E\System\Filetools to copytemp\PP3E\System\Filetools
Files unique to C:\temp\PP3E\Examples\PP3E\System\Filetools
... cpall_visitor.py
Comparing contents
commands.py DIFFERS
cpall.py matches
...more omitted...
-----
Comparing C:\temp\PP3E\Examples\PP3E\TempParts to copytemp\PP3E\TempParts
Directory lists are identical
Comparing contents
109_0237.JPG matches
lawnlake1-jan-03.jpg matches
part-001.txt matches
part-002.html matches
=====
Diffs found: 5
- unique files at C:\temp\PP3E\Examples\PP3E - copytemp\PP3E
- files differ at C:\temp\PP3E\Examples\PP3E\README-PP3E.txt -
    copytemp\PP3E\README-PP3E.txt
- files differ at C:\temp\PP3E\Examples\PP3E\__init__.py -
    copytemp\PP3E\__init__.py
- unique files at C:\temp\PP3E\Examples\PP3E\System\Filetools -
    copytemp\PP3E\System\Filetools
```

```
- files differ at C:\temp\PP3E\Examples\PP3E\System\Filetools\commands.py -
copytemp\PP3E\System\Filetools\commands.py
```

I added line breaks and tabs in a few of these output lines to make them fit on this page, but the report is simple to understand. In a tree with 1,430 files and 185 directories, we found five differences—the three files we changed by edits, and the two directories we threw out of sync with the three removal commands.

Verifying Backups

So how does this script placate CD backup paranoia? To double-check my CD writer's work, I run a command such as the following. I can also use a command like this to find out what has been changed since the last backup. Again, since the CD is "G:" on my machine when plugged in, I provide a path rooted there; use a root such as `/dev/cdrom` or `/mnt/cdrom` on Linux:

```
C:\...\PP4E\System\Filetools> python diffall.py Examples g:\PP3E\Examples > diff0226
C:\...\PP4E\System\Filetools> more diff0226
...output omitted...
```

The CD spins, the script compares, and a summary of differences appears at the end of the report. For an example of a full difference report, see the file `diff*.txt` files in the book's examples distribution package. And to be *really* sure, I run the following global comparison command to verify the entire book development tree backed up to a memory stick (which works just like a CD in terms of the filesystem):

```
C:\...\PP4E\System\Filetools> diffall.py F:\writing-backups\feb-26-10\dev
                                         C:\Users\mark\Stuff\Books\4E\PP4E\dev > diff3.txt
C:\...\PP4E\System\Filetools> more diff3.txt
-----
Comparing F:\writing-backups\feb-26-10\dev to C:\Users\mark\Stuff\Books\4E\PP4E\dev
Directory lists are identical
Comparing contents
ch00.doc DIFFERS
ch01.doc matches
ch02.doc DIFFERS
ch03.doc matches
ch04.doc DIFFERS
ch05.doc matches
ch06.doc DIFFERS
...more output omitted...
-----
Comparing F:\writing-backups\feb-26-10\dev\Examples\PP4E\System\Filetools to C:\
Files unique to C:\Users\mark\Stuff\Books\4E\PP4E\dev\Examples\PP4E\System\Filetools
... copytemp
... cpall.py
... diff2.txt
... diff3.txt
... diffall.py
... diffs.txt
... dirdiff.py
... dirdiff.pyc
```

```

Comparing contents
bigext-tree.py matches
bigpy-dir.py matches
...more output omitted...
=====
Diffs found: 7
- files differ at F:\writing-backups\feb-26-10\dev\ch00.doc -
  C:\Users\mark\Stuff\Books\4E\PP4E\dev\ch00.doc
- files differ at F:\writing-backups\feb-26-10\dev\ch02.doc -
  C:\Users\mark\Stuff\Books\4E\PP4E\dev\ch02.doc
- files differ at F:\writing-backups\feb-26-10\dev\ch04.doc -
  C:\Users\mark\Stuff\Books\4E\PP4E\dev\ch04.doc
- files differ at F:\writing-backups\feb-26-10\dev\ch06.doc -
  C:\Users\mark\Stuff\Books\4E\PP4E\dev\ch06.doc
- files differ at F:\writing-backups\feb-26-10\dev\TOC.txt -
  C:\Users\mark\Stuff\Books\4E\PP4E\dev\TOC.txt
- unique files at F:\writing-backups\feb-26-10\dev\Examples\PP4E\System\Filetools -
  C:\Users\mark\Stuff\Books\4E\PP4E\dev\Examples\PP4E\System\Filetools
- files differ at F:\writing-backups\feb-26-10\dev\Examples\PP4E\Tools\visitor.py -
  C:\Users\mark\Stuff\Books\4E\PP4E\dev\Examples\PP4E\Tools\visitor.py

```

This particular run indicates that I've added a few examples and changed some chapter files since the last backup; if run immediately after a backup, nothing should show up on `diffall` radar except for any files that cannot be copied in general. This global comparison can take a few minutes. It performs byte-for-byte comparisons of all chapter files and screenshots, the examples tree, and more, but it's an accurate and complete verification. Given that this book development tree contained many files, a more manual verification procedure without Python's help would be utterly impossible.

After writing this script, I also started using it to verify full automated backups of my laptops onto an external hard-drive device. To do so, I run the `cpall` copy script we wrote earlier in the preceding section of this chapter, and then the comparison script developed here to check results and get a list of files that didn't copy correctly. The last time I did this, this procedure copied and compared 225,000 files and 15,000 directories in 20 GB of space—not the sort of task that lends itself to manual labor!

Here are the magic incantations on my Windows laptop. `f:\` is a partition on my external hard drive, and you shouldn't be surprised if each of these commands runs for half an hour or more on currently common hardware. A drag-and-drop copy takes at least as long (assuming it works at all!):

```
C:\...\System\Filetools> cpall.py c:\ f:\ > f:\copy-log.txt
C:\...\System\Filetools> diffall.py f:\ c:\ > f:\diff-log.txt
```

Reporting Differences and Other Ideas

Finally, it's worth noting that this script still only *detects* differences in the tree but does not give any further details about individual file differences. In fact, it simply loads and compares the binary contents of corresponding files with string comparisons. It's a simple yes/no result.

If and when I need more details about how two reported files actually differ, I either edit the files or run the file-comparison command on the host platform (e.g., `fc` on Windows/DOS, `diff` or `cmp` on Unix and Linux). That's not a portable solution for this last step; but for my purposes, just finding the differences in a 1,400-file tree was much more critical than reporting which lines differ in files flagged in the report.

Of course, since we can always run shell commands in Python, this last step could be automated by spawning a `diff` or `fc` command with `os.popen` as differences are encountered (or after the traversal, by scanning the report summary). The output of these system calls could be displayed verbatim, or parsed for relevant parts.

We also might try to do a bit better here by opening true text files in text mode to ignore line-terminator differences caused by transferring across platforms, but it's not clear that such differences should be ignored (what if the caller wants to know whether line-end markers have been changed?). For example, after downloading a website with an FTP script we'll meet in [Chapter 13](#), the `diffall` script detected a discrepancy between the local copy of a file and the one at the remote server. To probe further, I simply ran some interactive Python code:

```
>>> a = open('lp2e-updates.html', 'rb').read()
>>> b = open(r'C:\Mark\WEBSITE\public_html\lp2e-updates.html', 'rb').read()
>>> a == b
False
```

This verifies that there really is a binary difference in the downloaded and local versions of the file; to see whether it's because a Unix or DOS line end snuck into the file, try again in text mode so that line ends are all mapped to the standard `\n` character:

```
>>> a = open('lp2e-updates.html', 'r').read()
>>> b = open(r'C:\Mark\WEBSITE\public_html\lp2e-updates.html', 'r').read()
>>> a == b
True
```

Sure enough; now, to find where the difference is, the following code checks character by character until the first mismatch is found (in binary mode, so we retain the difference):

```
>>> a = open('lp2e-updates.html', 'rb').read()
>>> b = open(r'C:\Mark\WEBSITE\public_html\lp2e-updates.html', 'rb').read()

>>> for (i, (ac, bc)) in enumerate(zip(a, b)):
...     if ac != bc:
...         print(i, repr(ac), repr(bc))
...         break
...
37966 '\r' '\n'
```

This means that at byte offset 37,966, there is a `\r` in the downloaded file, but a `\n` in the local copy. This line has a DOS line end in one and a Unix line end in the other. To see more, print text around the mismatch:

```
>>> for (i, (ac, bc)) in enumerate(zip(a, b)):
...     if ac != bc:
...         print(i, repr(ac), repr(bc))
...         print(repr(a[i-20:i+20]))
...         print(repr(b[i-20:i+20]))
...         break
...
37966 '\r' '\n'
're>\r\nndef min(*args):\r\n    tmp = list(arg'
're>\r\nndef min(*args):\n    tmp = list(args'
```

Apparently, I wound up with a Unix line end at one point in the local copy and a DOS line end in the version I downloaded—the combined effect of the text mode used by the download script itself (which translated `\n` to `\r\n`) and years of edits on both Linux and Windows PDAs and laptops (I probably coded this change on Linux and copied it to my local Windows copy in binary mode). Code such as this could be integrated into the `diffall` script to make it more intelligent about text files and difference reporting.

Because Python excels at processing files and strings, it's even possible to go one step further and code a Python equivalent of the `fc` and `diff` commands. In fact, much of the work has already been done; the standard library module `difflib` could make this task simple. See the Python library manual for details and usage examples.

We could also be smarter by avoiding the load and compare steps for files that differ in size, and we might use a smaller block size to reduce the script's memory requirements. For most trees, such optimizations are unnecessary; reading multimegabyte files into strings is very fast in Python, and garbage collection reclaims the space as you go.

Since such extensions are beyond both this script's scope and this chapter's size limits, though, they will have to await the attention of a curious reader (this book doesn't have formal exercises, but that almost sounds like one, doesn't it?). For now, let's move on to explore ways to code one more common directory task: search.

Searching Directory Trees

Engineers love to change things. As I was writing this book, I found it almost *irresistible* to move and rename directories, variables, and shared modules in the book examples tree whenever I thought I'd stumbled onto a more coherent structure. That was fine early on, but as the tree became more intertwined, this became a maintenance nightmare. Things such as program directory paths and module names were hardcoded all over the place—in package import statements, program startup calls, text notes, configuration files, and more.

One way to repair these references, of course, is to edit every file in the directory by hand, searching each for information that has changed. That's so tedious as to be utterly impossible in this book's examples tree, though; the examples of the prior edition contained 186 directories and 1,429 files! Clearly, I needed a way to automate updates after

changes. There are a variety of solutions to such goals—from shell commands, to find operations, to custom tree walkers, to general-purpose frameworks. In this and the next section, we’ll explore each option in turn, just as I did while refining solutions to this real-world dilemma.

Greps and Globs and Finds

If you work on Unix-like systems, you probably already know that there is a standard way to search files for strings on such platforms—the command-line program `grep` and its relatives list all lines in one or more files containing a string or string pattern.¹¹ Given that shells expand (i.e., “glob”) filename patterns automatically, a command such as the following will search a single directory’s Python files for a string named on the command line (this uses the `grep` command installed with the Cygwin Unix-like system for Windows that I described in the prior chapter):

```
C:\...\PP4E\System\Filetools> c:\cygwin\bin\grep.exe walk *.py
bigext-tree.py:for (thisDir, subsHere, filesHere) in os.walk(dirname):
bigpy-path.py:   for (thisDir, subsHere, filesHere) in os.walk(srkdir):
bigpy-tree.py:for (thisDir, subsHere, filesHere) in os.walk(dirname):
```

As we’ve seen, we can often accomplish the same within a Python script by running such a shell command with `os.system` or `os.popen`. And if we search its results manually, we can also achieve similar results with the Python `glob` module we met in [Chapter 4](#); it expands a filename pattern into a list of matching filename strings much like a shell:

```
C:\...\PP4E\System\Filetools> python
>>> import os
>>> for line in os.popen(r'c:\cygwin\bin\grep.exe walk *.py'):
...     print(line, end='')

bigext-tree.py:for (thisDir, subsHere, filesHere) in os.walk(dirname):
bigpy-path.py:   for (thisDir, subsHere, filesHere) in os.walk(srkdir):
bigpy-tree.py:for (thisDir, subsHere, filesHere) in os.walk(dirname):

>>> from glob import glob
>>> for filename in glob('*.py'):
...     if 'walk' in open(filename).read():
...         print(filename)

...
bigext-tree.py
bigpy-path.py
bigpy-tree.py
```

Unfortunately, these tools are generally limited to a single directory. `glob` can visit multiple directories given the right sort of pattern string, but it’s not a general directory walker of the sort I need to maintain a large examples tree. On Unix-like systems, a `find` shell command can go the extra mile to traverse an entire directory tree. For

¹¹ In fact, the act of searching files often goes by the colloquial name “grepping” among developers who have spent any substantial time in the Unix ghetto.

instance, the following Unix command line would pinpoint lines and files at and below the current directory that mention the string `popen`:

```
find . -name "*.py" -print -exec fgrep popen {} \;
```

If you happen to have a Unix-like `find` command on every machine you will ever use, this is one way to process directories.

Rolling Your Own `find` Module

But if you don't happen to have a Unix `find` on all your computers, not to worry—it's easy to code a portable one in Python. Python itself used to have a `find` module in its standard library, which I used frequently in the past. Although that module was removed between the second and third editions of this book, the newer `os.walk` makes writing your own simple. Rather than lamenting the demise of a module, I decided to spend 10 minutes coding a custom equivalent.

[Example 6-13](#) implements a `find` utility in Python, which collects all matching filenames in a directory tree. Unlike `glob.glob`, its `find` automatically matches through an entire tree. And unlike the tree walk structure of `os.walk`, we can treat `find`.`find` results as a simple linear group.

Example 6-13. PP4E\Tools\find.py

```
#!/usr/bin/python
"""
#####
Return all files matching a filename pattern at and below a root directory;

custom version of the now deprecated find module in the standard library:
import as "PP4E.Tools.find"; like original, but uses os.walk loop, has no
support for pruning subdirs, and is runnable as a top-level script;

find() is a generator that uses the os.walk() generator to yield just
matching filenames: use findlist() to force results list generation;
#####
"""

import fnmatch, os

def find(pattern, startdir=os.curdir):
    for (thisDir, subsHere, filesHere) in os.walk(startdir):
        for name in subsHere + filesHere:
            if fnmatch.fnmatch(name, pattern):
                fullpath = os.path.join(thisDir, name)
                yield fullpath

def findlist(pattern, startdir=os.curdir, dosort=False):
    matches = list(find(pattern, startdir))
    if dosort: matches.sort()
    return matches
```

```
if __name__ == '__main__':
    import sys
    namepattern, startdir = sys.argv[1], sys.argv[2]
    for name in find(namepattern, startdir): print(name)
```

There's not much to this file—it's largely just a minor extension to `os.walk`—but calling its `find` function provides the same utility as both the deprecated `find` standard library module and the Unix utility of the same name. It's also much more portable, and noticeably easier than repeating all of this file's code every time you need to perform a find-type search. Because this file is instrumented to be both a script and a library, it can also be both run as a command-line tool or called from other programs.

For instance, to process every Python file in the directory tree rooted one level up from the current working directory, I simply run the following command line from a system console window. Run this yourself to watch its progress; the script's standard output is piped into the `more` command to page it here, but it can be piped into any processing program that reads its input from the standard input stream:

```
C:\...\PP4E\Tools> python find.py *.py .. | more
..\LaunchBrowser.py
..\Launcher.py
..\__init__.py
..\Preview\attachgui.py
..\Preview\customizegui.py
...more lines omitted...
```

For more control, run the following sort of Python code from a script or interactive prompt. In this mode, you can apply any operation to the found files that the Python language provides:

```
C:\...\PP4E\System\Filetools> python
>>> from PP4E.Tools import find
...      # or just import find if in cwd
>>> for filename in find.find('*.*', '..'):
...     if 'walk' in open(filename).read():
...         print(filename)
...
..\Launcher.py
..\System\Filetools\bigext-tree.py
..\System\Filetools\bigpy-path.py
..\System\Filetools\bigpy-tree.py
..\Tools\cleanpyc.py
..\Tools\find.py
..\Tools\visitor.py
```

Notice how this avoids having to recode the nested loop structure required for `os.walk` every time you want a list of matching file names; for many use cases, this seems conceptually simpler. Also note that because this finder is a generator function, your script doesn't have to wait until all matching files have been found and collected; `os.walk` yields results as it goes, and `find.find` yields matching files among that set.

Here's a more complex example of our `find` module at work: the following system command line lists all Python files in directory `C:\temp\PP3E` whose names begin with

the letter *q* or *t*. Note how `find` returns full directory paths that begin with the start directory specification:

```
C:\...\PP4E\Tools> find.py [qx]*.py C:\temp\PP3E
C:\temp\PP3E\Examples\PP3E\Database\SQLscripts\querydb.py
C:\temp\PP3E\Examples\PP3E\Gui\Tools\queuetest-gui-class.py
C:\temp\PP3E\Examples\PP3E\Gui\Tools\queuetest-gui.py
C:\temp\PP3E\Examples\PP3E\Gui\Tour\quitter.py
C:\temp\PP3E\Examples\PP3E\Internet\Other\Grail\Question.py
C:\temp\PP3E\Examples\PP3E\Internet\Other\XML\xmlrpc.py
C:\temp\PP3E\Examples\PP3E\System\Threads\queuetest.py
```

And here's some Python code that does the same `find` but also extracts base names and file sizes for each file found:

```
C:\...\PP4E\Tools> python
>>> import os
>>> from find import find
>>> for name in find('[qx]*.py', r'C:\temp\PP3E'):
...     print(os.path.basename(name), os.path.getsize(name))
...
querydb.py 635
queuetest-gui-class.py 1152
queuetest-gui.py 963
quitter.py 801
Question.py 817
xmlrpc.py 705
queuetest.py 1273
```

The `fnmatch` module

To achieve such code economy, the `find` module calls `os.walk` to walk the tree and simply yields matching filenames along the way. New here, though, is the `fnmatch` module—yet another Python standard library module that performs Unix-like pattern matching against filenames. This module supports common operators in name pattern strings: `*` to match any number of characters, `?` to match any single character, and `[...]` and `[!...]` to match any character inside the bracket pairs or not; other characters match themselves. Unlike the `re` module, `fnmatch` supports only common Unix shell matching operators, not full-blown regular expression patterns; we'll see why this distinction matters in [Chapter 19](#).

Interestingly, Python's `glob.glob` function also uses the `fnmatch` module to match names: it combines `os.listdir` and `fnmatch` to match in directories in much the same way our `find.find` combines `os.walk` and `fnmatch` to match in trees (though `os.walk` ultimately uses `os.listdir` as well). One ramification of all this is that you can pass byte strings for both pattern and start-directory to `find.find` if you need to suppress Unicode filename decoding, just as you can for `os.walk` and `glob.glob`; you'll receive byte strings for filenames in the result. See [Chapter 4](#) for more details on Unicode filenames.

By comparison, `find.find` with just “`*`” for its name pattern is also roughly equivalent to platform-specific directory tree listing shell commands such as `dir /B /S` on DOS and Windows. Since all files match “`*`”, this just exhaustively generates all the file names in a tree with a single traversal. Because we can usually run such shell commands in a Python script with `os.popen`, the following do the same work, but the first is inherently nonportable and must start up a separate program along the way:

```
>>> import os  
>>> for line in os.popen('dir /B /S'): print(line, end='')  
  
>>> from PP4E.Tools.find import find  
>>> for name in find(pattern='*', startdir='.'): print(name)
```

Watch for this utility to show up in action later in this chapter and book, including an arguably strong showing in the next section and a cameo appearance in the Grep dialog of [Chapter 11](#)’s PyEdit text editor GUI, where it will serve a central role in a threaded external files search tool. The standard library’s `find` module may be gone, but it need not be forgotten.



In fact, you *must* pass a `bytes` pattern string for a `bytes` filename to `fnnmatch` (or pass both as `str`), because the `re` pattern matching module it uses does not allow the string types of subject and pattern to be mixed. This rule is inherited by our `find.find` for directory and pattern. See [Chapter 19](#) for more on `re`.

Curiously, the `fnnmatch` module in Python 3.1 also converts a `bytes` pattern string to and from Unicode `str` in order to perform internal text processing, using the Latin-1 encoding. This suffices for many contexts, but may not be entirely sound for some encodings which do not map to Latin-1 cleanly. `sys.getfilesystemencoding` might be a better encoding choice in such contexts, as this reflects the underlying file system’s constraints (as we learned in [Chapter 4](#), `sys.getdefaultencoding` reflects file content, not names).

In the absence of `bytes`, `os.walk` assumes filenames follow the platform’s convention and does not ignore decoding errors triggered by `os.listdir`. In the “grep” utility of [Chapter 11](#)’s PyEdit, this picture is further clouded by the fact that a `str` pattern string from a GUI would have to be encoded to `bytes` using a potentially inappropriate encoding for some files present. See `fnnmatch.py` and `os.py` in Python’s library and the Python library manual for more details. Unicode can be a very subtle affair.

Cleaning Up Bytecode Files

The `find` module of the prior section isn’t quite the general string searcher we’re after, but it’s an important first step—it collects files that we can then search in an automated script. In fact, the act of collecting matching files in a tree is enough by itself to support a wide variety of day-to-day system tasks.

For example, one of the other common tasks I perform on a regular basis is removing all the bytecode files in a tree. Because these are not always portable across major Python releases, it's usually a good idea to ship programs without them and let Python create new ones on first imports. Now that we're expert `os.walk` users, we could cut out the middleman and use it directly. [Example 6-14](#) codes a portable and general command-line tool, with support for arguments, exception processing, tracing, and list-only mode.

Example 6-14. PP4E\Tools\cleanpyc.py

```
"""
delete all .pyc bytecode files in a directory tree: use the
command line arg as root if given, else current working dir
"""

import os, sys
findonly = False
rootdir = os.getcwd() if len(sys.argv) == 1 else sys.argv[1]

found = removed = 0
for (thisDirLevel, subsHere, filesHere) in os.walk(rootdir):
    for filename in filesHere:
        if filename.endswith('.pyc'):
            fullname = os.path.join(thisDirLevel, filename)
            print('=>', fullname)
            if not findonly:
                try:
                    os.remove(fullname)
                    removed += 1
                except:
                    type, inst = sys.exc_info()[:2]
                    print('*'*4, 'Failed:', filename, type, inst)
            found += 1

print('Found', found, 'files, removed', removed)
```

When run, this script walks a directory tree (the CWD by default, or else one passed in on the command line), deleting any and all bytecode files along the way:

```
C:\...\Examples\PP4E> Tools\cleanpyc.py
=> C:\Users\mark\Stuff\Books\4E\PP4E\dev\Examples\PP4E\__init__.pyc
=> C:\Users\mark\Stuff\Books\4E\PP4E\dev\Examples\PP4E\Preview\initdata.pyc
=> C:\Users\mark\Stuff\Books\4E\PP4E\dev\Examples\PP4E\Preview\make_db_file.pyc
=> C:\Users\mark\Stuff\Books\4E\PP4E\dev\Examples\PP4E\Preview\manager.pyc
=> C:\Users\mark\Stuff\Books\4E\PP4E\dev\Examples\PP4E\Preview\person.pyc
...more lines here...
Found 24 files, removed 24

C:\...\PP4E\Tools> cleanpyc.py .
=> .\find.pyc
=> .\visitor.pyc
=> .\__init__.pyc
Found 3 files, removed 3
```

This script works, but it's a bit more manual and code-y than it needs to be. In fact, now that we also know about find operations, writing scripts based upon them is almost trivial when we just need to match filenames. [Example 6-15](#), for instance, falls back on spawning shell find commands if you have them.

Example 6-15. PP4E\Tools\cleanpyc-find-shell.py

```
"""
find and delete all "*.pyc" bytecode files at and below the directory
named on the command-line; assumes a nonportable Unix-like find command
"""

import os, sys

rundir = sys.argv[1]
if sys.platform[:3] == 'win':
    findcmd = r'c:\cygwin\bin\find %s -name "*.pyc" -print' % rundir
else:
    findcmd = 'find %s -name "*.pyc" -print' % rundir
print(findcmd)

count = 0
for fileline in os.popen(findcmd):                      # for all result lines
    count += 1                                         # have \n at the end
    print(fileline, end='')
    os.remove(fileline.rstrip())

print('Removed %d .pyc files' % count)
```

When run, files returned by the shell command are removed:

```
C:\...\PP4E\Tools> cleanpyc-find-shell.py .
c:\cygwin\bin\find . -name "*.pyc" -print
./find.pyc
./visitor.pyc
./__init__.pyc
Removed 3 .pyc files
```

This script uses `os.popen` to collect the output of a Cygwin `find` program installed on one of my Windows computers, or else the standard `find` tool on the Linux side. It's also *completely nonportable* to Windows machines that don't have the Unix-like `find` program installed, and that includes other computers of my own (not to mention those throughout most of the world at large). As we've seen, spawning shell commands also incurs performance penalties for starting a new program.

We can do much better on the portability and performance fronts and still retain code simplicity, by applying the `find` tool we wrote in Python in the prior section. The new script is shown in [Example 6-16](#).

Example 6-16. PP4E\Tools\cleanpyc-find-py.py

```
"""
find and delete all "*.pyc" bytecode files at and below the directory
named on the command-line; this uses a Python-coded find utility, and
```

```

so is portable; run this to delete .pyc's from an old Python release;
"""

import os, sys, find  # here, gets Tools.find

count = 0
for filename in find.find('.pyc', sys.argv[1]):
    count += 1
    print(filename)
    os.remove(filename)

print('Removed %d .pyc files' % count)

```

When run, all bytecode files in the tree rooted at the passed-in directory name are removed as before; this time, though, our script works just about everywhere Python does:

```

C:\...\PP4E\Tools> cleanpyc-find-py.py .
.\find.pyc
.\visitor.pyc
.\__init__.pyc
Removed 3 .pyc files

```

This works portably, and it avoids external program startup costs. But `find` is really just half the story—it collects files matching a name pattern but doesn’t search their content. Although extra code can add such searching to a `find`’s result, a more manual approach can allow us to tap into the search process more directly. The next section shows how.

A Python Tree Searcher

After experimenting with greps and globs and finds, in the end, to help ease the task of performing global searches on all platforms I might ever use, I wound up coding a task-specific Python script to do most of the work for me. [Example 6-17](#) employs the following standard Python tools that we met in the preceding chapters: `os.walk` to visit files in a directory, `os.path.splitext` to skip over files with binary-type extensions, and `os.path.join` to portably combine a directory path and filename.

Because it’s pure Python code, it can be run the same way on both Linux and Windows. In fact, it should work on any computer where Python has been installed. Moreover, because it uses direct system calls, it will likely be faster than approaches that rely on underlying shell commands.

Example 6-17. PP4E\Tools\search_all.py

```

"""
#####
# Use: "python ...\\Tools\\search_all.py dir string".
# Search all files at and below a named directory for a string; uses the
# os.walk interface, rather than doing a find.find to collect names first;
# similar to calling visitfile for each find.find result for "*" pattern;

```

```
#####
"""

import os, sys
listonly = False
textexts = ['.py', '.pyw', '.txt', '.c', '.h'] # ignore binary files

def searcher(startdir, searchkey):
    global fcount, vcount
    fcount = vcount = 0
    for (thisDir, dirsHere, filesHere) in os.walk(startdir):
        for fname in filesHere: # do non-dir files here
            fpath = os.path.join(thisDir, fname) # fnames have no dirpath
            visitfile(fpath, searchkey)

def visitfile(fpath, searchkey): # for each non-dir file
    global fcount, vcount # search for string
    print(vcount+1, '=>', fpath) # skip protected files
    try:
        if not listonly:
            if os.path.splitext(fpath)[1] not in textexts:
                print('Skipping', fpath)
            elif searchkey in open(fpath).read():
                input('%s has %s' % (fpath, searchkey))
                fcount += 1
    except:
        print('Failed:', fpath, sys.exc_info()[0])
    vcount += 1

if __name__ == '__main__':
    searcher(sys.argv[1], sys.argv[2])
    print('Found in %d files, visited %d' % (fcount, vcount))
```

Operationally, this script works roughly the same as calling its `visitfile` function for every result generated by our `find.find` tool with a pattern of “*”; but because this version is specific to searching content it can better tailored for its goal. Really, this equivalence holds only because a “*” pattern invokes an exhaustive traversal in `find.find`, and that’s all that this new script’s `searcher` function does. The finder is good at selecting specific file types, but this script benefits from a more custom single traversal.

When run standalone, the search key is passed on the command line; when imported, clients call this module’s `searcher` function directly. For example, to search (that is, grep) for all appearances of a string in the book examples tree, I run a command line like this in a DOS or Unix shell:

```
C:\PP4E> Tools\search_all.py . mimetypes
1 => .\LaunchBrowser.py
2 => .\Launcher.py
3 => .\Launch_PyDemos.pyw
4 => .\Launch_PyGadgets_bar.pyw
5 => .\__init__.py
6 => .\__init__.pyc
```

```
Skipping .\__init__.pyc
7 => .\Preview\attachgui.py
8 => .\Preview\bob.pkl
Skipping .\Preview\bob.pkl
...more lines omitted: pauses for Enter key press at matches...
Found in 2 files, visited 184
```

The script lists each file it checks as it goes, tells you which files it is skipping (names that end in extensions not listed in the variable `textexts` that imply binary data), and pauses for an Enter key press each time it announces a file containing the search string. The `search_all` script works the same way when it is *imported* rather than run, but there is no final statistics output line (`fcount` and `vcount` live in the module and so would have to be imported to be inspected here):

```
C:\...\PP4E\dev\Examples\PP4E> python
>>> import Tools.search_all
>>> search_all.searcher(r'C:\temp\PP3E\Examples', 'mimetypes')
...more lines omitted: 8 pauses for Enter key press along the way...
>>> search_all.fcount, search_all.vcount      # matches, files
(8, 1429)
```

However launched, this script tracks down all references to a string in an entire directory tree: a name of a changed book examples file, object, or directory, for instance. It's exactly what I was looking for—or at least I thought so, until further deliberation drove me to seek more complete and better structured solutions, the topic of the next section.



Be sure to also see the coverage of regular expressions in [Chapter 19](#). The `search_all` script here searches for a simple string in each file with the `in` string membership expression, but it would be trivial to extend it to search for a regular expression pattern match instead (roughly, just replace `in` with a call to a regular expression object's `search` method). Of course, such a mutation will be much more trivial after we've learned how.

Also notice the `textexts` list in [Example 6-17](#), which attempts to list all possible binary file types: it would be more general and robust to use the `mimetypes` logic we will meet near the end of this chapter in order to guess file content type from its name, but the skips list provides more control and sufficed for the trees I used this script against.

Finally note that for simplicity many of the directory searches in this chapter assume that text is encoded per the underlying platform's Unicode default. They could open text in binary mode to avoid decoding errors, but searches might then be inaccurate because of encoding scheme differences in the raw encoded bytes. To see how to do better, watch for the "grep" utility in [Chapter 11](#)'s PyEdit GUI, which will apply an encoding name to all the files in a searched tree and ignore those text or binary files that fail to decode.

Visitor: Walking Directories “++”

Laziness is the mother of many a framework. Armed with the portable `search_all` script from [Example 6-17](#), I was able to better pinpoint files to be edited every time I changed the book examples tree content or structure. At least initially, in one window I ran `search_all` to pick out suspicious files and edited each along the way by hand in another window.

Pretty soon, though, this became tedious, too. Manually typing filenames into editor commands is no fun, especially when the number of files to edit is large. Since I occasionally have better things to do than manually start dozens of text editor sessions, I started looking for a way to *automatically* run an editor on each suspicious file.

Unfortunately, `search_all` simply prints results to the screen. Although that text could be intercepted with `os.popen` and parsed by another program, a more direct approach that spawns edit sessions during the search may be simpler. That would require major changes to the tree search script as currently coded, though, and make it useful for just one specific purpose. At this point, three thoughts came to mind:

Redundancy

After writing a few directory walking utilities, it became clear that I was rewriting the same sort of code over and over again. Traversals could be even further simplified by wrapping common details for reuse. Although the `os.walk` tool avoids having to write recursive functions, its model tends to foster redundant operations and code (e.g., directory name joins, tracing prints).

Extensibility

Past experience informed me that it would be better in the long run to add features to a general directory searcher as external components, rather than changing the original script itself. Because editing files was just one possible extension (what about automating text replacements, too?), a more general, customizable, and reusable approach seemed the way to go. Although `os.walk` is straightforward to use, its nested loop-based structure doesn’t quite lend itself to customization the way a class can.

Encapsulation

Based on past experience, I also knew that it’s a generally good idea to insulate programs from implementation details as much as possible. While `os.walk` hides the details of recursive traversal, it still imposes a very specific interface on its clients, which is prone to change over time. Indeed it has—as I’ll explain further at the end of this section, one of Python’s tree walkers was removed altogether in 3.X, instantly breaking code that relied upon it. It would be better to hide such dependencies behind a more neutral interface, so that clients won’t break as our needs change.

Of course, if you’ve studied Python in any depth, you know that all these goals point to using an *object-oriented framework* for traversals and searching. [Example 6-18](#) is a

concrete realization of these goals. It exports a general `FileVisitor` class that mostly just wraps `os.walk` for easier use and extension, as well as a generic `SearchVisitor` class that generalizes the notion of directory searches.

By itself, `SearchVisitor` simply does what `search_all` did, but it also opens up the search process to customization—bits of its behavior can be modified by overloading its methods in subclasses. Moreover, its core search logic can be reused everywhere we need to search. Simply define a subclass that adds extensions for a specific task. The same goes for `FileVisitor`—by redefining its methods and using its attributes, we can tap into tree search using OOP coding techniques. As is usual in programming, once you repeat *tactical* tasks often enough, they tend to inspire this kind of *strategic* thinking.

Example 6-18. PP4E\Tools\visitor.py

```
"""
#####
Test: "python ...\\Tools\\visitor.py dir testmask [string]". Uses classes and
subclasses to wrap some of the details of os.walk call usage to walk and search;
testmask is an integer bitmask with 1 bit per available self-test; see also:
visitor_*/.py subclasses use cases; frameworks should generally use__X pseudo
private names, but all names here are exported for use in subclasses and clients;
redefine reset to support multiple independent walks that require subclass updates;
#####
"""

import os, sys

class FileVisitor:
    """
    Visits all nondirectory files below startDir (default '.');
    override visit* methods to provide custom file/dir handlers;
    context arg/attribute is optional subclass-specific state;
    trace switch: 0 is silent, 1 is directories, 2 adds files
    """
    def __init__(self, context=None, trace=2):
        self.fcount = 0
        self.dcount = 0
        self.context = context
        self.trace = trace

    def run(self, startDir=os.curdir, reset=True):
        if reset: self.reset()
        for (thisDir, dirsHere, filesHere) in os.walk(startDir):
            self.visitdir(thisDir)
            for fname in filesHere:                      # for non-dir files
                fpath = os.path.join(thisDir, fname)      # fnames have no path
                self.visitfile(fpath)

    def reset(self):                                     # to reuse walker
        self.fcount = self.dcount = 0                     # for independent walks

    def visitdir(self, dirpath):                         # called for each dir
```

```

        self.dcount += 1                                # override or extend me
        if self.trace > 0: print(dirpath, '...')

    def visitfile(self, filepath):                  # called for each file
        self.fcount += 1                            # override or extend me
        if self.trace > 1: print(self.fcount, '=>', filepath)

class SearchVisitor(FileVisitor):
    """
    Search files at and below startDir for a string;
    subclass: redefine visitmatch, extension lists, candidate as needed;
    subclasses can use testexts to specify file types to search (but can
    also redefine candidate to use mimetypes for text content: see ahead)
    """

    skipexts = []
    testexts = ['.txt', '.py', '.pyw', '.html', '.c', '.h'] # search these exts
    #skipexts = ['.gif', '.jpg', '.pyc', '.o', '.a', '.exe'] # or skip these exts

    def __init__(self, searchkey, trace=2):
        FileVisitor.__init__(self, searchkey, trace)
        self.scount = 0

    def reset(self):                                # on independent walks
        self.scount = 0

    def candidate(self, fname):                      # redef for mimetypes
        ext = os.path.splitext(fname)[1]
        if self.testexts:
            return ext in self.testexts             # in test list
        else:
            return ext not in self.skipexts       # or not in skip list

    def visitfile(self, fname):                      # test for a match
        FileVisitor.visitfile(self, fname)
        if not self.candidate(fname):
            if self.trace > 0: print('Skipping', fname)
        else:
            text = open(fname).read()              # 'rb' if undecodable
            if self.context in text:
                self.visitmatch(fname, text)      # or text.find() != -1
            self.scount += 1

    def visitmatch(self, fname, text):               # process a match
        print('%s has %s' % (fname, self.context))  # override me lower

if __name__ == '__main__':
    # self-test logic
    dolist = 1
    dosearch = 2      # 3=do list and search
    donext = 4        # when next test added

    def selftest(testmask):

```

```

if testmask & dolist:
    visitor = FileVisitor(trace=2)
    visitor.run(sys.argv[2])
    print('Visited %d files and %d dirs' % (visitor.fcount, visitor.dcount))

if testmask & dosearch:
    visitor = SearchVisitor(sys.argv[3], trace=0)
    visitor.run(sys.argv[2])
    print('Found in %d files, visited %d' % (visitor.scount, visitor.fcount))

selftest(int(sys.argv[1]))    # e.g., 3 = dolist | dosearch

```

This module primarily serves to export classes for external use, but it does something useful when run standalone, too. If you invoke it as a script with a test mask of 1 and a root directory name, it makes and runs a `FileVisitor` object and prints an exhaustive listing of every file and directory at and below the root:

```

C:\...\PP4E\Tools> visitor.py 1 C:\temp\PP3E\Examples
C:\temp\PP3E\Examples ...
1 => C:\temp\PP3E\Examples\README-root.txt
C:\temp\PP3E\Examples\PP3E ...
2 => C:\temp\PP3E\Examples\PP3E\echoEnvironment.pyw
3 => C:\temp\PP3E\Examples\PP3E\LaunchBrowser.pyw
4 => C:\temp\PP3E\Examples\PP3E\Launcher.py
5 => C:\temp\PP3E\Examples\PP3E\Launcher.pyc
...more output omitted (pipe into more or a file)...
1424 => C:\temp\PP3E\Examples\PP3E\System\Threads\thread-count.py
1425 => C:\temp\PP3E\Examples\PP3E\System\Threads\thread1.py
C:\temp\PP3E\Examples\PP3E\TempParts ...
1426 => C:\temp\PP3E\Examples\PP3E\TempParts\109_0237.JPG
1427 => C:\temp\PP3E\Examples\PP3E\TempParts\lawnlake1-jan-03.jpg
1428 => C:\temp\PP3E\Examples\PP3E\TempParts\part-001.txt
1429 => C:\temp\PP3E\Examples\PP3E\TempParts\part-002.html
Visited 1429 files and 186 dirs

```

If you instead invoke this script with a 2 as its first command-line argument, it makes and runs a `SearchVisitor` object using the third argument as the search key. This form is similar to running the `search_all.py` script we met earlier, but it simply reports each matching file without pausing:

```

C:\...\PP4E\Tools> visitor.py 2 C:\temp\PP3E\Examples mimetypes
C:\temp\PP3E\Examples\PP3E\extras\LosAlamosAdvancedClass\day1-system\data.txt has mimetypes
C:\temp\PP3E\Examples\PP3E\Internet\Email\mailtools\mailParser.py has mimetypes
C:\temp\PP3E\Examples\PP3E\Internet\Email\mailtools\mailSender.py has mimetypes
C:\temp\PP3E\Examples\PP3E\Internet\Ftp\mirror\downloadflat.py has mimetypes
C:\temp\PP3E\Examples\PP3E\Internet\Ftp\mirror\downloadflat_modular.py has mimetypes
C:\temp\PP3E\Examples\PP3E\Internet\Ftp\mirror\ftptools.py has mimetypes
C:\temp\PP3E\Examples\PP3E\Internet\Ftp\mirror\uploadflat.py has mimetypes
C:\temp\PP3E\Examples\PP3E\System\Media\playfile.py has mimetypes
Found in 8 files, visited 1429

```

Technically, passing this script a first argument of 3 runs *both* a `FileVisitor` and a `SearchVisitor` (two separate traversals are performed). The first argument is really used as a bit mask to select one or more supported self-tests; if a test's bit is on in the binary value of the argument, the test will be run. Because 3 is 011 in binary, it selects both a search (010) and a listing (001). In a more user-friendly system, we might want to be more symbolic about that (e.g., check for `-search` and `-list` arguments), but bit masks work just as well for this script's scope.

As usual, this module can also be used interactively. The following is one way to determine how many files and directories you have in specific directories; the last command walks over your entire drive (after a generally noticeable delay!). See also the “biggest file” example at the start of this chapter for issues such as potential repeat visits not handled by this walker:

```
C:\...\PP4E\Tools> python
>>> from visitor import FileVisitor
>>> V = FileVisitor(trace=0)
>>> V.run(r'C:\temp\PP3E\Examples')
>>> V.dcount, V.fcount
(186, 1429)

>>> V.run('..')                      # independent walk (reset counts)
>>> V.dcount, V.fcount
(19, 181)

>>> V.run('..', reset=False)          # accumulative walk (keep counts)
>>> V.dcount, V.fcount
(38, 362)

>>> V = FileVisitor(trace=0)          # new independent walker (own counts)
>>> V.run(r'C:\\')                  # entire drive: try '/' on Unix-en
>>> V.dcount, V.fcount
(24992, 198585)
```

Although the visitor module is useful by itself for listing and searching trees, it was really designed to be extended. In the rest of this section, let's quickly step through a handful of visitor clients which add more specific tree operations, using normal OO customization techniques.

Editing Files in Directory Trees (Visitor)

After genericizing tree traversals and searches, it's easy to add automatic file editing in a brand-new, separate component. [Example 6-19](#) defines a new `EditVisitor` class that simply customizes the `visitmatch` method of the `SearchVisitor` class to open a text editor on the matched file. Yes, this is the complete program—it needs to do something special only when visiting matched files, and so it needs to provide only that behavior. The rest of the traversal and search logic is unchanged and inherited.

Example 6-19. PP4E\Tools\visitor_edit.py

```
"""
Use: "python ...\\Tools\\visitor_edit.py string rootdir?".
Add auto-editor startup to SearchVisitor in an external subclass component;
Automatically pops up an editor on each file containing string as it traverses;
can also use editor='edit' or 'notepad' on Windows; to use texteditor from
later in the book, try r'python Gui\\TextEditor\\textEditor.py'; could also
send a search command to go to the first match on start in some editors;
"""

import os, sys
from visitor import SearchVisitor

class EditVisitor(SearchVisitor):
    """
    edit files at and below startDir having string
    """
    editor = r'C:\\cygwin\\bin\\vim-nox.exe' # ymmv!

    def visitmatch(self, fpathname, text):
        os.system('%s %s' % (self.editor, f pathname))

    if __name__ == '__main__':
        visitor = EditVisitor(sys.argv[1])
        visitor.run('.' if len(sys.argv) < 3 else sys.argv[2])
        print('Edited %d files, visited %d' % (visitor.scount, visitor.fcount))
```

When we make and run an `EditVisitor`, a text editor is started with the `os.system` command-line spawn call, which usually blocks its caller until the spawned program finishes. As coded, when run on my machines, each time this script finds a matched file during the traversal, it starts up the vi text editor within the console window where the script was started; exiting the editor resumes the tree walk.

Let's find and edit some files. When run as a script, we pass this program the search string as a command argument (here, the string `mimetypes` is the search key). The root directory passed to the `run` method is either the second argument or `.` (the current run directory) by default. Traversal status messages show up in the console, but each matched file now automatically pops up in a text editor along the way. In the following, the editor is started eight times—try this with an editor and tree of your own to get a better feel for how it works:

```
C:\\...\\PP4E\\Tools> visitor_edit.py mimetypes C:\\temp\\PP3E\\Examples
C:\\temp\\PP3E\\Examples ...
1 => C:\\temp\\PP3E\\Examples\\README-root.txt
C:\\temp\\PP3E\\Examples\\PP3E ...
2 => C:\\temp\\PP3E\\Examples\\PP3E\\echoEnvironment.pyw
3 => C:\\temp\\PP3E\\Examples\\PP3E\\LaunchBrowser.pyw
4 => C:\\temp\\PP3E\\Examples\\PP3E\\Launcher.py
5 => C:\\temp\\PP3E\\Examples\\PP3E\\Launcher.pyc
Skipping C:\\temp\\PP3E\\Examples\\PP3E\\Launcher.pyc
...more output omitted...
1427 => C:\\temp\\PP3E\\Examples\\PP3E\\TempParts\\lawnlake1-jan-03.jpg
```

```
Skipping C:\temp\PP3E\Examples\PP3E\TempParts\lawnlake1-jan-03.jpg
1428 => C:\temp\PP3E\Examples\PP3E\TempParts\part-001.txt
1429 => C:\temp\PP3E\Examples\PP3E\TempParts\part-002.html
Edited 8 files, visited 1429
```

This, finally, is the exact tool I was looking for to simplify global book examples tree maintenance. After major changes to things such as shared modules and file and directory names, I run this script on the examples root directory with an appropriate search string and edit any files it pops up as needed. I still need to change files by hand in the editor, but that's often safer than blind global replacements.

Global Replacements in Directory Trees (Visitor)

But since I brought it up: given a general tree traversal class, it's easy to code a global search-and-replace subclass, too. The `ReplaceVisitor` class in [Example 6-20](#) is a `SearchVisitor` subclass that customizes the `visitfile` method to globally replace any appearances of one string with another, in all text files at and below a root directory. It also collects the names of all files that were changed in a list just in case you wish to go through and verify the automatic edits applied (a text editor could be automatically popped up on each changed file, for instance).

Example 6-20. PP4E\Tools\visitor_replace.py

```
"""
Use: "python ...\\Tools\\visitor_replace.py rootdir fromStr toStr".
Does global search-and-replace in all files in a directory tree: replaces
fromStr with toStr in all text files; this is powerful but dangerous!!
visitor_edit.py runs an editor for you to verify and make changes, and so
is safer; use visitor_collect.py to simply collect matched files list;
listonly mode here is similar to both SearchVisitor and CollectVisitor;
"""

import sys
from visitor import SearchVisitor

class ReplaceVisitor(SearchVisitor):
    """
    Change fromStr to toStr in files at and below startDir;
    files changed available in obj.changed list after a run
    """
    def __init__(self, fromStr, toStr, listOnly=False, trace=0):
        self.changed = []
        self.toStr = toStr
        self.listOnly = listOnly
        SearchVisitor.__init__(self, fromStr, trace)

    def visitmatch(self, fname, text):
        self.changed.append(fname)
        if not self.listOnly:
            fromStr, toStr = self.context, self.toStr
            text = text.replace(fromStr, toStr)
            open(fname, 'w').write(text)
```

```

if __name__ == '__main__':
    listonly = input('List only?') == 'y'
    visitor = ReplaceVisitor(sys.argv[2], sys.argv[3], listonly)
    if listonly or input('Proceed with changes?') == 'y':
        visitor.run(startDir=sys.argv[1])
        action = 'Changed' if not listonly else 'Found'
        print('Visited %d files' % visitor.fcount)
        print(action, '%d files:' % len(visitor.changed))
        for fname in visitor.changed: print(fname)

```

To run this script over a directory tree, run the following sort of command line with appropriate “from” and “to” strings. On my shockingly underpowered netbook machine, doing this on a 1429-file tree and changing 101 files along the way takes roughly three seconds of real clock time when the system isn’t particularly busy.

```

C:\...\PP4E\Tools> visitor_replace.py C:\temp\PP3E\Examples PP3E PP4E
List only?y
Visited 1429 files
Found 101 files:
C:\temp\PP3E\Examples\README-root.txt
C:\temp\PP3E\Examples\PP3E\echoEnvironment.pyw
C:\temp\PP3E\Examples\PP3E\Launcher.py
...more matching filenames omitted...

```

```

C:\...\PP4E\Tools> visitor_replace.py C:\temp\PP3E\Examples PP3E PP4E
List only?n
Proceed with changes?y
Visited 1429 files
Changed 101 files:
C:\temp\PP3E\Examples\README-root.txt
C:\temp\PP3E\Examples\PP3E\echoEnvironment.pyw
C:\temp\PP3E\Examples\PP3E\Launcher.py
...more changed filenames omitted...

```

```

C:\...\PP4E\Tools> visitor_replace.py C:\temp\PP3E\Examples PP3E PP4E
List only?n
Proceed with changes?y
Visited 1429 files
Changed 0 files:

```

Naturally, we can also check our work by running the visitor script (and SearchVisitor superclass):

```

C:\...\PP4E\Tools> visitor.py 2 C:\temp\PP3E\Examples PP3E
Found in 0 files, visited 1429

C:\...\PP4E\Tools> visitor.py 2 C:\temp\PP3E\Examples PP4E
C:\temp\PP3E\Examples\README-root.txt has PP4E
C:\temp\PP3E\Examples\PP3E\echoEnvironment.pyw has PP4E
C:\temp\PP3E\Examples\PP3E\Launcher.py has PP4E
...more matching filenames omitted...
Found in 101 files, visited 1429

```

This is both wildly powerful and dangerous. If the string to be replaced can show up in places you didn't anticipate, you might just ruin an entire tree of files by running the `ReplaceVisitor` object defined here. On the other hand, if the string is something very specific, this object can obviate the need to manually edit suspicious files. For instance, website addresses in HTML files are likely too specific to show up in other places by chance.

Counting Source Code Lines (Visitor)

The two preceding `visitor` module clients were both search-oriented, but it's just as easy to extend the basic walker class for more specific goals. [Example 6-21](#), for instance, extends `FileVisitor` to count the number of lines in program source code files of various types throughout an entire tree. The effect is much like calling the `visitfile` method of this class for each filename returned by the `find` tool we wrote earlier in this chapter, but the OO structure here is arguably more flexible and extensible.

Example 6-21. PP4E\Tools\visitor_sloc.py

```
"""
Count lines among all program source files in a tree named on the command
line, and report totals grouped by file types (extension). A simple SLOC
(source lines of code) metric: skip blank and comment lines if desired.
"""

import sys, pprint, os
from visitor import FileVisitor

class LinesByType(FileVisitor):
    srcExts = [] # define in subclass

    def __init__(self, trace=1):
        FileVisitor.__init__(self, trace=trace)
        self.srcLines = self.srcFiles = 0
        self.extSums = {ext: dict(files=0, lines=0) for ext in self.srcExts}

    def visitsource(self, fpath, ext):
        if self.trace > 0: print(os.path.basename(fpath))
        lines = len(open(fpath, 'rb').readlines())
        self.srcFiles += 1
        self.srcLines += lines
        self.extSums[ext]['files'] += 1
        self.extSums[ext]['lines'] += lines

    def visitfile(self, filepath):
        FileVisitor.visitfile(self, filepath)
        for ext in self.srcExts:
            if filepath.endswith(ext):
                self.visitsource(filepath, ext)
                break

class PyLines(LinesByType):
    srcExts = ['.py', '.pyw'] # just python files
```

```

class SourceLines(LinesByType):
    srcExtSums = {'.py': 783, '.pyw': 11, '.c': 45, '.h': 7, '.i': 4, '.cgi': 5, '.html': 48, '.cxx': 4, '.htm': 278, '.js': 122, '.css': 297, '.xml': 49, '.json': 500}

    def __init__(self):
        self.fcount = 0
        self.dcount = 0
        self.srcFiles = 0
        self.srcLines = 0

    def run(self, path):
        for root, dirs, files in os.walk(path):
            self.fcount += len(files)
            self.dcount += len(dirs)
            for file in files:
                ext = os.path.splitext(file)[1]
                if ext in self.srcExtSums:
                    self.srcFiles += 1
                    self.srcLines += self.srcExtSums[ext]

    def __str__(self):
        return f'Visited {self.fcount} files and {self.dcount} dirs'

```

When run as a script, we get trace messages during the walk (omitted here to save space), and a report with line counts grouped by file type. Run this on trees of your own to watch its progress; my tree has 907 source files and 48K source lines, including 783 files and 34K lines of “.py” Python code:

```

C:\...\PP4E\Tools> visitor_sloc.py C:\temp\PP3E\Examples
Visited 1429 files and 186 dirs
-----
Source files=>907, lines=>48047
By Types:
{'.c': {'files': 45, 'lines': 7370},
 '.cgi': {'files': 5, 'lines': 122},
 '.cxx': {'files': 4, 'lines': 2278},
 '.h': {'files': 7, 'lines': 297},
 '.html': {'files': 48, 'lines': 2830},
 '.i': {'files': 4, 'lines': 49},
 '.py': {'files': 783, 'lines': 34601},
 '.pyw': {'files': 11, 'lines': 500}}
```

Check sums: 48047 907

Python only walk:
{'.py': {'files': 783, 'lines': 34601}, '.pyw': {'files': 11, 'lines': 500}}

Recoding Copies with Classes (Visitor)

Let’s peek at one more visitor use case. When I first wrote the `cpall.py` script earlier in this chapter, I couldn’t see a way that the `visitor` class hierarchy we met earlier would help. Two directories needed to be traversed in parallel (the original and the copy), and `visitor` is based on walking just one tree with `os.walk`. There seemed no easy way to keep track of where the script was in the copy directory.

The trick I eventually stumbled onto is not to keep track at all. Instead, the script in [Example 6-22](#) simply replaces the “from” directory path string with the “to” directory path string, at the front of all directory names and pathnames passed in from `os.walk`. The results of the string replacements are the paths to which the original files and directories are to be copied.

Example 6-22. PP4E\Tools\visitor_cpall.py

```
"""
Use: "python ...\\Tools\\visitor_cpall.py fromDir toDir trace?"
Like System\\Filetools\\cpall.py, but with the visitor classes and os.walk;
does string replacement of fromDir with toDir at the front of all the names
that the walker passes in; assumes that the toDir does not exist initially;
"""

import os
from visitor import FileVisitor           # visitor is in '.'
from PP4E.System.Filetools.cpall import copyfile    # PP4E is in a dir on path

class CpallVisitor(FileVisitor):
    def __init__(self, fromDir, toDir, trace=True):
        self.fromDirLen = len(fromDir) + 1
        self.toDir      = toDir
        FileVisitor.__init__(self, trace=trace)

    def visitdir(self, dirpath):
        toPath = os.path.join(self.toDir, dirpath[self.fromDirLen:])
        if self.trace: print('d', dirpath, '=>', toPath)
        os.mkdir(toPath)
        self.dcount += 1

    def visitfile(self, filepath):
        toPath = os.path.join(self.toDir, filepath[self.fromDirLen:])
        if self.trace: print('f', filepath, '=>', toPath)
        copyfile(filepath, toPath)
        self.fcount += 1

if __name__ == '__main__':
    import sys, time
    fromDir, toDir = sys.argv[1:3]
    trace = len(sys.argv) > 3
    print('Copying...')
    start = time.clock()
    walker = CpallVisitor(fromDir, toDir, trace)
    walker.run(startDir=fromDir)
    print('Copied', walker.fcount, 'files,', walker.dcount, 'directories', end=' ')
    print('in', time.clock() - start, 'seconds')
```

This version accomplishes roughly the same goal as the original, but it has made a few assumptions to keep the code simple. The “to” directory is assumed not to exist initially, and exceptions are not ignored along the way. Here it is copying the book examples tree from the prior edition again on Windows:

```
C:\...\PP4E\Tools> set PYTHONPATH
PYTHONPATH=C:\Users\Mark\Stuff\Books\4E\PP4E\dev\Examples

C:\...\PP4E\Tools> rmdir /S copytemp
copytemp, Are you sure (Y/N)? y

C:\...\PP4E\Tools> visitor_cpall.py C:\temp\PP3E\Examples copytemp
Copying...
Copied 1429 files, 186 directories in 11.1722033777 seconds

C:\...\PP4E\Tools> fc /B copytemp\PP3E\Launcher.py
                           C:\temp\PP3E\Examples\PP3E\Launcher.py
Comparing files COPYTEMP\PP3E\Launcher.py and C:\TEMP\PP3E\EXAMPLES\PP3E\LAUNCHER.PY
FC: no differences encountered
```

Despite the extra string slicing going on, this version seems to run just as fast as the original (the actual difference can be chalked up to system load variations). For tracing purposes, this version also prints all the “from” and “to” copy paths during the traversal if you pass in a third argument on the command line:

```
C:\...\PP4E\Tools> rmdir /S copytemp
copytemp, Are you sure (Y/N)? y

C:\...\PP4E\Tools> visitor_cpall.py C:\temp\PP3E\Examples copytemp 1
Copying...
d C:\temp\PP3E\Examples => copytemp\
f C:\temp\PP3E\Examples\README-root.txt => copytemp\README-root.txt
d C:\temp\PP3E\Examples\PP3E => copytemp\PP3E
...more lines omitted: try this on your own for the full output...
```

Other Visitor Examples (External)

Although the visitor is widely applicable, we don’t have space to explore additional subclasses in this book. For more example clients and use cases, see the following examples in book’s examples distribution package described in the [Preface](#):

- *Tools\visitor_collect.py* collects and/or prints files containing a search string
- *Tools\visitor_poundbang.py* replaces directory paths in “#!” lines at the top of Unix scripts
- *Tools\visitor_cleanpyc.py* is a visitor-based recoding of our earlier bytecode cleanup scripts
- *Tools\visitor_bigpy.py* is a visitor-based version of the “biggest file” example at the start of this chapter

Most of these are almost as trivial as the `visitor_edit.py` code in [Example 6-19](#), because the visitor framework handles walking details automatically. The collector, for instance, simply appends to a list as a search visitor detects matched files and allows the default list of text filename extensions in the search visitor to be overridden per instance—it's roughly like a combination of `find` and `grep` on Unix:

```
>>> from visitor.collect import CollectVisitor
>>> V = CollectVisitor('mimetypes', testtexts=['.py', '.pyw'], trace=0)
>>> V.run(r'C:\temp\PP3E\Examples')
>>> for name in V.matches: print(name)           # .py and .pyw files with 'mimetypes'
...
C:\temp\PP3E\Examples\PP3E\Internet\Email\mailtools\mailParser.py
C:\temp\PP3E\Examples\PP3E\Internet\Email\mailtools\mailSender.py
C:\temp\PP3E\Examples\PP3E\Internet\Ftp\mirror\downloadflat.py
C:\temp\PP3E\Examples\PP3E\Internet\Ftp\mirror\downloadflat_modular.py
C:\temp\PP3E\Examples\PP3E\Internet\Ftp\mirror\ftptools.py
C:\temp\PP3E\Examples\PP3E\Internet\Ftp\mirror\uploadflat.py
C:\temp\PP3E\Examples\PP3E\System\Media\playfile.py

C:\...\PP4E\Tools> visitor_collect.py mimetypes C:\temp\PP3E\Examples  # as script
```

The core logic of the biggest-file visitor is similarly straightforward, and harkens back to chapter start:

```
class BigPy(FileVisitor):
    def __init__(self, trace=0):
        FileVisitor.__init__(self, context=[], trace=trace)

    def visitfile(self, filepath):
        FileVisitor.visitfile(self, filepath)
        if filepath.endswith('.py'):
            self.context.append((os.path.getsize(filepath), filepath))
```

And the bytecode-removal visitor brings us back full circle, showing an additional alternative to those we met earlier in this chapter. It's essentially the same code, but it runs `os.remove` on ".pyc" file visits.

In the end, while the visitor classes are really just simple wrappers for `os.walk`, they further automate walking chores and provide a general framework and alternative class-based structure which may seem more natural to some than simple unstructured loops. They're also representative of how Python's OOP support maps well to real-world structures like file systems. Although `os.walk` works well for one-off scripts, the better extensibility, reduced redundancy, and greater encapsulation possible with OOP can be a major asset in real work as our needs change and evolve over time.



In fact, those needs *have* changed over time. Between the third and fourth editions of this book, the original `os.path.walk` call was removed in Python 3.X, and `os.walk` became the only automated way to perform tree walks in the standard library. Examples from the prior edition that used `os.path.walk` were effectively broken. By contrast, although the visitor classes used this call, too, its clients did not. Because updating the visitor classes to use `os.walk` internally did not alter those classes' interfaces, visitor-based tools continued to work unchanged.

This seems a prime example of the benefits of OOP's support for encapsulation. Although the future is never completely predictable, in practice, user-defined tools like visitor tend to give you more control over changes than standard library tools like `os.walk`. Trust me on that; as someone who has had to update three Python books over the last 15 years, I can say with some certainty that Python change is a constant!

Playing Media Files

We have space for just one last, quick example in this chapter, so we'll close with a bit of fun. Did you notice how the file extensions for text and binary file types were hard-coded in the directory search scripts of the prior two sections? That approach works for the trees they were applied to, but it's not necessarily complete or portable. It would be better if we could deduce file type from file name automatically. That's exactly what Python's `mimetypes` module can do for us. In this section, we'll use it to build a script that attempts to launch a file based upon its media type, and in the process develop general tools for opening media portably with specific or generic players.

As we've seen, on Windows this task is trivial—the `os.startfile` call opens files per the Windows registry, a system-wide mapping of file extension types to handler programs. On other platforms, we can either run specific media handlers per media type, or fall back on a resident web browser to open the file generically using Python's `webbrowser` module. [Example 6-23](#) puts these ideas into code.

Example 6-23. PP4E\System\Media\playfile.py

```
#!/usr/local/bin/python
#####
# Try to play an arbitrary media file. Allows for specific players instead of
# always using general web browser scheme. May not work on your system as is;
# audio files use filters and command lines on Unix, and filename associations
# on Windows via the start command (i.e., whatever you have on your machine to
# run .au files--an audio player, or perhaps a web browser). Configure and
# extend as needed. playknownfile assumes you know what sort of media you wish
# to open, and playfile tries to determine media type automatically using Python
# mimetypes module; both try to launch a web browser with Python webbrowser module
# as a last resort when mimetype or platform unknown.
#####
```

```

import os, sys, mimetypes, webbrowser

helpmsg = """
Sorry: can't find a media player for '%s' on your system!
Add an entry for your system to the media player dictionary
for this type of file in playfile.py, or play the file manually.
"""

def trace(*args): print(*args) # with spaces between

#####
# player techniques: generic and otherwise: extend me
#####

class MediaTool:
    def __init__(self, runtext=''):
        self.runtexxt = runtext
    def run(self, mediafile, **options):          # most ignore options
        fullpath = os.path.abspath(mediafile)       # cwd may be anything
        self.open(fullpath, **options)

class Filter(MediaTool):
    def open(self, mediafile, **ignored):
        media = open(mediafile, 'rb')
        player = os.popen(self.runtexxt, 'w')      # spawn shell tool
        player.write(media.read())                 # send to its stdin

class Cmdline(MediaTool):
    def open(self, mediafile, **ignored):
        cmdline = self.runtexxt % mediafile         # run any cmd line
        os.system(cmdline)                         # use %s for filename

class Winstart(MediaTool):                      # use Windows registry
    def open(self, mediafile, wait=False, **other): # or os.system('start file')
        if not wait:                            # allow wait for curr media
            os.startfile(mediafile)
        else:
            os.system('start /WAIT ' + mediafile)

class Webbrowser(MediaTool):
    # file:// requires abs path
    def open(self, mediafile, **options):
        webbrowser.open_new('file://%s' % mediafile, **options)

#####
# media- and platform-specific policies: change me, or pass one in
#####

# map platform to player: change me!

audiotools = {
    'sunos5': Filter('/usr/bin/audioplay'),           # os.popen().write()
    'linux2': Cmdline('cat %s > /dev/audio'),        # on zaurus, at least
    'sunos4': Filter('/usr/demo/SOUND/play'),          # yes, this is that old!
}

```

```

'win32':  Winstart()                                # startfile or system
#'win32':  Cmdline('start %s')
}

videotools = {
    'linux2':  Cmdline('tkcVideo_c700 %s'),           # zaurus pda
    'win32':   Winstart(),                            # avoid DOS pop up
}

imagetools = {
    'linux2':  Cmdline('zimager %s'),                 # zaurus pda
    'win32':   Winstart(),
}

texttools = {
    'linux2':  Cmdline('vi %s'),                      # zaurus pda
    'win32':   Cmdline('notepad %s')                  # or try PyEdit?
}

apptools = {
    'win32':   Winstart()    # doc, xls, etc: use at your own risk!
}

# map mimetype of filenames to player tables

mimetype = {'audio':      audiotools,
            'video':      videotools,
            'image':      imagetools,
            'text':       texttools,          # not html text: browser
            'application': apptools}

#####
# top-level interfaces
#####

def trywebbrowser(filename, helpmsg=helpmsg, **options):
    """
    try to open a file in a web browser
    last resort if unknown mimetype or platform, and for text/html
    """
    trace('trying browser', filename)
    try:
        player = Webbrowser()                      # open in local browser
        player.run(filename, **options)
    except:
        print(helpmsg % filename)                # else nothing worked

def playknownfile(filename, playertable={}, **options):
    """
    play media file of known type: uses platform-specific
    player objects, or spawns a web browser if nothing for
    this platform; accepts a media-specific player table
    """
    if sys.platform in playertable:
        playertable[sys.platform].run(filename, **options)  # specific tool

```

```

else:
    trywebbrowser(filename, **options)                      # general scheme

def playfile(filename, mimetable=mimetable, **options):
    """
    play media file of any type: uses mimetypes to guess media
    type and map to platform-specific player tables; spawn web
    browser if text/html, media type unknown, or has no table
    """
    contenttype, encoding = mimetypes.guess_type(filename)
    if contenttype == None or encoding is not None:
        contenttype = '?/?'
    maintype, subtype = contenttype.split('/', 1)
    if maintype == 'text' and subtype == 'html':
        trywebbrowser(filename, **options)                  # special case
    elif maintype in mimetable:
        playknownfile(filename, mimetable[maintype], **options) # try table
    else:
        trywebbrowser(filename, **options)                  # other types

#####
# self-test code
#####

if __name__ == '__main__':
    # media type known
    playknownfile('sousa.au', audiotools, wait=True)
    playknownfile('ora-pp3e.gif', imagetools, wait=True)
    playknownfile('ora-lp4e.jpg', imagetools)

    # media type guessed
    input('Stop players and press Enter')
    playfile('ora-lp4e.jpg')                            # image/jpeg
    playfile('ora-pp3e.gif')                            # image/gif
    playfile('priorcalendar.html')                     # text/html
    playfile('lp4e-preface-preview.html')               # text/html
    playfile('lp-code-readme.txt')                      # text/plain
    playfile('spam.doc')                             # app
    playfile('spreadsheet.xls')                       # app
    playfile('sousa.au', wait=True)                   # audio/basic
    input('Done')                                     # stay open if clicked

```

Although it's generally possible to open most media files by passing their names to a web browser these days, this module provides a simple framework for launching media files with more specific tools, tailored by both media type and platform. A web browser is used only as a fallback option, if more specific tools are not available. The net result is an extendable media file player, which is as specific and portable as the customizations you provide for its tables.

We've seen the program launch tools employed by this script in prior chapters. The script's main new concepts have to do with the modules it uses: the `webbrowser` module to open some files in a local web browser, as well as the Python `mimetypes` module to

determine media type from file name. Since these are the heart of this code’s matter, let’s explore these briefly before we run the script.

The Python `webbrowser` Module

The standard library `webbrowser` module used by this example provides a portable interface for launching web browsers from Python scripts. It attempts to locate a suitable web browser on your local machine to open a given URL (file or web address) for display. Its interface is straightforward:

```
>>> import webbrowser  
>>> webbrowser.open_new('file://' + fullfilename)           # use os.path.abspath()
```

This code will open the named file in a new web browser window using whatever browser is found on the underlying computer, or raise an exception if it cannot. You can tailor the browsers used on your platform, and the order in which they are attempted, by using the `BROWSER` environment variable and `register` function. By default, `webbrowser` attempts to be automatically portable across platforms.

Use an argument string of the form “file://...” or “http://...” to open a file on the local computer or web server, respectively. In fact, you can pass in any URL that the browser understands. The following pops up Python’s home page in a new locally-running browser window, for example:

```
>>> webbrowser.open_new('http://www.python.org')
```

Among other things, this is an easy way to display HTML documents as well as media files, as demonstrated by this section’s example. For broader applicability, this module can be used as both command-line script (Python’s `-m` module search path flag helps here) and as importable tool:

```
C:\Users\mark\Stuff\Websites\public_html> python -m webbrowser about-pp.html  
C:\Users\mark\Stuff\Websites\public_html> python -m webbrowser -n about-pp.html  
C:\Users\mark\Stuff\Websites\public_html> python -m webbrowser -t about-pp.html  
  
C:\Users\mark\Stuff\Websites\public_html> python  
>>> import webbrowser  
>>> webbrowser.open('about-pp.html')                  # reuse, new window, new tab  
True  
>>> webbrowser.open_new('about-pp.html')            # file:// optional on Windows  
True  
>>> webbrowser.open_new_tab('about-pp.html')  
True
```

In both modes, the difference between the three usage forms is that the first tries to reuse an already-open browser window if possible, the second tries to open a new window, and the third tries to open a new tab. In practice, though, their behavior is totally dependent on what the browser selected on your platform supports, and even on the platform in general. All three forms may behave the same.

On Windows, for example, all three simply run `os.startfile` by default and thus create a new tab in an existing window under Internet Explorer 8. This is also why I didn't need the "file://" full URL prefix in the preceding listing. Technically, Internet Explorer is only run if this is what is registered on your computer for the file type being opened; if not, that file type's handler is opened instead. Some images, for example, may open in a photo viewer instead. On other platforms, such as Unix and Mac OS X, browser behavior differs, and non-URL file names might not be opened; use "file://" for portability.

We'll use this module again later in this book. For example, the PyMailGUI program in [Chapter 14](#) will employ it as a way to display HTML-formatted email messages and attachments, as well as program help. See the Python library manual for more details. In Chapters 13 and 15, we'll also meet a related call, `urllib.request.urlopen`, which fetches a web page's text given a URL, but does not open it in a browser; it may be parsed, saved, or otherwise used.

The Python `mimetypes` Module

To make this media player module even more useful, we also use the Python `mimetypes` standard library module to automatically determine the media type from the filename. We get back a `type/subtype` MIME content-type string if the type can be determined or `None` if the guess failed:

```
>>> import mimetypes
>>> mimetypes.guess_type('spam.jpg')
('image/jpeg', None)

>>> mimetypes.guess_type('TheBrightSideOfLife.mp3')
('audio/mpeg', None)

>>> mimetypes.guess_type('lifeofbrian.mpg')
('video/mpeg', None)

>>> mimetypes.guess_type('lifeofbrian.xyz')      # unknown type
(None, None)
```

Stripping off the first part of the content-type string gives the file's general media type, which we can use to select a generic player; the second part (subtype) can tell us if text is plain or HTML:

```
>>> ctype, encoding = mimetypes.guess_type('spam.jpg')
>>> ctype.split('/')[0]
'image'

>>> mimetypes.guess_type('spam.txt')            # subtype is 'plain'
('text/plain', None)

>>> mimetypes.guess_type('spam.html')
('text/html', None)
```

```
>>> mimetypes.guess_type('spam.html')[0].split('/')[1]
'html'
```

A subtle thing: the second item in the tuple returned from the `mimetypes` guess is an encoding type we won't use here for opening purposes. We still have to pay attention to it, though—if it is not `None`, it means the file is compressed (`gzip` or `compress`), even if we receive a media content type. For example, if the filename is something like `spam.gif.gz`, it's a compressed image that we don't want to try to open directly:

```
>>> mimetypes.guess_type('spam.gz')                      # content unknown
(None, 'gzip')

>>> mimetypes.guess_type('spam.gif.gz')                 # don't play me!
('image/gif', 'gzip')

>>> mimetypes.guess_type('spam.zip')                   # archives
('application/zip', None)

>>> mimetypes.guess_type('spam.doc')                  # office app files
('application/msword', None)
```

If the filename you pass in contains a directory path, the path portion is ignored (only the extension is used). This module is even smart enough to give us a filename extension for a type—useful if we need to go the other way, and create a file name from a content type:

```
>>> mimetypes.guess_type(r'C:\songs\sousa.au')
('audio/basic', None)

>>> mimetypes.guess_extension('audio/basic')
'.au'
```

Try more calls on your own for more details. We'll use the `mimetypes` module again in FTP examples in [Chapter 13](#) to determine transfer type (text or binary), and in our email examples in Chapters [13](#), [14](#), and [16](#) to send, save, and open mail attachments.

In [Example 6-23](#), we use `mimetypes` to select a table of platform-specific player commands for the media type of the file to be played. That is, we pick a player table for the file's media type, and then pick a command from the player table for the platform. At both steps, we give up and run a web browser if there is nothing more specific to be done.

Using `mimetypes` guesses for `SearchVisitor`

To use this module for directing our text file search scripts we wrote earlier in this chapter, simply extract the first item in the content-type returned for a file's name. For instance, all in the following list are considered text (except “.pyw”, which we may have to special-case if we must care):

```
>>> for ext in ['.txt', '.py', '.pyw', '.html', '.c', '.h', '.xml']:
...     print(ext, mimetypes.guess_type('spam' + ext))
...
```

```

.txt ('text/plain', None)
.py ('text/x-python', None)
.pyw (None, None)
.html ('text/html', None)
.c ('text/plain', None)
.h ('text/plain', None)
.xml ('text/xml', None)

```

We can add this technique to our earlier `SearchVisitor` class by redefining its candidate selection method, in order to replace its default extension lists with `mimetypes` guesses—yet more evidence of the power of OOP customization at work:

```

C:\...\PP4E\Tools> python
>>> import mimetypes
>>> from visitor import SearchVisitor           # or PP4E.Tools.visitor if not .
>>>
>>> class SearchMimeVisitor(SearchVisitor):
...     def candidate(self, fname):
...         contype, encoding = mimetypes.guess_type(fname)
...         return (contype and
...                 contype.split('/')[-1] == 'text' and
...                 encoding == None)
...
>>> V = SearchMimeVisitor('mimetypes', trace=0)      # search key
>>> V.run(r'C:\temp\PP3E\Examples')                  # root dir
C:\temp\PP3E\Examples\PP3E\extras\LosAlamosAdvancedClass\day1-system\data.txt has mimetypes
C:\temp\PP3E\Examples\PP3E\Internet\Email\mailtools\mailParser.py has mimetypes
C:\temp\PP3E\Examples\PP3E\Internet\Email\mailtools\mailSender.py has mimetypes
C:\temp\PP3E\Examples\PP3E\Internet\Ftp\mirror\downloadflat.py has mimetypes
C:\temp\PP3E\Examples\PP3E\Internet\Ftp\mirror\downloadflat_modular.py has mimetypes
C:\temp\PP3E\Examples\PP3E\Internet\Ftp\mirror\ftptools.py has mimetypes
C:\temp\PP3E\Examples\PP3E\Internet\Ftp\mirror\uploadflat.py has mimetypes
C:\temp\PP3E\Examples\PP3E\System\Media\playfile.py has mimetypes
>>> V.scount, V.fcount, V.dcount
(8, 1429, 186)

```

Because this is not completely accurate, though (you may need to add logic to include extensions like “.pyw” missed by the guess), and because it’s not even appropriate for all search clients (some may want to search specific kinds of text only), this scheme was not used for the original class. Using and tailoring it for your own searches is left as optional exercise.

Running the Script

Now, when [Example 6-23](#) is run from the command line, if all goes well its canned self-test code at the end opens a number of audio, image, text, and other file types located in the script’s directory, using either platform-specific players or a general web browser. On my Windows 7 laptop, GIF and HTML files open in new IE browser tabs; JPEG files in Windows Photo Viewer; plain text files in Notepad; DOC and XLS files in Microsoft Word and Excel; and audio files in Windows Media Player.

Because the programs used and their behavior may vary widely from machine to machine, though, you’re best off studying this script’s code and running it on your own computer and with your own test files to see what happens. As usual, you can also test it interactively (use the package path like this one to import from a different directory, assuming your module search path includes the PP4E root):

```
>>> from PP4E.System.Media.playfile import playfile  
>>> playfile(r'C:\movies\mov10428.mpg') # video/mpeg
```

We’ll use the `playfile` module again as an imported library like this in [Chapter 13](#) to open media files downloaded by FTP. Again, you may want to tweak this script’s tables for your players. This script also assumes the media file is located on the local machine (even though the `webbrowser` module supports remote files with “`http://`” names), and it does not currently allow different players for most different MIME subtypes (it special-cases text to handle “plain” and “html” differently, but no others). In fact, this script is really just something of a simple framework that was designed to be extended. As always, hack on; this is Python, after all.

Automated Program Launchers (External)

Finally, some optional reading—in the examples distribution package for this book (available at sites listed in the [Preface](#)) you can find additional system-related scripts we do not have space to cover here:

- `PP4ELauncher.py`—contains tools used by some GUI programs later in the book to start Python programs without any environment configuration. Roughly, it sets up both the system path and module import search paths as needed to run book examples, which are inherited by spawned programs. By using this module to search for files and configure environments automatically, users can avoid (or at least postpone) having to learn the intricacies of manual environment configuration before running programs. Though there is not much new in this example from a system interfaces perspective, we’ll refer back to it later, when we explore GUI programs that use its tools, as well as those of its `launchmodes` cousin, which we wrote in [Chapter 5](#).
- `PP4ELaunch_PyDemos.pyw` and `PP4ELaunch_PyGadgets_bar.pyw`—use `Launcher.py` to start major GUI book examples without any environment configuration. Because all spawned processes inherit configurations performed by the launcher, they all run with proper search path settings. When run directly, the underlying `PyDemos2.pyw` and `PyGadgets_bar.pyw` scripts (which we’ll explore briefly at the end of [Chapter 10](#)) instead rely on the configuration settings on the underlying machine. In other words, `Launcher` effectively hides configuration details from the GUI interfaces by enclosing them in a configuration program layer.
- `PP4ELaunchBrowser.pyw`—portably locates and starts an Internet web browser program on the host machine in order to view a local file or remote web page. In

prior versions, it used tools in *Launcher.py* to search for a reasonable browser to run. The original version of this example has now been largely superseded by the standard library’s `webbrowser` module, which arose after this example had been developed (reptilian minds think alike!). In this edition, `LaunchBrowser` simply parses command-line arguments for backward compatibility and invokes the `open` function in `webbrowser`. See this module’s help text, or PyGadgets and PyDemos in [Chapter 10](#), for example command-line usage.

That’s the end of our system tools exploration. In the next part of this book we leave the realm of the system shell and move on to explore ways to add graphical user interfaces to our program. Later, we’ll do the same using web-based approaches. As we continue, keep in mind that the system tools we’ve studied in this part of the book see action in a wide variety of programs. For instance, we’ll put threads to work to spawn long-running tasks in the GUI part, use both threads and processes when exploring server implementations in the Internet part, and use files and file-related system calls throughout the remainder of the book.

Whether your interfaces are command lines, multiwindow GUIs, or distributed client/server websites, Python’s system interfaces toolbox is sure to play an important part in your Python programming future.

PART III

GUI Programming

This part of the book shows you how to apply Python to build portable graphical user interfaces, primarily with Python’s standard `tkinter` library. The following chapters cover this topic in depth:

Chapter 7

This chapter outlines GUI options available to Python developers, and then presents a brief tutorial that illustrates core `tkinter` coding concepts.

Chapter 8

This chapter begins a two-part tour of the `tkinter` library—its widget set and related tools. This first tour chapter covers simpler library tools and widgets: pop-up windows, various types of buttons, images, and so on.

Chapter 9

This chapter continues the library tour begun in the prior chapter. It presents the rest of the `tkinter` widget library, including menus, text, canvases, scroll bars, grids, and time-based events and animation.

Chapter 10

This chapter takes a look at GUI programming techniques: we’ll learn how to build menus automatically from object templates, spawn GUIs as separate programs, run long-running tasks in parallel with threads and queues, and more.

Chapter 11

This chapter pulls the earlier chapters’ ideas together to implement a collection of user interfaces. It presents a number of larger GUIs—clocks, text editors, drawing programs, image viewers, and so on—which also demonstrate general Python programming-in-the-large concepts along the way.

As in the first part of this book, the material presented here is applicable to a wide variety of domains and will be utilized again to build domain-specific user interfaces in later chapters of this book. For instance, the `PyMailGUI` and `PyCalc` examples of later chapters will assume that you’ve covered the basics here.

Graphical User Interfaces

“Here’s Looking at You, Kid”

For most software systems, a graphical user interface (GUI) has become an expected part of the package. Even if the GUI acronym is new to you, chances are that you are already familiar with such interfaces—the windows, buttons, and menus that we use to interact with software programs. In fact, most of what we do on computers today is done with some sort of point-and-click graphical interface. From web browsers to system tools, programs are routinely dressed up with a GUI component to make them more flexible and easier to use.

In this part of the book, we will learn how to make Python scripts sprout such graphical interfaces, too, by studying examples of programming with the *tkinter* module, a portable GUI library that is a standard part of the Python system and the toolkit most widely used by Python programmers. As we’ll see, it’s easy to program user interfaces in Python scripts thanks to both the simplicity of the language and the power of its GUI libraries. As an added bonus, GUIs programmed in Python with *tkinter* are automatically portable to all major computer systems.

GUI Programming Topics

Because GUIs are a major area, I want to say a few more words about this part of the book before we get started. To make them easier to absorb, GUI programming topics are split over the next five chapters:

- This chapter begins with a quick *tkinter* tutorial to teach coding basics. Interfaces are kept simple here on purpose, so you can master the fundamentals before moving on to the following chapter’s interfaces. On the other hand, this chapter covers all the basics: event processing, the *pack* geometry manager, using inheritance and composition in GUIs, and more. As we’ll see, object-oriented programming (OOP) isn’t required for *tkinter*, but it makes GUIs structured and reusable.

- Chapters 8 and 9 take you on a tour of the tkinter widget set.* Roughly, Chapter 8 presents simple widgets and Chapter 9 covers more advanced widgets and related tools. Most of the interface devices you’re accustomed to seeing—sliders, menus, dialogs, images, and their kin—show up here. These two chapters are not a fully complete tkinter reference (which could easily fill a large book by itself), but they should be enough to help you get started coding substantial Python GUIs. The examples in these chapters are focused on widgets and tkinter tools, but Python’s support for code reuse is also explored along the way.
- Chapter 10 covers more advanced GUI programming techniques. It includes an exploration of techniques for automating common GUI tasks with Python. Although tkinter is a full-featured library, a small amount of reusable Python code can make its interfaces even more powerful and easier to use.
- Chapter 11 wraps up by presenting a handful of complete GUI programs that make use of coding and widget techniques presented in the four preceding chapters. We’ll learn how to implement text editors, image viewers, clocks, and more.

Because GUIs are actually cross-domain tools, other GUI examples will also show up throughout the remainder of this book. For example, we’ll later see complete email GUIs and calculators, as well as a basic FTP client GUI; additional examples such as tree viewers and table browsers are available externally in the book examples package. Chapter 11 gives a list of forward pointers to other tkinter examples in this text.

After we explore GUIs, in Part IV we’ll also learn how to build basic user interfaces within a web browser using HTML and Python scripts that run on a server—a very different model with advantages and tradeoffs all its own that are important to understand. Newer technologies such as the RIAs described later in this chapter build on the web browser model to offer even more interface choices.

For now, though, our focus here is on more traditional GUIs—known as “desktop” applications to some, and as “standalone” GUIs to others. As we’ll see when we meet FTP and email client GUIs in the Internet part of this book, though, such programs often connect to a network to do their work as well.

* The term “widget set” refers to the objects used to build familiar point-and-click user interface devices—push buttons, sliders, input fields, and so on. tkinter comes with Python classes that correspond to all the widgets you’re accustomed to seeing in graphical displays. Besides widgets, tkinter also comes with tools for other activities, such as scheduling events to occur, waiting for socket data to arrive, and so on.

Running the Examples

One other point I'd like to make right away: most GUIs are dynamic and interactive interfaces, and the best I can do here is show static screenshots representing selected states in the interactions such programs implement. This really won't do justice to most examples. If you are not working along with the examples already, I encourage you to run the GUI examples in this and later chapters on your own.

On Windows, the standard Python install comes with tkinter support built in, so all these examples should work immediately. Mac OS X comes bundled with a tkinter-aware Python as well. For other systems, Pythons with tkinter support are either provided with the system itself or are readily available (see the top-level *README-PP4E.txt* file in the book examples distribution for more details). Getting tkinter to work on your computer is worth whatever extra install details you may need to absorb, though; experimenting with these programs is a great way to learn about both GUI programming and Python itself.

Also see the description of book example portability in general in this book's Preface. Although Python and tkinter are both largely platform neutral, you may run into some minor platform-specific issues if you try to run this book's examples on platforms other than that used to develop this book. Mac OS X, for example, might pose subtle differences in some of the examples' operation. Be sure to watch this book's website for pointers and possible future patches for using the examples on other platforms.

Has Anyone Noticed That G-U-I Are the First Three Letters of "GUIDO"?

Python creator Guido van Rossum didn't originally set out to build a GUI development tool, but Python's ease of use and rapid turnaround have made this one of its primary roles. From an implementation perspective, GUIs in Python are really just instances of C extensions, and extensibility was one of the main ideas behind Python. When a script builds push buttons and menus, it ultimately talks to a C library; and when a script responds to a user event, a C library ultimately talks back to Python. It's really just an example of what is possible when Python is used to script external libraries.

But from a practical point of view, GUIs are a critical part of modern systems and an ideal domain for a tool like Python. As we'll see, Python's simple syntax and object-oriented flavor blend well with the GUI model—it's natural to represent each device drawn on a screen as a Python class. Moreover, Python's quick turnaround lets programmers experiment with alternative layouts and behavior rapidly, in ways not possible with traditional development techniques. In fact, you can usually make a change to a Python-based GUI and observe its effects in a matter of seconds. Don't try this with C++!

Python GUI Development Options

Before we start wading into the tkinter pond, let's begin with some perspective on Python GUI options in general. Because Python has proven to be such a good match for GUI work, this domain has seen much activity over the years. In fact, although tkinter is by most accounts still the most widely used GUI toolkit in the Python world, there are a variety of ways to program user interfaces in Python today. Some are specific to Windows or X Windows,[†] some are cross-platform solutions, and all have followings and strong points of their own. To be fair to all the alternatives, here is a brief inventory of GUI toolkits available to Python programmers as I write these words:

tkinter

An open source GUI library and the continuing de facto standard for portable GUI development in Python. Python scripts that use tkinter to build GUIs run portably on Windows, X Windows (Unix and Linux), and Macintosh OS X, and they display a native look-and-feel on each of these platforms today. tkinter makes it easy to build simple and portable GUIs quickly. Moreover, it can be easily augmented with Python code, as well as with larger extension packages such as *Pmw* (a third-party widget library); *Tix* (another widget library, and now a standard part of Python); *PIL* (an image-processing extension); and *ttk* (Tk themed widgets, also now a standard part of Python as of version 3.1). More on such extensions like these later in this introduction.

The underlying Tk library used by tkinter is a standard in the open source world at large and is also used by the Perl, Ruby, PHP, Common Lisp, and Tcl scripting languages, giving it a user base that likely numbers in the millions. The Python binding to Tk is enhanced by Python's simple object model—Tk widgets become customizable and embeddable objects, instead of string commands. tkinter takes the form of a module package in Python 3.X, with nested modules that group some of its tools by functionality (it was formerly known as module *Tkinter* in Python 2.X, but was renamed to follow naming conventions, and restructured to provide a more hierarchical organization).

tkinter is mature, robust, widely used, and well documented. It includes roughly 25 basic widget types, plus various dialogs and other tools. Moreover, there is a dedicated book on the subject, plus a large library of published tkinter and Tk documentation. Perhaps most importantly, because it is based on a library

[†] In this book, “Windows” refers to the Microsoft Windows interface common on PCs, and “X Windows” refers to the X11 interface most commonly found on Unix and Linux platforms. These two interfaces are generally tied to the Microsoft and Unix (and Unix-like) platforms, respectively. It’s possible to run X Windows on top of a Microsoft operating system and Windows emulators on Unix and Linux, but it’s not common. As if to muddy the waters further, Mac OS X supports Python’s tkinter on both X Windows and the native Aqua GUI system directly, in addition to platform-specific cocoa options (though it’s usually not too misleading to lump OS X in with the “Unix-like” crowd).

developed for scripting languages, tkinter is also a relatively lightweight toolkit, and as such it meshes well with a scripting language like Python.

Because of such attributes, Python's tkinter module ships with Python as a standard library module and is the basis of Python's standard IDLE integrated development environment GUI. In fact, tkinter is the only GUI toolkit that is part of Python; all others on this list are third-party extensions. The underlying Tk library is also shipped with Python on some platforms (including Windows, Mac OS X, and most Linux and Unix-like systems). You can be reasonably sure that tkinter will be present when your script runs, and you can guarantee this if needed by freezing your GUI into a self-contained binary executable with tools like PyInstaller and py2exe (see the Web for details).

Although tkinter is easy to use, its text and canvas widgets are powerful enough to implement web pages, three-dimensional visualization, and animation. In addition, a variety of systems aim to provide GUI builders for Python/tkinter today, including GUI Builder (formerly part of the Komodo IDE and relative of SpecTCL), Rapyd-Tk, xRope, and others (though this set has historically tended to change much over time; see <http://wiki.python.org/moin/GuiProgramming> or search the Web for updates). As we will see, though, tkinter is usually so easy to code that GUI builders are not widely used. This is especially true once we leave the realm of the static layouts that builders typically support.

wxPython

A Python interface for the open source wxWidgets (formerly called wxWindows) library, which is a portable GUI class framework originally written to be used from the C++ programming language. The wxPython system is an extension module that wraps wxWidgets classes. This library is generally considered to excel at building sophisticated interfaces and is probably the second most popular Python GUI toolkit today, behind tkinter. GUIs coded in Python with wxPython are portable to Windows, Unix-like platforms, and Mac OS X.

Because wxPython is based on a C++ class library, most observers consider it to be more complex than tkinter: it provides hundreds of classes, generally requires an object-oriented coding style, and has a design that some find reminiscent of the MFC class library on Windows. wxPython often expects programmers to write more code, partly because it is a more functional and thus complex system, and partly because it inherits this mindset from its underlying C++ library.

Moreover, some of wxPython's documentation is oriented toward C++, though this story has been improved recently with the publication of a book dedicated to wxPython. By contrast, tkinter is covered by one book dedicated to it, large sections of other Python books, and an even larger library of existing literature on the underlying Tk toolkit. Since the world of Python books has been remarkably dynamic over the years, though, you should investigate the accuracy of these observations at the time that you read these words; some books fade, while new Python books appear on a regular basis.

On the other hand, in exchange for its added complexity, wxPython provides a powerful toolkit. wxPython comes with a richer set of widgets out of the box than tkinter, including trees and HTML viewers—things that may require extensions such as Pmw, Tix, or ttk in tkinter. In addition, some prefer the appearance of the interfaces it renders. BoaConstructor and wxDesigner, among other options, provide a GUI builder that generates wxPython code. Some wxWidgets tools also support non-GUI Python work as well. For a quick look at wxPython widgets and code, run the demo that comes with the system (see <http://wxpython.org/>, or search the Web for links).

PyQt

A Python interface to the Qt toolkit (now from Nokia, formerly by Trolltech), and perhaps the third most widely used GUI toolkit for Python today. PyQt is a full-featured GUI library and runs portably today on Windows, Mac OS X, and Unix and Linux. Like wxPython, Qt is generally more complex, yet more feature rich, than tkinter; it contains hundreds of classes and thousands of functions and methods. Qt grew up on Linux but became portable to other systems over time; reflecting this heritage, the PyQt and PyKDE extension packages provide access to KDE development libraries (PyKDE requires PyQt). The BlackAdder and Qt Designer systems provide GUI builders for PyQt.

Perhaps Qt's most widely cited drawback in the past has been that it was not completely open source for full commercial use. Today, Qt provides both GPL and LGPL open source licensing, as well as commercial license options. The LGPL and GPL versions are open source, but conform to GPL licensing constraints (GPL may also impose requirements beyond those of the Python BSD-style license; you must, for example, make your source code freely available to end users).

PyGTK

A Python interface to GTK, a portable GUI library originally used as the core of the Gnome window system on Linux. The gnome-python and PyGTK extension packages export Gnome and GTK toolkit calls. At this writing, PyGTK runs portably on Windows and POSIX systems such as Linux and Mac OS X (according to its documentation, it currently requires that an X server for Mac OS X has been installed, though a native Mac version is in the works).

Jython

Jython (the system formerly known as JPython) is a Python implementation for Java, which compiles Python source code to Java bytecode, and gives Python scripts seamless access to Java class libraries on the local machine. Because of that, Java GUI libraries such as `swing` and `awt` become another way to construct GUIs in Python code run by the Jython system. Such solutions are obviously Java specific and limited in portability to that of Java and its libraries. Furthermore, `swing` may be one of the largest and most complex GUI option for Python work. A new package named `jTkinter` also provides a tkinter port to Jython using Java's JNI; if

installed, Python scripts may also use tkinter to build GUIs under Jython. Jython also has Internet roles we'll meet briefly in [Chapter 12](#).

IronPython

In a very similar vein, the IronPython system—an implementation of the Python language for the .NET environment and runtime engine, which, among other things, compiles Python programs to .NET bytecode—also offers Python scripts GUI construction options in the .NET framework. You write Python code, but use C#/.NET components to construct interfaces, and applications at large. IronPython code can be run on .NET on Windows, but also on Linux under the Mono implementation of .NET, and in the Silverlight client-side RIA framework for web browsers (discussed ahead).

PythonCard

An open source GUI builder and library built on top of the wxPython toolkit and considered by some to be one of Python's closest equivalents to the kind of GUI builders familiar to Visual Basic developers. PythonCard describes itself as a GUI construction kit for building cross-platform desktop applications on Windows, Mac OS X, and Linux, using the Python language.

Dabo

An open source GUI builder also built on wxPython, and a bit more. Dabo is a portable, three-tier, cross-platform desktop application development framework, inspired by Visual FoxPro and written in Python. Its tiers support database access, business logic, and user interface. Its open design is intended to eventually support a variety of databases and multiple user interfaces (wxPython, tkinter, and even HTML over HTTP).

Rich Internet Applications (RIAs)

Although web pages rendered with HTML are also a kind of user interface, they have historically been too limited to include in the general GUI category. However, some observers would extend this category today to include systems which allow browser-based interfaces to be much more dynamic than traditional web pages have allowed. Because such systems provide widget toolkits rendered by web browsers, they can offer some of the same portability benefits as web pages in general.

The going buzzword for this brave new breed of toolkits is *rich Internet applications* (RIAs). It includes AJAX and JavaScript-oriented frameworks for use on the client, such as:

Flex

An open source framework from Adobe and part of the Flash platform

Silverlight

A Microsoft framework which is also usable on Linux with Mono's Moonlight, and can be accessed by Python code with the IronPython system described above

JavaFX

A Java platform for building RIAs which can run across a variety of connected devices

pyjamas

An AJAX-based port of the Google Web Toolkit to Python, which comes with a set of interface widgets and compiles Python code that uses those widgets into JavaScript, to be run in a browser on a client

The *HTML5* standard under development proposes to address this domain as well. Web browsers ultimately are “desktop” GUI applications, too, but are more pervasive than GUI libraries, and can be generalized with RIA tools to render other GUIs. While it’s possible to build a widget-based GUI with such frameworks, they can also add overheads associated with networking in general and often imply a substantially heavier software stack than traditional GUI toolkits. Indeed, in order to morph browsers into general GUI platforms, RIAs may imply extra software layers and dependencies, and even multiple programming languages. Because of that, and because not everyone codes for the Web today (despite what you may have heard), we won’t include them in our look at traditional standalone/desktop GUIs in this part of the book.

See the Internet part for more on RIAs and user interfaces based on browsers, and be sure to watch for news and trends on this front over time. The interactivity these tools provide is also a key part of what some refer to as “Web 2.0” when viewed more from the perspective of the Web than GUIs. Since we’re concerned with the latter here (and since user interaction is user interaction regardless of what jargon we use for it), we’ll postpone further enumeration of this topic until the next part of the book.

Platform-specific options

Besides the portable toolkits like `tkinter`, `wxPython`, and `PyQt`, and platform-agnostic approaches such as RIAs, most major platforms have nonportable options for Python-coded GUIs as well. For instance, on Macintosh OS X, `PyObjC` provides a Python binding to Apple’s Objective-C/Cocoa framework, which is the basis for much Mac development. On Windows, the `PyWin32` extensions package for Python includes wrappers for the C++ Microsoft Foundation Classes (MFC) framework (a library that includes interface components), as well as `Pythonwin`, an MFC sample program that implements a Python development GUI. Although .NET technically runs on Linux, too, the IronPython system mentioned earlier offers additional Windows-focused options.

See the websites of these toolkits for more details. There are other lesser-known GUI toolkits for Python, and new ones are likely to emerge by the time you read this book (in fact, IronPython was new in the third edition, and RIAs are new in the fourth). Moreover, packages like those in this list are prone to mutate over time. For an up-to-date list of available tools, search the Web or browse <http://www.python.org> and the PyPI third-party packages index maintained there.

tkinter Overview

Of all the prior section's GUI options, though, tkinter is by far the de facto standard way to implement portable user interfaces in Python today, and the focus of this part of the book. The rationale for this approach was explained in [Chapter 1](#); in short, we elected to present one toolkit in satisfying depth instead of many toolkits in less-than-useful fashion. Moreover, most of the tkinter programming concepts you learn here will translate directly to any other GUI toolkit you choose to utilize.

tkinter Pragmatics

Perhaps more to the point, though, there are pragmatic reasons that the Python world still gravitates to tkinter as its de facto standard portable GUI toolkit. Among them, tkinter's accessibility, portability, availability, documentation, and extensions have made it the most widely used Python GUI solution for many years running:

Accessibility

tkinter is generally regarded as a *lightweight toolkit* and one of the simplest GUI solutions for Python available today. Unlike larger frameworks, it is easy to get started in tkinter right away, without first having to grasp a much larger class interaction model. As we'll see, programmers can create simple tkinter GUIs in a few lines of Python code and scale up to writing industrial-strength GUIs gradually. Although the tkinter API is basic, additional widgets can be coded in Python or obtained in extension packages such as Pmw, Tix, and ttk.

Portability

A Python script that builds a GUI with tkinter will run without source code changes on all major windowing platforms today: Microsoft Windows, X Windows (on Unix and Linux), and the Macintosh OS X (and also ran on Mac classics). Further, that same script will provide a native look-and-feel to its users on each of these platforms. In fact, this feature became more apparent as Tk matured. A Python/tkinter script today looks like a Windows program on Windows; on Unix and Linux, it provides the same interaction but sports an appearance familiar to X Windows users; and on the Mac, it looks like a Mac program should.

Availability

tkinter is a standard module in the Python library, shipped with the interpreter. If you have Python, you have tkinter. Moreover, most Python installation packages (including the standard Python self-installer for Windows, that provided on Mac OS X, and many Linux distributions) come with tkinter support bundled. Because of that, scripts written to use the tkinter module work immediately on most Python interpreters, without any extra installation steps. tkinter is also generally better supported than its alternatives today. Because the underlying Tk library is also used by the Tcl and Perl programming languages (and others), it tends to receive more development resources than other toolkits available.

Naturally, other factors such as documentation and extensions are important when using a GUI toolkit, too; let's take a quick look at the story tkinter has to tell on these fronts as well.

tkinter Documentation

This book explores tkinter fundamentals and most widgets tools, and it should be enough to get started with substantial GUI development in Python. On the other hand, it is not an exhaustive reference to the tkinter library or extensions to it. Happily, at least one book dedicated to using tkinter in Python is now commercially available as I write this paragraph, and others are on the way (search the Web for details). Besides books, you can also find tkinter documentation online; a complete set of tkinter manuals is currently maintained on the Web at <http://www.pythonware.com/library>.

In addition, because the underlying Tk toolkit used by tkinter is also a de facto standard in the open source scripting community at large, other documentation sources apply. For instance, because Tk has also been adopted by the Tcl and Perl programming languages, Tk-oriented books and documentation written for both of these are directly applicable to Python/tkinter as well (albeit, with some syntactic mapping).

Frankly, I learned tkinter by studying Tcl/Tk texts and references—just replace Tcl strings with Python objects and you have additional reference libraries at your disposal (see [Table 7-2](#), the Tk-to-tkinter conversion guide, at the end of this chapter for help reading Tk documentation). For instance, the book *Tcl/Tk Pocket Reference* (O'Reilly) can serve as a nice supplement to the tkinter tutorial material in this part of the book. Moreover, since Tk concepts are familiar to a large body of programmers, Tk support is also readily available on the Net.

After you've learned the basics, examples can help, too. You can find tkinter demo programs, besides those you'll study in this book, at various locations around the Web. Python itself includes a set of demo programs in the `Demos\tkinter` subdirectory of its source distribution package. The IDLE development GUI mentioned in the next section makes for an interesting code read as well.

tkinter Extensions

Because tkinter is so widely used, programmers also have access to precoded Python extensions designed to work with or augment it. Some of these may not yet be available for Python 3.X as I write this but are expected to be soon. For instance:

Pmw

Python Mega Widgets is an extension toolkit for building high-level compound widgets in Python using the tkinter module. It extends the tkinter API with a collection of more sophisticated widgets for advanced GUI development and a framework for implementing some of your own. Among the precoded and extensible megawidgets shipped with the package are notebooks, combo boxes, selection

widgets, paned widgets, scrolled widgets, dialog windows, button boxes, balloon help, and an interface to the Blt graph widget.

The interface to Pmw megawidgets is similar to that of basic tkinter widgets, so Python scripts can freely mix Pmw megawidgets with standard tkinter widgets. Moreover, Pmw is pure Python code, and so requires no C compiler or tools to install. To view its widgets and the corresponding code you use to construct them, run the *demos\All.py* script in the Pmw distribution package. You can find Pmw at <http://pmw.sourceforge.net>.

Tix

Tix is a collection of more than 40 advanced widgets, originally written for Tcl/Tk but now available for use in Python/tkinter programs. This package is now a Python standard library module, called `tkinter.tix`. Like Tk, the underlying Tix library is also shipped today with Python on Windows. In other words, on Windows, if you install Python, you also have Tix as a preinstalled library of additional widgets.

Tix includes many of the same devices as Pmw, including spin boxes, trees, tabbed notebooks, balloon help pop ups, paned windows, and much more. See the Python library manual's entry for the Tix module for more details. For a quick look at its widgets, as well as the Python source code used to program them, run the *tixwidgets.py* demonstration program in the *Demo\tix* directory of the Python source distribution (this directory is not installed by default on Windows and is prone to change—you can generally find it after fetching and unpacking Python's source code from Python.org).

ttk

Tk themed widgets, ttk, is a relatively new widget set which attempts to separate the code implementing a widget's behavior from that implementing its appearance. Widget classes handle state and callback invocation, whereas widget appearance is managed separately by themes. Much like Tix, this extension began life separately, but was very recently incorporated into Python's standard library in Python 3.1, as module `tkinter.ttk`.

Also like Tix, this extension comes with advanced widget types, some of which are not present in standard tkinter itself. Specifically, ttk comes with 17 widgets, 11 of which are already present in tkinter and are meant as replacements for some of tkinter's standard widgets, and 6 of which are new—Combobox, Notebook, Progressbar, Separator, Sizegrip and Treeview. In a nutshell, scripts import from the `ttk` module after `tkinter` in order to use its replacement widgets and configure style objects possibly shared by multiple widgets, instead of configuring widgets themselves.

As we'll see in this chapter, it's possible to provide a common look-and-feel for a set of widgets with standard tkinter, by subclassing its widget classes using normal OOP techniques (see “[Customizing Widgets with Classes](#)” on page 400). However, ttk offers additional style options and advanced widget types. For more details

on ttk widgets, see the entry in the Python library manual or search the Web; this book focuses on tkinter fundamentals, and tix and ttk are both too large to cover in a useful fashion here.

PIL

The Python Imaging Library (PIL) is an open source extension package that adds image-processing tools to Python. Among other things, it provides tools for image thumbnails, transforms, and conversions, and it extends the basic tkinter image object to add support for displaying many image file types. PIL, for instance, allows tkinter GUIs to display JPEG, TIFF, and PNG images not supported by the base tkinter toolkit itself (without extension, tkinter supports GIFs and a handful of bitmap formats). See the end of [Chapter 8](#) for more details and examples; we'll use PIL in this book in a number of image-related example scripts. PIL can be found at <http://www.pythonware.com> or via a web search.

IDLE

The IDLE integrated Python development environment is both written in Python with tkinter and shipped and installed with the Python package (if you have a recent Python interpreter, you should have IDLE too; on Windows, click the Start button, select the Programs menu, and click the Python entry to find it). IDLE provides syntax-coloring text editors for Python code, point-and-click debugging, and more, and is an example of tkinter's utility.

Others

Many of the extensions that provide visualization tools for Python are based on the tkinter library and its canvas widget. See the PyPI website and your favorite web search engine for more tkinter extension examples.

If you plan to do any commercial-grade GUI development with tkinter, you'll probably want to explore extensions such as Pmw, PIL, Tix, and ttk after learning tkinter basics in this text. They can save development time and add pizzazz to your GUIs. See the Python-related websites mentioned earlier for up-to-date details and links.

tkinter Structure

From a more nuts-and-bolts perspective, tkinter is an integration system that implies a somewhat unique program structure. We'll see what this means in terms of code in a moment, but here is a brief introduction to some of the terms and concepts at the core of Python GUI programming.

Implementation structure

Strictly speaking, tkinter is simply the name of Python's interface to *Tk*—a GUI library originally written for use with the Tcl programming language and developed by Tcl's creator, John Ousterhout. Python's tkinter module talks to Tk, and the Tk library in turn interfaces with the underlying window system: Microsoft Windows, X Windows

on Unix, or whatever GUI system your Python uses on your Macintosh. The portability of tkinter actually stems from the underling Tk library it wraps.

Python’s tkinter adds a software layer on top of Tk that allows Python scripts to call out to Tk to build and configure interfaces and routes control back to Python scripts that handle user-generated events (e.g., mouse clicks). That is, GUI calls are internally routed from Python script, to tkinter, to Tk; GUI events are routed from Tk, to tkinter, and back to a Python script. In [Chapter 20](#), we’ll know these transfers by their C integration terms, *extending* and *embedding*.

Technically, tkinter is today structured as a combination of the Python-coded `tkinter` module package’s files and an extension module called `_tkinter` that is written in C. `_tkinter` interfaces with the Tk library using extending tools and dispatches callbacks back to Python objects using embedding tools; `tkinter` simply adds a class-based interface on top of `_tkinter`. You should almost always import `tkinter` in your scripts, though, not `_tkinter`; the latter is an implementation module meant for internal use only (and was oddly named for that reason).

Programming structure

Luckily, Python programmers don’t normally need to care about all this integration and call routing going on internally; they simply make widgets and register Python functions to handle widget events. Because of the overall structure, though, event handlers are usually known as *callback* handlers, because the GUI library “calls back” to Python code when events occur.

In fact, we’ll find that Python/tkinter programs are entirely *event driven*: they build displays and register handlers for events, and then do nothing but wait for events to occur. During the wait, the Tk GUI library runs an event loop that watches for mouse clicks, keyboard presses, and so on. All application program processing happens in the registered callback handlers in response to events. Further, any information needed across events must be stored in long-lived references such as global variables and class instance attributes. The notion of a traditional linear program control flow doesn’t really apply in the GUI domain; you need to think in terms of smaller chunks.

In Python, Tk also becomes *object oriented* simply because Python is object oriented: the tkinter layer exports Tk’s API as Python classes. With tkinter, we can either use a simple function-call approach to create widgets and interfaces, or apply object-oriented techniques such as inheritance and composition to customize and extend the base set of tkinter classes. Larger tkinter GUIs are generally constructed as trees of linked tkinter widget objects and are often implemented as Python classes to provide structure and retain state information between events. As we’ll see in this part of the book, a tkinter GUI coded with classes almost by default becomes a reusable software component.

Climbing the GUI Learning Curve

On to the code; let's start out by quickly stepping through a few small examples that illustrate basic concepts and show the windows they create on the computer display. The examples will become progressively more sophisticated as we move along, but let's get a handle on the fundamentals first.

“Hello World” in Four Lines (or Less)

The usual first example for GUI systems is to show how to display a “Hello World” message in a window. As coded in [Example 7-1](#), it's just four lines in Python.

Example 7-1. PP4E\Gui\Intro\gui1.py

```
from tkinter import Label  
widget = Label(None, text='Hello GUI world!')  
widget.pack()  
widget.mainloop()
```

get a widget object
make one
arrange it
start event loop

This is a complete Python tkinter GUI program. When this script is run, we get a simple window with a label in the middle; it looks like [Figure 7-1](#) on my Windows 7 laptop (I stretched some windows in this book horizontally to reveal their window titles; your platform's window system may vary).

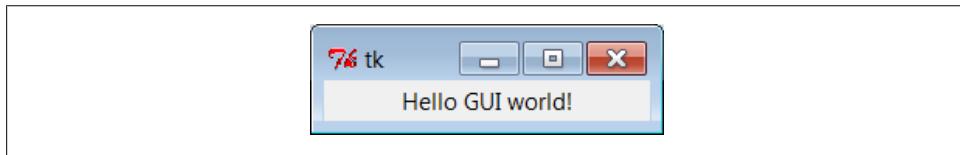


Figure 7-1. “Hello World” (gui1) on Windows

This isn't much to write home about yet, but notice that this is a completely functional, independent window on the computer's display. It can be maximized to take up the entire screen, minimized to hide it in the system bar, and resized. Click on the window's “X” box in the top right to kill the window and exit the program.

The script that builds this window is also fully portable. Run this script on your machine to see how it renders. When this same file is run on Linux it produces a similar window, but it behaves according to the underlying Linux window manager. Even on the same operating system, the same Python code might yield a different look-and-feel for different window systems (for instance, under KDE and Gnome on Linux). The same script file would look different still when run on Macintosh and other Unix-like window managers. On all platforms, though, its basic functional behavior will be the same.

tkinter Coding Basics

The `gui1` script is a trivial example, but it illustrates steps common to most tkinter programs. This Python code does the following:

1. Loads a widget class from the `tkinter` module
2. Makes an instance of the imported `Label` class
3. Packs (arranges) the new `Label` in its parent widget
4. Calls `mainloop` to bring up the window and start the tkinter event loop

The `mainloop` method called last puts the label on the screen and enters a tkinter wait state, which watches for user-generated GUI events. Within the `mainloop` function, tkinter internally monitors things such as the keyboard and mouse to detect user-generated events. In fact, the tkinter `mainloop` function is similar in spirit to the following pseudo-Python code:

```
def mainloop():
    while the main window has not been closed:
        if an event has occurred:
            run the associated event handler function
```

Because of this model, the `mainloop` call in [Example 7-1](#) never returns to our script while the GUI is displayed on-screen.[‡] When we write larger scripts, the only way we can get anything done after calling `mainloop` is to register callback handlers to respond to events.

This is called *event-driven programming*, and it is perhaps one of the most unusual aspects of GUIs. GUI programs take the form of a set of event handlers that share saved information rather than of a single main control flow. We'll see how this looks in terms of real code in later examples.

Note that for code in a script file, you really need to do steps 3 and 4 in the preceding list to open this script's GUI. To display a GUI's window at all, you need to call `mainloop`; to display widgets within the window, they must be packed (or otherwise arranged) so that the tkinter geometry manager knows about them. In fact, if you call either `mainloop` or `pack` without calling the other, your window won't show up as expected: a `mainloop` without a `pack` shows an empty window, and a `pack` without a `mainloop` in a script shows nothing since the script never enters an event wait state (try it). The `mainloop` call is sometimes optional when you're coding interactively, but you shouldn't rely on this in general.

Since the concepts illustrated by this simple script are at the core of most tkinter programs, let's take a deeper look at some of them before moving on.

[‡] Technically, the `mainloop` call returns to your script only after the tkinter event loop exits. This normally happens when the GUI's main window is closed, but it may also occur in response to explicit `quit` method calls that terminate nested event loops but leave open the GUI at large. You'll see why this matters in [Chapter 8](#).

Making Widgets

When widgets are constructed in tkinter, we can specify how they should be configured. The `gui1` script passes two arguments to the `Label` class constructor:

- The first is a parent-widget object, which we want the new label to be attached to. Here, `None` means “attach the new `Label` to the default top-level window of this program.” Later, we’ll pass real widgets in this position to attach our labels to other container objects.
- The second is a configuration option for the `Label`, passed as a keyword argument: the `text` option specifies a text string to appear as the label’s message. Most widget constructors accept multiple keyword arguments for specifying a variety of options (color, size, callback handlers, and so on). Most widget configuration options have reasonable defaults per platform, though, and this accounts for much of tkinter’s simplicity. You need to set most options only if you wish to do something custom.

As we’ll see, the parent-widget argument is the hook we use to build up complex GUIs as widget trees. tkinter works on a “what-you-build-is-what-you-get” principle—we construct widget object trees as models of what we want to see on the screen, and then ask the tree to display itself by calling `mainloop`.

Geometry Managers

The `pack` widget method called by the `gui1` script invokes the packer geometry manager, one of three ways to control how widgets are arranged in a window. tkinter geometry managers simply arrange one or more widgets within a container (sometimes called a parent or master). Both top-level windows and frames (a special kind of widget we’ll meet later) can serve as containers, and containers may be nested inside other containers to build hierarchical displays.

The packer geometry manager uses constraint option settings to automatically position widgets in a window. Scripts supply higher-level instructions (e.g., “attach this widget to the top of its container, and stretch it to fill its space vertically”), not absolute pixel coordinates. Because such constraints are so abstract, the packer provides a powerful and easy-to-use layout system. In fact, you don’t even have to specify constraints. If you don’t pass any arguments to `pack`, you get default packing, which attaches the widget to the top side of its container.

We’ll visit the packer repeatedly in this chapter and use it in many of the examples in this book. In [Chapter 9](#), we will also meet an alternative `grid` geometry manager—a layout system that arranges widgets within a container in tabular form (i.e., by rows and columns) and works well for input forms. A third alternative, called the *placer* geometry manager system, is described in Tk documentation but not in this book; it’s less popular than the `pack` and `grid` managers and can be difficult to use for larger GUIs coded by hand.

Running GUI Programs

Like all Python code, the module in [Example 7-1](#) can be started in a number of ways—by running it as a top-level program file:

```
C:\...\PP4E\Gui\Intro> python gui1.py
```

by importing it from a Python session or another module file:

```
>>> import gui1
```

by running it as a Unix executable if we add the special `#!` line at the top:

```
% gui1.py &
```

and in any other way Python programs can be launched on your platform. For instance, the script can also be run by clicking on the file's name in a Windows file explorer, and its code can be typed interactively at the `>>>` prompt.[§] It can even be run from a C program by calling the appropriate embedding API function (see [Chapter 20](#) for details on C integration).

In other words, there are really no special rules to follow when launching GUI Python code. The tkinter interface (and Tk itself) is linked into the Python interpreter. When a Python program calls GUI functions, they're simply passed to the embedded GUI system behind the scenes. That makes it easy to write command-line tools that pop up windows; they are run the same way as the purely text-based scripts we studied in the prior part of this book.

Avoiding DOS consoles on Windows

In Chapters [3](#) and [6](#) we noted that if a program's name ends in a `.pyw` extension rather than a `.py` extension, the Windows Python port does not pop up a DOS console box to serve as its standard streams when the file is launched by clicking its filename icon. Now that we've finally started making windows of our own, that filename trick will start to become even more useful.

If you just want to see the windows that your script makes no matter how it is launched, be sure to name your GUI scripts with a `.pyw` if they might be run on Windows. For instance, clicking on the file in [Example 7-2](#) in a Windows explorer creates just the window in [Figure 7-1](#).

Example 7-2. PP4E\Gui\Intro\gui1.pyw

...same as gui1.py...

[§] Tip: As suggested earlier, when typing tkinter GUI code *interactively*, you may or may not need to call `mainloop` to display widgets. This is required in the current IDLE interface, but not from a simple interactive session running in a system console window. In either case, control will return to the interactive prompt when you kill the window you created. Note that if you create an explicit main-window widget by calling `Tk()` and attach widgets to it (described later), you must call this again after killing the window; otherwise, the application window will not exist.

You can also avoid the DOS pop up on Windows by running the program with the `pythonw.exe` executable, not `python.exe` (in fact, `.pyw` files are simply registered to be opened by `pythonw`). On Linux, the `.pyw` doesn't hurt, but it isn't necessary; there is no notion of a streams pop up on Unix-like machines. On the other hand, if your GUI scripts might run on Windows in the future, adding an extra "w" at the end of their names now might save porting effort later. In this book, `.py` filenames are still sometimes used to pop up console windows for viewing printed messages on Windows.

tkinter Coding Alternatives

As you might expect, there are a variety of ways to code the `gui1` example. For instance, if you want to make all your tkinter imports more explicit in your script, grab the whole module and prefix all of its names with the module's name, as in [Example 7-3](#).

Example 7-3. PP4E\Gui\Intro\gui1b.py—import versus from

```
import tkinter
widget = tkinter.Label(None, text='Hello GUI world!')
widget.pack()
widget.mainloop()
```

That will probably get tedious in realistic examples, though—tkinter exports dozens of widget classes and constants that show up all over Python GUI scripts. In fact, it is usually easier to use a `*` to import everything from the tkinter module by name in one shot. This is demonstrated in [Example 7-4](#).

Example 7-4. PP4E\Gui\Intro\gui1c.py—roots, sides, pack in place

```
from tkinter import *
root = Tk()
Label(root, text='Hello GUI world!).pack(side=TOP)
root.mainloop()
```

The tkinter module goes out of its way to export only what we really need, so it's one of the few for which the `*` import form is relatively safe to apply.¹¹ The `TOP` constant in the `pack` call here, for instance, is one of those many names exported by the `tkinter` module. It's simply a variable name (`TOP="top"`) preassigned in `constants`, a module automatically loaded by `tkinter`.

When widgets are packed, we can specify which side of their parent they should be attached to—`TOP`, `BOTTOM`, `LEFT`, or `RIGHT`. If no `side` option is sent to `pack` (as in prior examples), a widget is attached to its parent's `TOP` by default. In general, larger tkinter GUIs can be constructed as sets of rectangles, attached to the appropriate sides of other,

¹¹ If you study the main tkinter file in the Python source library (currently, `Lib\tkinter_init__.py`), you'll notice that top-level module names not meant for export start with a single underscore. Python never copies over such names when a module is accessed with the `*` form of the `from` statement. The constants module is today `constants.py` in the same module package directory, though this can change (and has) over time.

enclosing rectangles. As we'll see later, tkinter arranges widgets in a rectangle according to both their packing order and their `side` attachment options. When widgets are gridded, they are assigned row and column numbers instead. None of this will become very meaningful, though, until we have more than one widget in a window, so let's move on.

Notice that this version calls the `pack` method right away after creating the label, without assigning it a variable. If we don't need to save a widget, we can pack it in place like this to eliminate a statement. We'll use this form when a widget is attached to a larger structure and never again referenced. This can be tricky if you assign the `pack` result, though, but I'll postpone an explanation of why until we've covered a few more basics.

We also use a `Tk` widget class instance, instead of `None`, as the parent here. `Tk` represents the main ("root") window of the program—the one that starts when the program does. An automatically created `Tk` instance is also used as the default parent widget, both when we don't pass any parent to other widget calls and when we pass the parent as `None`. In other words, widgets are simply attached to the main program window by default. This script just makes this default behavior explicit by making and passing the `Tk` object itself. In [Chapter 8](#), we'll see that `Toplevel` widgets are typically used to generate new pop-up windows that operate independently of the program's main window.

In tkinter, some widget methods are exported as functions, and this lets us shave [Example 7-5](#) to just three lines of code.

Example 7-5. PP4E\Gui\Intro\gui1d.py—a minimal version

```
from tkinter import *
Label(text='Hello GUI world!').pack()
mainloop()
```

The tkinter `mainloop` can be called with or without a widget (i.e., as a function or method). We didn't pass `Label` a parent argument in this version, either: it simply defaults to `None` when omitted (which in turn defaults to the automatically created `Tk` object). But relying on that default is less useful once we start building larger displays. Things such as labels are more typically attached to other widget containers.

Widget Resizing Basics

Top-level windows, such as the one built by all of the coding variants we have seen thus far, can normally be resized by the user; simply drag out the window with your mouse. [Figure 7-2](#) shows how our window looks when it is expanded.

This isn't very good—the label stays attached to the top of the parent window instead of staying in the middle on expansion—but it's easy to improve on this with a pair of `pack` options, demonstrated in [Example 7-6](#).



Figure 7-2. Expanding gui1

Example 7-6. PP4E\Gui\Intro\gui1e.py—expansion

```
from tkinter import *
Label(text='Hello GUI world!).pack(expand=YES, fill=BOTH)
mainloop()
```

When widgets are packed, we can specify whether a widget should expand to take up all available space, and if so, how it should stretch to fill that space. By default, widgets are not expanded when their parent is. But in this script, the names YES and BOTH (imported from the `tkinter` module) specify that the label should grow along with its parent, the main window. It does so in [Figure 7-3](#).

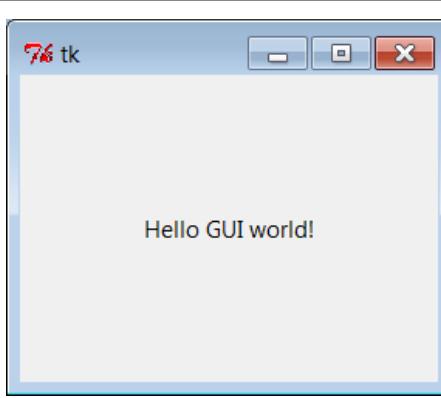


Figure 7-3. gui1e with widget resizing

Technically, the packer geometry manager assigns a size to each widget in a display based on what it contains (text string lengths, etc.). By default, a widget can occupy only its allocated space and is no bigger than its assigned size. The `expand` and `fill` options let us be more specific about such things:

`expand=YES option`

Asks the packer to expand the allocated space for the widget in general into any unclaimed space in the widget’s parent.

`fill option`

Can be used to stretch the widget to occupy all of its allocated space.

Combinations of these two options produce different layout and resizing effects, some of which become meaningful only when there are multiple widgets in a window. For example, using `expand` without `fill` centers the widget in the expanded space, and the `fill` option can specify vertical stretching only (`fill=Y`), horizontal stretching only (`fill=X`), or both (`fill=BOTH`). By providing these constraints and attachment sides for all widgets in a GUI, along with packing order, we can control the layout in fairly precise terms. In later chapters, we’ll find that the `grid` geometry manager uses a different resizing protocol entirely, but it provides similar control when needed.

All of this can be confusing the first time you hear it, and we’ll return to this later. But if you’re not sure what an `expand` and `fill` combination will do, simply try it out—this is Python, after all. For now, remember that the combination of `expand=YES` and `fill=BOTH` is perhaps the most common setting; it means “expand my space allocation to occupy all available space on my side, and stretch me to fill the expanded space in both directions.” For our “Hello World” example, the net result is that the label grows as the window is expanded, and so is always centered.

Configuring Widget Options and Window Titles

So far, we’ve been telling tkinter what to display on our label by passing its text as a keyword argument in label constructor calls. It turns out that there are two other ways to specify widget configuration options. In [Example 7-7](#), the `text` option of the label is set after it is constructed, by assigning to the widget’s `text` key. Widget objects overload (intercept) index operations such that options are also available as mapping keys, much like a dictionary.

Example 7-7. PP4E\Gui\Intro\gui1f.py—option keys

```
from tkinter import *
widget = Label()
widget['text'] = 'Hello GUI world!'
widget.pack(side=TOP)
mainloop()
```

More commonly, widget options can be set after construction by calling the widget `config` method, as in [Example 7-8](#).

Example 7-8. PP4E\Gui\Intro\gui1g.py—config and titles

```
from tkinter import *
root = Tk()
widget = Label(root)
widget.config(text='Hello GUI world!')
widget.pack(side=TOP, expand=YES, fill=BOTH)
root.title('gui1g.py')
root.mainloop()
```

The `config` method (which can also be called by its synonym, `configure`) can be called at any time after construction to change the appearance of a widget on the fly. For instance, we could call this label’s `config` method again later in the script to change the text that it displays; watch for such dynamic reconfigurations in later examples in this part of the book.

Notice that this version also calls a `root.title` method; this call sets the label that appears at the top of the window, as pictured in [Figure 7-4](#). In general terms, top-level windows like the `Tk` `root` here export window-manager interfaces—i.e., things that have to do with the border around the window, not its contents.



Figure 7-4. gui1g with expansion and a window title

Just for fun, this version also centers the label upon resizes by setting the `expand` and `fill` pack options. In fact, this version makes just about everything explicit and is more representative of how labels are often coded in full-blown interfaces; their parents, expansion policies, and attachments are usually spelled out rather than defaulted.

One More for Old Times’ Sake

Finally, if you are a minimalist and you’re nostalgic for old Python coding styles, you can also program this “Hello World” example as in [Example 7-9](#).

Example 7-9. PP4E\Gui\Intro\gui1-old.py—dictionary calls

```
from tkinter import *
Label(None, {'text': 'Hello GUI world!', Pack: {'side': 'top'}}).mainloop()
```

This makes the window in just two lines, albeit arguably gruesome ones! This scheme relies on an old coding style that was widely used until Python 1.3, which passed configuration options in a dictionary instead of keyword arguments.[#] In this scheme, packer options can be sent as values of the key `Pack` (a class in the `tkinter` module).

The dictionary call scheme still works and you may see it in old Python code, but it's probably best to not do this in code you type. Use keywords to pass options, and use explicit `pack` method calls in your `tkinter` scripts instead. In fact, the only reason I didn't cut this example completely is that dictionaries can still be useful if you want to compute and pass a set of options dynamically.

On the other hand, the `func(*pargs, **kargs)` syntax now also allows you to pass an explicit dictionary of keyword arguments in its third argument slot:

```
options = {'text': 'Hello GUI world!'}
layout  = {'side': 'top'}
Label(None, **options).pack(**layout)      # keyword must be strings
```

Even in dynamic scenarios where widget options are determined at run time, there's no compelling reason to ever use the pre-1.3 `tkinter` dictionary call form.

Packing Widgets Without Saving Them

In `gui1c.py` (shown in [Example 7-4](#)), I started packing labels without assigning them to names. This works, and it is an entirely valid coding style, but because it tends to confuse beginners at first glance, I need to explain why it works in more detail here.

In `tkinter`, Python class objects correspond to real objects displayed on a screen; we make the Python object to make a screen object, and we call the Python object's methods to configure that screen object. Because of this correspondence, the lifetime of the Python object must generally correspond to the lifetime of the corresponding object on the screen.

Luckily, Python scripts don't usually have to care about managing object lifetimes. In fact, they do not normally need to maintain a reference to widget objects created along the way at all unless they plan to reconfigure those objects later. For instance, it's common in `tkinter` programming to pack a widget immediately after creating it if no further reference to the widget is required:

```
Label(text='hi').pack()                      # OK
```

This expression is evaluated left to right, as usual. It creates a new label and then immediately calls the new object's `pack` method to arrange it in the display. Notice, though, that the Python `Label` object is temporary in this expression; because it is not

[#]In fact, Python's pass-by-name keyword arguments were first introduced to help clean up `tkinter` calls such as this one. Internally, keyword arguments really are passed as a dictionary (which can be collected with the `**name` argument form in a `def` header), so the two schemes are similar in implementation. But they vary widely in the number of characters you need to type and debug.

assigned to a name, it would normally be garbage collected (destroyed and reclaimed) by Python immediately after running its `pack` method.

However, because tkinter emits Tk calls when objects are constructed, the label will be drawn on the display as expected, even though we haven't held onto the corresponding Python object in our script. In fact, tkinter internally cross-links widget objects into a long-lived tree used to represent the display, so the `Label` object made during this statement actually is retained, even if not by our code.*

In other words, your scripts don't generally have to care about widget object lifetimes, and it's OK to make widgets and pack them immediately in the same statement without maintaining a reference to them explicitly in your code.

But that does not mean that it's OK to say something like this:

```
widget = Label(text='hi').pack()           # wrong!
...use widget...
```

This statement almost seems like it should assign a newly packed label to `widget`, but it does not do this. In fact, it's really a notorious tkinter beginner's mistake. The widget `pack` method packs the widget but does not return the widget thus packed. Really, `pack` returns the Python object `None`; after such a statement, `widget` will be a reference to `None`, and any further widget operations through that name will fail. For instance, the following fails, too, for the same reason:

```
Label(text='hi').pack().mainloop()        # wrong!
```

Since `pack` returns `None`, asking for its `mainloop` attribute generates an exception (as it should). If you really want to both pack a widget and retain a reference to it, say this instead:

```
widget = Label(text='hi')                 # OK too
widget.pack()
...use widget...
```

This form is a bit more verbose but is less tricky than packing a widget in the same statement that creates it, and it allows you to hold onto the widget for later processing. It's probably more common in realistic scripts that perform more complex widget configuration and layouts.

On the other hand, scripts that compose layouts often add some widgets once and for all when they are created and never need to reconfigure them later; assigning to long-lived names in such programs is pointless and unnecessary.

* Ex-Tcl programmers in the audience may be interested to know that, at least at the time I was writing this footnote, Python not only builds the widget tree internally, but uses it to automatically generate widget pathname strings coded manually in Tcl/Tk (e.g., `.panel.row.cmd`). Python uses the addresses of widget class objects to fill in the path components and records pathnames in the widget tree. A label attached to a container, for instance, might have an assigned name such as `.8220096.8219408` inside tkinter. You don't have to care, though. Simply make and link widget objects by passing parents, and let Python manage pathname details based on the object tree. See the end of this chapter for more on Tk/tkinter mappings.



In [Chapter 8](#), we'll meet two exceptions to this rule. Scripts must manually retain a reference to *image* objects because the underlying image data is discarded if the Python image object is garbage collected. tkinter variable class objects also temporarily unset an associated *Tk variable* if reclaimed, but this is uncommon and less harmful.

Adding Buttons and Callbacks

So far, we've learned how to display messages in labels, and we've met tkinter core concepts along the way. Labels are nice for teaching the basics, but user interfaces usually need to do a bit more...like actually responding to users. To show how, the program in [Example 7-10](#) creates the window in [Figure 7-5](#).

Example 7-10. PP4E\Gui\Intro\gui2.py

```
import sys
from tkinter import *
widget = Button(None, text='Hello widget world', command=sys.exit)
widget.pack()
widget.mainloop()
```

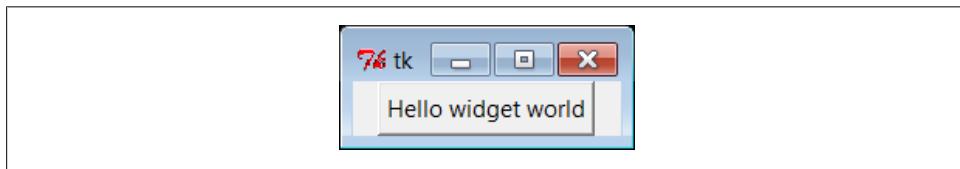


Figure 7-5. A button on the top

Here, instead of making a label, we create an instance of the tkinter `Button` class. It's attached to the default top level window as before on the default `TOP` packing side. But the main thing to notice here is the button's configuration arguments: we set an option called `command` to the `sys.exit` function.

For buttons, the `command` option is the place where we specify a callback handler function to be run when the button is later pressed. In effect, we use `command` to register an action for tkinter to call when a widget's event occurs. The callback handler used here isn't very interesting: as we learned in [Chapter 5](#), the built-in `sys.exit` function simply shuts down the calling program. Here, that means that pressing this button makes the window go away.

Just as for labels, there are other ways to code buttons. [Example 7-11](#) is a version that packs the button in place without assigning it to a name, attaches it to the `LEFT` side of its parent window explicitly, and specifies `root.quit` as the callback handler—a standard Tk object method that shuts down the GUI and so ends the program. Technically, `quit` ends the current `mainloop` event loop call, and thus the entire program here; when

we start using multiple top-level windows in [Chapter 8](#), we'll find that `quit` usually closes all windows, but its relative `destroy` erases just one window.

Example 7-11. PP4E\Gui\Intro\gui2b.py

```
from tkinter import *
root = Tk()
Button(root, text='press', command=root.quit).pack(side=LEFT)
root.mainloop()
```

This version produces the window in [Figure 7-6](#). Because we didn't tell the button to expand into all available space, it does not do so.



Figure 7-6. A button on the left

In both of the last two examples, pressing the button makes the GUI program exit. In older tkinter code, you may sometimes see the string `exit` assigned to the `command` option to make the GUI go away when pressed. This exploits a tool in the underlying Tk library and is less Pythonic than `sys.exit` or `root.quit`.

Widget Resizing Revisited: Expansion

Even with a GUI this simple, there are many ways to lay out its appearance with tkinter's constraint-based `pack` geometry manager. For example, to center the button in its window, add an `expand=YES` option to the button's `pack` method call in [Example 7-11](#). The line of changed code looks like this:

```
Button(root, text='press', command=root.quit).pack(side=LEFT, expand=YES)
```

This makes the packer allocate all available space to the button but does not stretch the button to fill that space. The result is the window captured in [Figure 7-7](#).

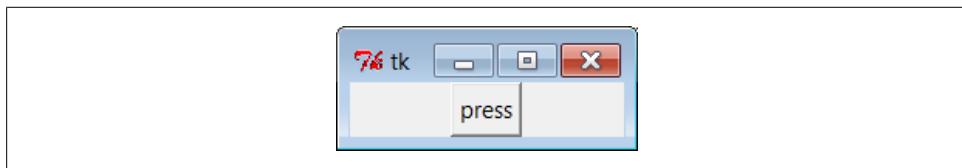


Figure 7-7. pack(side=LEFT, expand=YES)

If you want the button to be given all available space *and* to stretch to fill all of its assigned space horizontally, add `expand=YES` and `fill=X` keyword arguments to the `pack` call. This will create the scene in [Figure 7-8](#).

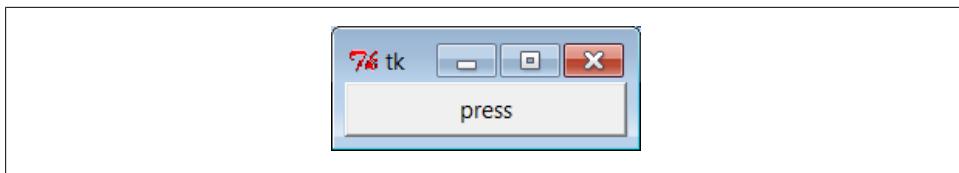


Figure 7-8. `pack(side=LEFT, expand=YES, fill=X)`

This makes the button fill the whole window initially (its allocation is expanded, and it is stretched to fill that allocation). It also makes the button grow as the parent window is resized. As shown in [Figure 7-9](#), the button in this window does expand when its parent expands, but only along the X horizontal axis.

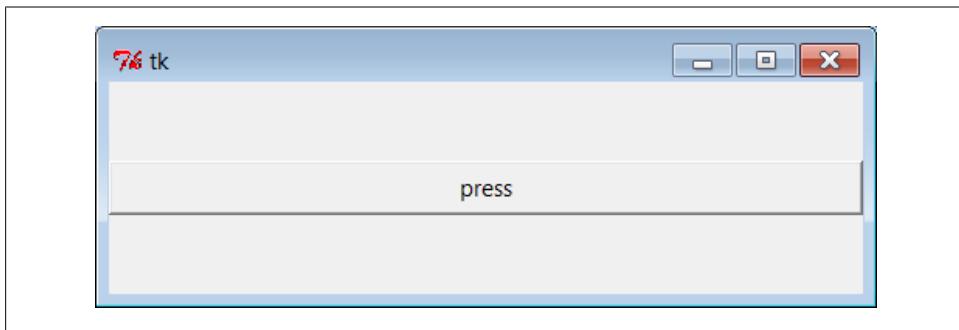


Figure 7-9. Resizing with `expand=YES, fill=X`

To make the button grow in both directions, specify both `expand=YES` and `fill=BOTH` in the `pack` call; now resizing the window makes the button grow in general, as shown in [Figure 7-10](#). In fact, for more fun, maximize this window to fill the entire screen; you'll get one very big tkinter button indeed.



Figure 7-10. Resizing with `expand=YES, fill=BOTH`

In more complex displays, such a button will expand only if all of the widgets it is contained by are set to expand too. Here, the button's only parent is the Tk root window of the program, so parent expandability isn't yet an issue; in later examples, we'll need to make enclosing Frame widgets expandable too. We will revisit the packer geometry manager when we meet multiple-widget displays that use such devices later in this tutorial, and again when we study the alternative grid call in [Chapter 9](#).

Adding User-Defined Callback Handlers

In the simple button examples in the preceding section, the callback handler was simply an existing function that killed the GUI program. It's not much more work to register callback handlers that do something a bit more useful. [Example 7-12](#) defines a callback handler of its own in Python.

Example 7-12. PP4E\Gui\Intro\gui3.py

```
import sys
from tkinter import *

def quit():
    print('Hello, I must be going...')      # a custom callback handler
    sys.exit()                            # kill windows and process

widget = Button(None, text='Hello event world', command=quit)
widget.pack()
widget.mainloop()
```

The window created by this script is shown in [Figure 7-11](#). This script and its GUI are almost identical to the last example. But here, the command option specifies a function we've defined locally. When the button is pressed, tkinter calls the quit function in this file to handle the event, passing it zero arguments. Inside quit, the print call statement types a message on the program's stdout stream, and the GUI process exits as before.

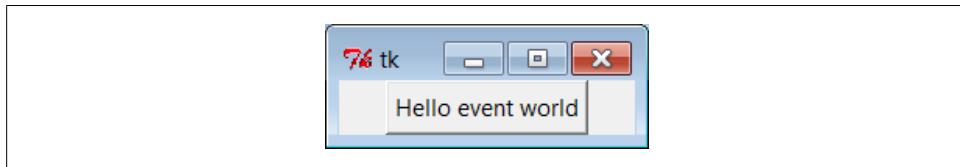


Figure 7-11. A button that runs a Python function

As usual, stdout is normally the window that the program was started from unless it's been redirected to a file. It's a pop-up DOS console if you run this program by clicking it on Windows; add an input call before sys.exit if you have trouble seeing the message before the pop up disappears. Here's what the printed output looks like back in standard stream world when the button is pressed; it is generated by a Python function called automatically by tkinter:

```
C:\...\PP4E\Gui\Intro> python gui3.py
Hello, I must be going...
```

```
C:\...\PP4E\Gui\Intro>
```

Normally, such messages would be displayed in the GUI, but we haven't gotten far enough to know how just yet. Callback functions usually do more, of course (and may even pop up new independent windows altogether), but this example illustrates the basics.

In general, callback handlers can be any callable object: functions, anonymous functions generated with lambda expressions, bound methods of class or type instances, or class instances that inherit a `__call__` operator overload method. For `Button` press callbacks, callback handlers always receive no arguments (other than an automatic `self`, for bound methods); any state information required by the callback handler must be provided in other ways—as global variables, class instance attributes, extra arguments provided by an indirection layer, and so on.

To make this a bit more concrete, let's take a quick look at some other ways to code the callback handler in this example.

Lambda Callback Handlers

Recall that the Python lambda expression generates a new, unnamed function object when run. If we need extra data passed in to the handler function, we can register lambda expressions to defer the call to the real handler function, and specify the extra data it needs.

Later in this part of the book, we'll see how this can be more useful, but to illustrate the basic idea, [Example 7-13](#) shows what [Example 7-12](#) looks like when recoded to use a lambda instead of a `def`.

Example 7-13. PP4E\Gui\Intro\gui3b.py

```
import sys
from tkinter import *                                     # lambda generates a function

widget = Button(None,                                     # but contains just an expression
                text='Hello event world',
                command=(lambda: print('Hello lambda world') or sys.exit()) )

widget.pack()
widget.mainloop()
```

This code is a bit tricky because lambdas can contain only an expression; to emulate the original script, this version uses an `or` operator to force two expressions to be run (`print` works as the first, because it's a function call in Python 3.X—we don't need to resort to using `sys.stdout` directly).

Deferring Calls with Lambdas and Object References

More typically, lambdas are used to provide an indirection layer that passes along extra data to a callback handler (I omit `pack` and `mainloop` calls in the following snippets for simplicity):

```
def handler(A, B):          # would normally be called with no args
    ...use A and B...

X = 42
Button(text='ni', command=(lambda: handler(X, 'spam')))      # lambda adds arguments
```

Although tkinter invokes `command` callbacks with no arguments, such a lambda can be used to provide an indirect anonymous function that wraps the real handler call and passes along information that existed when the GUI was first constructed. The call to the real handler is, in effect, *deferred*, so we can add the extra arguments it requires. Here, the value of global variable `X` and string '`spam`' will be passed to arguments `A` and `B`, even though tkinter itself runs callbacks with no arguments. The net effect is that the lambda serves to map a no-argument function call to one with arguments supplied by the lambda.

If lambda syntax confuses you, remember that a lambda expression such as the one in the preceding code can usually be coded as a simple `def` statement instead, nested or otherwise. In the following code, the second function does exactly the same work as the prior lambda—by referencing it in the button creation call, it effectively defers invocation of the actual callback handler so that extra arguments can be passed:

```
def handler(A, B):          # would normally be called with no args
    ...use A and B...

X = 42
def func():                  # indirection layer to add arguments
    handler(X, 'spam')

Button(text='ni', command=func)
```

To make the need for deferrals more obvious, notice what happens if you code a handler call in the button creation call itself without a lambda or other intermediate function—the callback runs immediately *when the button is created*, not when it is later clicked. That's why we need to wrap the call in an intermediate function to defer its invocation:

```
def handler(name):
    print(name)

Button(command=handler('spam'))          # BAD: runs the callback now!
```

Using either a lambda or a callable reference serves to defer callback invocation until the event later occurs. For example, using a lambda to pass extra data with an inline function definition that defers the call:

```
def handler(name):
    print(name)
```

```
Button(command=(lambda: handler('spam')))    # OK: wrap in a lambda to defer
```

is always equivalent to the longer, and to some observers less convenient, double-function form:

```
def handler(name):
    print(name)

def temp():
    handler('spam')

Button(command=temp)                                # OK: reference but do not call
```

We need only the zero-argument lambda or the zero-argument callable reference, though, *not both*—it makes no sense to code a lambda which simply calls a function if no extra data must be passed in and only adds an extra pointless call:

```
def handler(name):
    print(name)

def temp():
    handler('spam')

Button(command=(lambda: temp()))                   # BAD: this adds a pointless call!
```

As we'll see later, this includes references to other callables like bound methods and callable instances which retain state in themselves—if they take zero arguments when called, we can simply name them at widget construction time, and we don't need to wrap them in a superfluous lambda.

Callback Scope Issues

Although the prior section's lambda and intermediate function techniques defer calls and allow extra data to be passed in, they also raise some scoping issues that may seem subtle at first glance. This is core language territory, but it comes up often in practice in conjunction with GUI.

Arguments versus globals

For instance, notice that the `handler` function in the prior section's initial code could also refer to `X` directly, because it is a global variable (and would exist by the time the code inside the handler is run). Because of that, we might make the handler a one-argument function and pass in just the string '`spam`' in the lambda:

```
def handler(A):                                # X is in my global scope, implicitly
    ...use global X and argument A...

X = 42
Button(text='ni', command=(lambda: handler('spam')))
```

For that matter, `A` could be moved out to the global scope too, to remove the need for `lambda` here entirely; we could register the handler itself and cut out the middleman.

Although simple in this trivial example, arguments are generally preferred to globals, because they make external dependencies more explicit, and so make code easier to understand and change. In fact, the same handler might be usable in other contexts, if we don't couple it to global variables' values. While you'll have to take it on faith until we step up to larger examples with more complex state retention needs, avoiding glob-als in callbacks and GUIs in general both makes them more reusable, and supports the notion of multiple instances in the same program. It's good programming practice, GUI or not.

Passing in enclosing scope values with default arguments

More subtly, notice that if the button in this example was constructed inside a *function* rather than at the top level of the file, name `X` would no longer be global but would be in the enclosing function's local scope; it seems as if it would disappear after the function exits and before the callback event occurs and runs the lambda's code:

```
def handler(A, B):
    ...use A and B...

def makegui():
    X = 42
    Button(text='ni', command=(lambda: handler(X, 'spam')))      # remembers X

makegui()
mainloop()  # makegui's scope is gone by this point
```

Luckily, Python's enclosing scope reference model means that the value of `X` in the local scope enclosing the lambda function is automatically retained, for use later when the button press occurs. This usually works as we want today, and automatically handles variable references in this role.

To make such enclosing scope usage explicit, though, default argument values can also be used to remember the values of variables in the enclosing local scope, even after the enclosing function returns. In the following code, for instance, the default argument name `X` (on the left side of the `X=X` default) will remember object `42`, because the variable name `X` (on the right side of the `X=X`) is evaluated in the enclosing scope, and the generated function is later called without any arguments:

```
def handler(A, B):          # older Pythons: defaults save state
    ...use A and B...

def makegui():
    X = 42
    Button(text='ni', command=(lambda X=X: handler(X, 'spam')))
```

Since default arguments are evaluated and saved when the lambda runs (not when the function it creates is later called), they are a way to explicitly remember objects that

must be accessed again later, during event processing. Because tkinter calls the lambda function later with no arguments, all its defaults are used.

This was not an issue in the original version of this example because name `X` lived in the global scope, and the code of the lambda will find it there when it is run. When nested within a function, though, `X` may have disappeared after the enclosing function exits.

Passing in enclosing scope values with automatic references

While they can make some external dependencies more explicit, defaults are not usually required (since Python 2.2, at least) and are not used for this role in best practice code today. Rather, lambdas simply defer the call to the actual handler and provide extra handler arguments. Variables from the enclosing scope used by the lambda are *automatically* retained, even after the enclosing function exits.

The prior code listing, for example, can today normally be coded as we did earlier—name `X` in the handler will be automatically mapped to `X` in the enclosing scope, and so effectively remember what `X` was when the button was made:

```
def makegui():
    X = 42                                     # X is retained auto
    Button(text='ni', command=(lambda: handler(X, 'spam')))  # no need for defaults
```

We'll see this technique put to more concrete use later. When using classes to build your GUI, for instance, the `self` argument is a local variable in methods, and is thus automatically available in the bodies of lambda functions. There is no need to pass it in explicitly with defaults:

```
class Gui:
    def handler(self, A, B):
        ...use self, A and B...
    def makegui(self):
        X = 42
        Button(text='ni', command=(lambda: self.handler(X, 'spam')))

Gui().makegui()
mainloop()
```

When using classes, though, instance attributes can provide extra state for use in callback handlers, and so provide an alternative to extra call arguments. We'll see how in a moment. First, though, we need to take a quick non-GUI diversion into a dark corner of Python's scope rules to understand why default arguments are still sometimes necessary to pass values into nested lambda functions, especially in GUIs.

But you must still sometimes use defaults instead of enclosing scopes

Although you may still see defaults used to pass in enclosing scope references in some older Python code, automatic enclosing scope references are generally preferred today. In fact, it seems as though the newer nested scope lookup rules in Python automate

and replace the previously manual task of passing in enclosing scope values with defaults altogether.

Well, almost. There is a catch. It turns out that within a lambda (or `def`), references to names in the enclosing scope are actually resolved when the generated function is *called*, not when it is created. Because of this, when the function is later called, such name references will reflect the latest or final assignments made to the names anywhere in the enclosing scope, which are not necessarily the values they held when the function was made. This holds true even when the callback function is nested only in a module’s global scope, not in an enclosing function; in either case, all enclosing scope references are resolved at function call time, not at function creation time.

This is subtly different from default argument values, which are evaluated once when the function is *created*, not when it is later called. Because of that, defaults can still be useful for remembering the values of enclosing scope variables as they were when you made the function. Unlike enclosing scope name references, defaults will not have a different value if the variable later changes in the enclosing scope, between function creation and call. (In fact, this is why mutable defaults like lists retain their state between calls—they are created only once, when the function is made, and attached to the function itself.)

This is normally a nonissue, because most enclosing scope references name a variable that is assigned just once in the enclosing scope (the `self` argument in class methods, for example). But this can lead to coding mistakes if not understood, especially if you create functions within a *loop*; if those functions reference the loop variable, it will evaluate to the value it was given on the *last* loop iteration in *all* the functions generated. By contrast, if you use defaults instead, each function will remember the *current* value of the loop variable, not the last.

Because of this difference, nested scope references are not always sufficient to remember enclosing scope values, and defaults are sometimes still required today. Let’s see what this means in terms of code. Consider the following nested function (this section’s code snippets are saved in file `defaults.py` in the examples package, if you want to experiment with them).

```
def simple():
    spam = 'ni'
    def action():
        print(spam)           # name maps to enclosing function
    return action

act = simple()             # make and return nested function
act()                     # then call it: prints 'ni'
```

This is the simple case for enclosing scope references, and it works the same way whether the nested function is generated with a `def` or a lambda. But notice that this still works if we assign the enclosing scope’s `spam` variable *after* the nested function is created:

```

def normal():
    def action():
        return spam          # really, looked up when used
    spam = 'ni'
    return action

act = normal()
print(act())           # also prints 'ni'

```

As this implies, the enclosing scope name isn't resolved when the nested function is made—in fact, the name hasn't even been assigned yet in this example. The name is resolved when the nested function is *called*. The same holds true for lambdas:

```

def weird():
    spam = 42
    return (lambda: spam * 2)      # remembers spam in enclosing scope

act = weird()
print(act())                 # prints 84

```

So far, so good. The `spam` inside this nested lambda function remembers the value that this variable had in the enclosing scope, even after the enclosing scope exits. This pattern corresponds to a registered GUI callback handler run later on events. But once again, the nested scope reference really isn't being resolved when the lambda is run to create the function; it's being resolved when the generated function is later *called*. To make that more apparent, look at this code:

```

def weird():
    tmp = (lambda: spam * 2)      # remembers spam
    spam = 42                     # even though not set till here
    return tmp

act = weird()
print(act())                   # prints 84

```

Here again, the nested function refers to a variable that hasn't even been assigned yet when that function is made. Really, enclosing scope references yield the latest setting made in the enclosing scope, whenever the function is called. Watch what happens in the following code:

```

def weird():
    spam = 42
    handler = (lambda: spam * 2)    # func doesn't save 42 now
    spam = 50
    print(handler())              # prints 100: spam looked up now
    spam = 60
    print(handler())              # prints 120: spam looked up again now

weird()

```

Now, the reference to `spam` inside the lambda is different each time the generated function is called! In fact, it refers to what the variable was set to *last* in the enclosing scope at the time the nested function is called, because it is resolved at function call time, not at function creation time.

In terms of GUIs, this becomes significant most often when you generate callback handlers within loops and try to use enclosing scope references to remember extra data created within the loops. If you’re going to make functions within a loop, you have to apply the last example’s behavior to the loop variable:

```
def odd():
    funcs = []
    for c in 'abcdefg':
        funcs.append((lambda: c))      # c will be looked up later
    return funcs                      # does not remember current c

for func in odd():
    print(func(), end=' ')          # OOPS: print 7 g's, not a,b,c,... !
```

Here, the `func` list simulates registered GUI callback handlers associated with widgets. This doesn’t work the way most people expect it to. The variable `c` within the nested function will always be `g` here, the value that the variable was set to on the final iteration of the loop in the enclosing scope. The net effect is that all seven generated lambda functions wind up with the same extra state information when they are later called.

Analogous GUI code that adds information to lambda callback handlers will have similar problems—all buttons created in a loop, for instance, may wind up doing the same thing when clicked! To make this work, we still have to pass values into the nested function with defaults in order to save the current value of the loop variable (not its future value):

```
def odd():
    funcs = []
    for c in 'abcdefg':
        funcs.append((lambda c=c: c))  # force to remember c now
    return funcs                      # defaults eval now

for func in odd():
    print(func(), end=' ')          # OK: now prints a,b,c,...
```

This works now only because the default, unlike an external scope reference, is evaluated at function *creation* time, not at function call time. It remembers the value that a name in the enclosing scope had when the function was made, not the last assignment made to that name anywhere in the enclosing scope. The same is true even if the function’s enclosing scope is a module, not another function; if we don’t use the default argument in the following code, the loop variable will resolve to the same value in all seven functions:

```
funcs = []                                # enclosing scope is module
for c in 'abcdefg':                         # force to remember c now
    funcs.append((lambda c=c: c))            # else prints 7 g's again

for func in funcs:
    print(func(), end=' ')                  # OK: prints a,b,c,...
```

The moral of this story is that enclosing scope name references are a replacement for passing values in with defaults, *but only* as long as the name in the enclosing scope will

not change to a value you don’t expect after the nested function is created. You cannot generally reference enclosing scope loop variables within a nested function, for example, because they will change as the loop progresses. In most other cases, though, enclosing scope variables will take on only one value in their scope and so can be used freely.

We’ll see this phenomenon at work in later examples that construct larger GUIs. For now, remember that enclosing scopes are not a complete replacement for defaults; defaults are still required in some contexts to pass values into callback functions. Also keep in mind that classes are often a better and simpler way to retain extra state for use in callback handlers than are nested functions. Because state is explicit in classes, these scope issues do not apply. The next two sections cover this in detail.

Bound Method Callback Handlers

Let’s get back to coding GUIs. Although functions and lambdas suffice in many cases, bound methods of class instances work particularly well as callback handlers in GUIs—they record both an instance to send the event to and an associated method to call. For instance, [Example 7-14](#) shows Examples 7-12 and 7-13 rewritten to register a bound class method rather than a function or lambda result.

Example 7-14. PP4E\Gui\Intro\gui3c.py

```
import sys
from tkinter import *

class HelloClass:
    def __init__(self):
        widget = Button(None, text='Hello event world', command=self.quit)
        widget.pack()

    def quit(self):
        print('Hello class method world')  # self.quit is a bound method
        sys.exit()                         # retains the self+quit pair

HelloClass()
mainloop()
```

On a button press, tkinter calls this class’s `quit` method with no arguments, as usual. But really, it does receive one argument—the original `self` object—even though tkinter doesn’t pass it explicitly. Because the `self.quit` bound method retains both `self` and `quit`, it’s compatible with a simple function call; Python automatically passes the `self` argument along to the method function. Conversely, registering an unbound instance method that expects an argument, such as `HelloClass.quit`, won’t work, because there is no `self` object to pass along when the event later occurs.

Later, we'll see that class callback handler coding schemes provide a natural place to remember information for use on events—simply assign the information to `self` instance attributes:

```
class someGuiClass:  
    def __init__(self):  
        self.X = 42  
        self.Y = 'spam'  
        Button(text='Hi', command=self.handler)  
    def handler(self):  
        ...use self.X, self.Y...
```

Because the event will be dispatched to this class's method with a reference to the original instance object, `self` gives access to attributes that retain original data. In effect, the instance's attributes retain state information to be used when events occur. Especially in larger GUIs, this is a much more flexible technique than global variables or extra arguments added by lambdas.

Callable Class Object Callback Handlers

Because Python class instance objects can also be called if they inherit a `__call__` method to intercept the operation, we can pass one of these to serve as a callback handler too. [Example 7-15](#) shows a class that provides the required function-like interface.

Example 7-15. PP4E\Gui\Intro\gui3d.py

```
import sys  
from tkinter import *  
  
class HelloCallable:  
    def __init__(self):                      # __init__ run on object creation  
        self.msg = 'Hello __call__ world'  
  
    def __call__(self):                      # __call__ run later when called  
        print(self.msg)                      # class object looks like a function  
        sys.exit()  
  
widget = Button(None, text='Hello event world', command=HelloCallable())  
widget.pack()  
widget.mainloop()
```

Here, the `HelloCallable` instance registered with `command` can be called like a normal function; Python invokes its `__call__` method to handle the call operation made in `tkinter` on the button press. In effect, the general `__call__` method replaces a specific bound method in this case. Notice how `self.msg` is used to retain information for use on events here; `self` is the original instance when the special `__call__` method is automatically invoked.

All four `gui3` variants create the same sort of GUI window ([Figure 7-11](#)), but print different messages to `stdout` when their button is pressed:

```
C:\...\PP4E\Gui\Intro> python gui3.py  
Hello, I must be going...
```

```
C:\...\PP4E\Gui\Intro> python gui3b.py  
Hello lambda world
```

```
C:\...\PP4E\Gui\Intro> python gui3c.py  
Hello class method world
```

```
C:\...\PP4E\Gui\Intro> python gui3d.py  
Hello __call__ world
```

There are good reasons for each callback coding scheme (function, lambda, class method, callable class), but we need to move on to larger examples in order to uncover them in less theoretical terms.

Other tkinter Callback Protocols

For future reference, also keep in mind that using `command` options to intercept user-generated button press events is just one way to register callbacks in tkinter. In fact, there are a variety of ways for tkinter scripts to catch events:

Button command options

As we've just seen, button press events are intercepted by providing a callable object in widget `command` options. This is true of other kinds of button-like widgets we'll meet in [Chapter 8](#) (e.g., radio and check buttons and scales).

Menu command options

In the upcoming tkinter tour chapters, we'll also find that a `command` option is used to specify callback handlers for menu selections.

Scroll bar protocols

Scroll bar widgets register handlers with `command` options, too, but they have a unique event protocol that allows them to be cross-linked with the widget they are meant to scroll (e.g., listboxes, text displays, and canvases): moving the scroll bar automatically moves the widget, and vice versa.

General widget bind methods

A more general tkinter event `bind` method mechanism can be used to register callback handlers for lower-level interface events—key presses, mouse movement and clicks, and so on. Unlike `command` callbacks, `bind` callbacks receive an event object argument (an instance of the tkinter `Event` class) that gives context about the event—subject widget, screen coordinates, and so on.

Window manager protocols

In addition, scripts can also intercept window manager events (e.g., window close requests) by tapping into the window manager `protocol` method mechanism available on top-level window objects. Setting a handler for `WM_DELETE_WINDOW`, for instance, takes over window close buttons.

Scheduled event callbacks

Finally, tkinter scripts can also register callback handlers to be run in special contexts, such as timer expirations, input data arrival, and event-loop idle states. Scripts can also pause for state-change events related to windows and special variables. We'll meet these event interfaces in more detail near the end of [Chapter 9](#).

Binding Events

Of all the options listed in the prior section, `bind` is the most general, but also perhaps the most complex. We'll study it in more detail later, but to let you sample its flavor now, [Example 7-16](#) rewrites the prior section's GUI again to use `bind`, not the `command` keyword, to catch button presses.

Example 7-16. PP4E\Gui\Intro\gui3e.py

```
import sys
from tkinter import *

def hello(event):
    print('Press twice to exit')           # on single-left click

def quit(event):                         # on double-left click
    print('Hello, I must be going...')     # event gives widget, x/y, etc.
    sys.exit()

widget = Button(None, text='Hello event world')
widget.pack()
widget.bind('<Button-1>', hello)        # bind left mouse clicks
widget.bind('<Double-1>', quit)          # bind double-left clicks
widget.mainloop()
```

In fact, this version doesn't specify a `command` option for the button at all. Instead, it binds lower-level callback handlers for both left mouse clicks (`<Button-1>`) and double-left mouse clicks (`<Double-1>`) within the button's display area. The `bind` method accepts a large set of such event identifiers in a variety of formats, which we'll meet in [Chapter 8](#).

When run, this script makes the same window as before (see [Figure 7-11](#)). Clicking on the button once prints a message but doesn't exit; you need to double-click on the button now to exit as before. Here is the output after clicking twice and double-clicking once (a double-click fires the single-click callback first):

```
C:\...\PP4E\Gui\Intro> python gui3e.py
Press twice to exit
Press twice to exit
Press twice to exit
Hello, I must be going...
```

Although this script intercepts button clicks manually, the end result is roughly the same; widget-specific protocols such as button `command` options are really just higher-level interfaces to events you can also catch with `bind`.

We'll meet `bind` and all of the other tkinter event callback handler hooks again in more detail later in this book. First, though, let's focus on building GUIs that are larger than a single button and explore a few other ways to use classes in GUI work.

Adding Multiple Widgets

It's time to start building user interfaces with more than one widget. [Example 7-17](#) makes the window shown in [Figure 7-12](#).

Example 7-17. PP4E\Gui\Intro\gui4.py

```
from tkinter import *

def greeting():
    print('Hello stdout world!...')

win = Frame()
win.pack()
Label(win, text='Hello container world').pack(side=TOP)
Button(win, text='Hello', command=greeting).pack(side=LEFT)
Button(win, text='Quit', command=win.quit).pack(side=RIGHT)

win.mainloop()
```

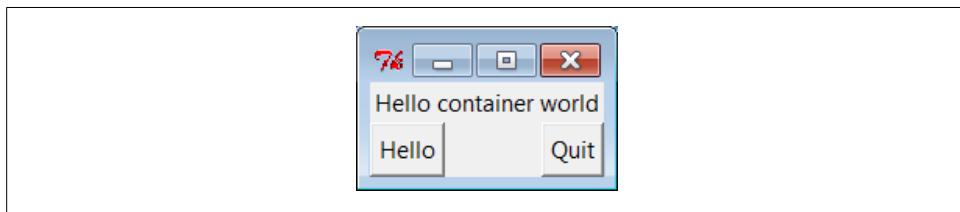


Figure 7-12. A multiple-widget window

This example makes a `Frame` widget (another tkinter class) and attaches three other widget objects to it, a `Label` and two `Buttons`, by passing the `Frame` as their first argument. In tkinter terms, we say that the `Frame` becomes a parent to the other three widgets. Both buttons on this display trigger callbacks:

- Pressing the Hello button triggers the `greeting` function defined within this file, which prints to `stdout` again.
- Pressing the Quit button calls the standard tkinter `quit` method, inherited by `win` from the `Frame` class (`Frame.quit` has the same effect as the `Tk.quit` we used earlier).

Here is the `stdout` text that shows up on Hello button presses, wherever this script's standard streams may be:

```
C:\...\PP4E\Gui\Intro> python gui4.py
Hello stdout world!...
Hello stdout world!...
```

```
Hello stdout world!...
Hello stdout world!...
```

The notion of attaching widgets to containers turns out to be at the core of layouts in tkinter. Before we go into more detail on that topic, though, let's get small.

Widget Resizing Revisited: Clipping

Earlier, we saw how to make widgets expand along with their parent window, by passing `expand` and `fill` options to the `pack` geometry manager. Now that we have a window with more than one widget, I can let you in on one of the more useful secrets in the packer. As a rule, widgets packed first are clipped last when a window is shrunk. That is, the order in which you pack items determines which items will be cut out of the display if it is made too small. Widgets packed later are cut out first. For example, [Figure 7-13](#) shows what happens when the `gui4` window is shrunk interactively.



Figure 7-13. `gui4` gets small

Try reordering the label and button lines in the script and see what happens when the window shrinks; the first one packed is always the last to go away. For instance, if the label is packed last, [Figure 7-14](#) shows that it is clipped first, even though it is attached to the top: `side` attachments and packing order both impact the overall layout, but only packing order matters when windows shrink. Here are the changed lines:

```
Button(win, text='Hello', command=greeting).pack(side=LEFT)
Button(win, text='Quit', command=win.quit).pack(side=RIGHT)
Label(win, text='Hello container world').pack(side=TOP)
```

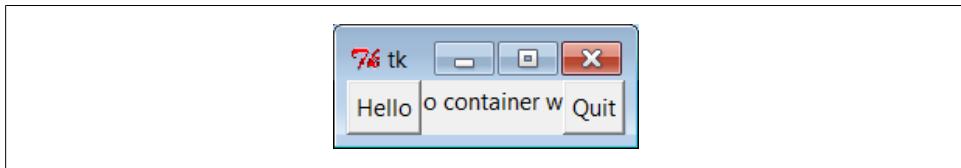


Figure 7-14. Label packed last, clipped first

tkinter keeps track of the packing order internally to make this work. Scripts can plan ahead for shrinkage by calling `pack` methods of more important widgets first. For instance, on the upcoming tkinter tour, we'll meet code that builds menus and toolbars at the top and bottom of the window; to make sure these are lost last as a window is shrunk, they are packed first, before the application components in the middle.

Similarly, displays that include scroll bars normally pack them before the items they scroll (e.g., text, lists) so that the scroll bars remain as the window shrinks.

Attaching Widgets to Frames

In larger terms, the critical innovation in this example is its use of frames: `Frame` widgets are just containers for other widgets, and so give rise to the notion of GUIs as widget hierarchies, or trees. Here, `win` serves as an enclosing window for the other three widgets. In general, though, by attaching widgets to frames, and frames to other frames, we can build up arbitrary GUI layouts. Simply divide the user interface into a set of increasingly smaller rectangles, implement each as a tkinter `Frame`, and attach basic widgets to the frame in the desired screen position.

In this script, when you specify `win` in the first argument to the `Label` and `Button` constructors, tkinter attaches them to the `Frame` (they become children of the `win` parent). `win` itself is attached to the default top-level window, since we didn't pass a parent to the `Frame` constructor. When we ask `win` to run itself (by calling `mainloop`), tkinter draws all the widgets in the tree we've built.

The three child widgets also provide `pack` options now: the `side` arguments tell which part of the containing frame (i.e., `win`) to attach the new widget to. The label hooks onto the top, and the buttons attach to the sides. `TOP`, `LEFT`, and `RIGHT` are all preassigned string variables imported from tkinter. Arranging widgets is a bit subtler than simply giving a `side`, though, but we need to take a quick detour into packer geometry management details to see why.

Layout: Packing Order and Side Attachments

When a widget tree is displayed, child widgets appear inside their parents and are arranged according to their order of packing and their packing options. Because of this, the order in which widgets are packed not only gives their clipping order, but also determines how their `side` settings play out in the generated display.

Here's how the packer's layout system works:

1. The packer starts out with an available space cavity that includes the entire parent container (e.g., the whole `Frame` or top-level window).
2. As each widget is packed on a side, that widget is given the entire requested side in the remaining space cavity, and the space cavity is shrunk.
3. Later pack requests are given an entire side of what is left, after earlier pack requests have shrunk the cavity.
4. After widgets are given cavity space, `expand` divides any space left, and `fill` and `anchor` stretch and position widgets within their assigned space.

For instance, if you recode the `gui4` child widget creation logic like this:

```
Button(win, text='Hello', command=greeting).pack(side=LEFT)
Label(win, text='Hello container world').pack(side=TOP)
Button(win, text='Quit', command=win.quit).pack(side=RIGHT)
```

you will wind up with the very different display shown in [Figure 7-15](#), even though you've moved the label code only one line down in the source file (contrast with [Figure 7-12](#)).

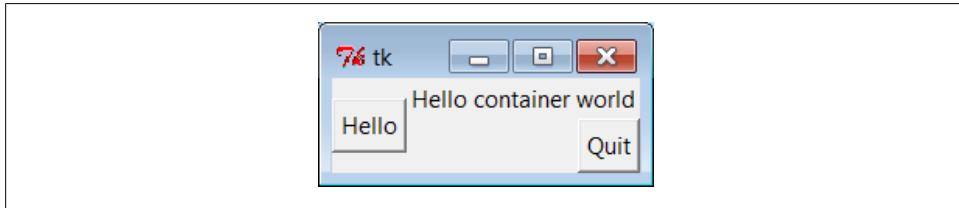


Figure 7-15. Packing the label second

Despite its `side` setting, the label does not get the entire top of the window now, and you have to think in terms of *shrinking cavities* to understand why. Because the Hello button is packed first, it is given the entire LEFT side of the Frame. Next, the label is given the entire TOP side of what is left. Finally, the Quit button gets the RIGHT side of the remainder—a rectangle to the right of the Hello button and under the label. When this window shrinks, widgets are clipped in reverse order of their packing: the Quit button disappears first, followed by the label.[†]

In the original version of this example ([Figure 7-12](#)), the label spans the entire top side just because it is the first one packed, not because of its `side` option. In fact, if you look at [Figure 7-14](#) closely, you'll see that it illustrates the same point—the label appeared between the buttons, because they had already carved off the entire left and right sides.

The Packer's Expand and Fill Revisited

Beyond the effects of packing order, the `fill` option we met earlier can be used to stretch the widget to occupy all the space in the cavity side it has been given, and any cavity space left after all packing is evenly allocated among widgets with the `expand=YES` we saw before. For example, coding this way creates the window in [Figure 7-16](#) (compare this to [Figure 7-15](#)):

```
Button(win, text='Hello', command=greeting).pack(side=LEFT, fill=Y)
Label(win, text='Hello container world').pack(side=TOP)
Button(win, text='Quit', command=win.quit).pack(side=RIGHT, expand=YES, fill=X)
```

[†] Technically, the packing steps are just rerun again after a window resize. But since this means that there won't be enough space left for widgets packed last when the window shrinks, it is as if widgets packed first are clipped last.

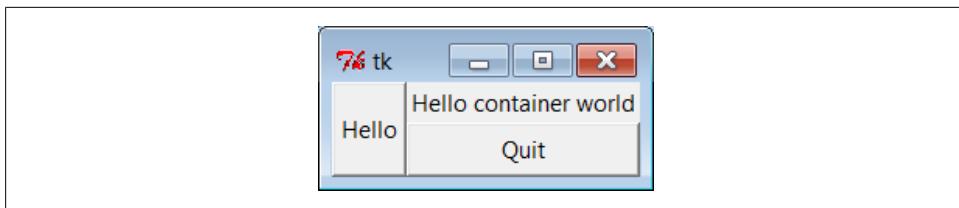


Figure 7-16. Packing with expand and fill options

To make all of these grow along with their window, though, we also need to make the container frame expandable; widgets expand beyond their initial packer arrangement only if *all of their parents expand, too*. Here are the changes in *gui4.py*:

```
win = Frame()  
win.pack(side=TOP, expand=YES, fill=BOTH)  
Button(win, text='Hello', command=greeting).pack(side=LEFT, fill=Y)  
Label(win, text='Hello container world').pack(side=TOP)  
Button(win, text='Quit', command=win.quit).pack(side=RIGHT, expand=YES, fill=X)
```

When this code runs, the `Frame` is assigned the entire top side of its parent as before (that is, the top parcel of the root window); but because it is now marked to expand into unused space in its parent and to fill that space both ways, it and all of its attached children expand along with the window. [Figure 7-17](#) shows how.

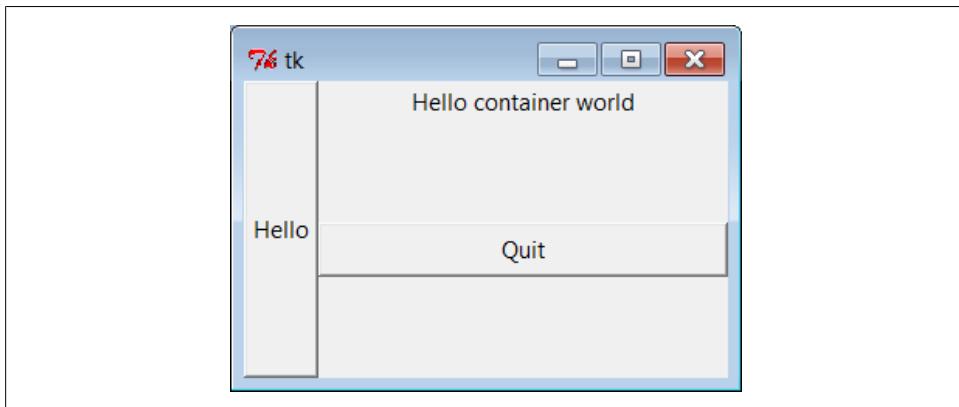


Figure 7-17. *gui4* gets big with an expandable frame

Using Anchor to Position Instead of Stretch

And as if that isn't flexible enough, the packer also allows widgets to be positioned within their allocated space with an `anchor` option, instead of filling that space with a `fill`. The `anchor` option accepts Tkinter constants identifying all eight points of the compass (N, NE, NW, S, etc.) and CENTER as its value (e.g., `anchor=NW`). It instructs the packer to position the widget at the desired position within its allocated space, if the space allocated for the widget is larger than the space needed to display the widget.

The default anchor is `CENTER`, so widgets show up in the middle of their space (the cavity side they were given) unless they are positioned with `anchor` or stretched with `fill`. To demonstrate, change `gui4` to use this sort of code:

```
Button(win, text='Hello', command=greeting).pack(side=LEFT, anchor=N)
Label(win, text='Hello container world').pack(side=TOP)
Button(win, text='Quit', command=win.quit).pack(side=RIGHT)
```

The only thing new here is that the Hello button is anchored to the north side of its space allocation. Because this button was packed first, it got the entire left side of the parent frame. This is more space than is needed to show the button, so it shows up in the middle of that side by default, as in [Figure 7-15](#) (i.e., anchored to the center). Setting the anchor to `N` moves it to the top of its side, as shown in [Figure 7-18](#).

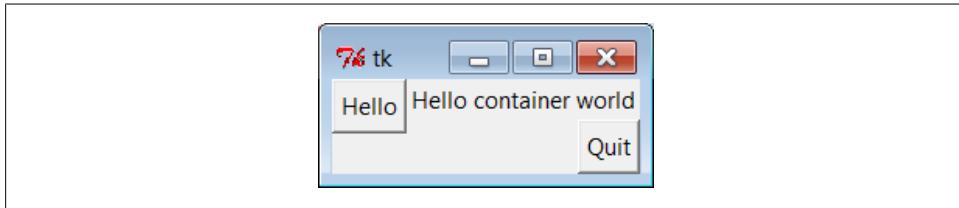


Figure 7-18. Anchoring a button to the north

Keep in mind that `fill` and `anchor` are applied after a widget has been allocated cavity side space by its `side`, packing order, and `expand` extra space request. By playing with packing orders, sides, fills, and anchors, you can generate lots of layout and clipping effects, and you should take a few moments to experiment with alternatives if you haven't already. In the original version of this example, for instance, the label spans the entire top side just because it is the first packed.

As we'll see later, frames can be nested in other frames, too, in order to make more complex layouts. In fact, because each parent container is a distinct space cavity, this provides a sort of escape mechanism for the packer cavity algorithm: to better control where a set of widgets show up, simply pack them within a nested subframe and attach the frame as a package to a larger container. A row of push buttons, for example, might be easier laid out in a frame of its own than if mixed with other widgets in the display directly.

Finally, also keep in mind that the widget tree created by these examples is really an implicit one; tkinter internally records the relationships implied by passed parent widget arguments. In OOP terms, this is a *composition* relationship—the `Frame` contains a `Label` and `Buttons`. Let's look at *inheritance* relationships next.

Customizing Widgets with Classes

You don't have to use OOP in tkinter scripts, but it can definitely help. As we just saw, tkinter GUIs are built up as class-instance object trees. Here's another way Python's

OOP features can be applied to GUI models: specializing widgets by inheritance. Example 7-18 builds the window in Figure 7-19.

Example 7-18. PP4E\Gui\Intro\gui5.py

```
from tkinter import *

class HelloButton(Button):
    def __init__(self, parent=None, **config):
        Button.__init__(self, parent, **config)
        self.pack()
        self.config(command=self.callback)

    def callback(self):
        print('Goodbye world...')
        # replace in subclasses
        self.quit()

if __name__ == '__main__':
    HelloButton(text='Hello subclass world').mainloop()
```

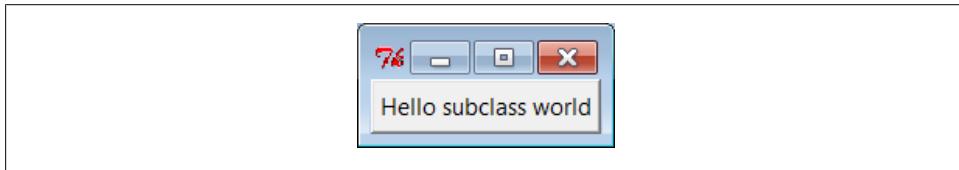


Figure 7-19. A button subclass in action

This example isn't anything special to look at: it just displays a single button that, when pressed, prints a message and exits. But this time, it is a button widget we created on our own. The `HelloButton` class inherits everything from the `tkinter Button` class, but adds a `callback` method and constructor logic to set the `command` option to `self.callback`, a bound method of the instance. When the button is pressed this time, the new widget class's `callback` method, not a simple function, is invoked.

The `**config` argument here is assigned unmatched keyword arguments in a dictionary, so they can be passed along to the `Button` constructor. The `**config` in the `Button` constructor call unpacks the dictionary back into keyword arguments (it's actually optional here, because of the old-style dictionary widget call form we met earlier, but doesn't hurt). We met the `config` widget method called in `HelloButton`'s constructor earlier; it is just an alternative way to pass configuration options after the fact (instead of passing constructor arguments).

Standardizing Behavior and Appearance

So what's the point of subclassing widgets like this? In short, it allows sets of widgets made from the customized classes to look and act the same. When coded well, we get both "for free" from Python's OOP model. This can be a powerful technique in larger programs.

Common behavior

[Example 7-18](#) standardizes behavior—it allows widgets to be configured by subclassing instead of by passing in options. In fact, its `HelloButton` is a true button; we can pass in configuration options such as its `text` as usual when one is made. But we can also specify callback handlers by overriding the `callback` method in subclasses, as shown in [Example 7-19](#).

Example 7-19. PP4E\Gui\Intro\gui5b.py

```
from gui5 import HelloButton

class MyButton(HelloButton):      # subclass HelloButton
    def callback(self):          # redefine press-handler method
        print("Ignoring press!...")

if __name__ == '__main__':
    MyButton(None, text='Hello subclass world').mainloop()
```

This script makes the same window; but instead of exiting, this `MyButton` button, when pressed, prints to `stdout` and stays up. Here is its standard output after being pressed a few times:

```
C:\...\PP4E\Gui\Intro> python gui5b.py
Ignoring press!...
Ignoring press!...
Ignoring press!...
Ignoring press!...
```

Whether it's simpler to customize widgets by subclassing or passing in options is probably a matter of taste in this simple example. But the larger point to notice is that Tk becomes truly object oriented in Python, just because Python is object oriented—we can specialize widget classes using normal class-based and object-oriented techniques. In fact this applies to both widget behavior and appearance.

Common appearance

For example, although we won't study widget configuration options until the next chapter, a similar customized button class could provide a standard look-and-feel different from tkinter's defaults for every instance created from it, and approach the notions of "styles" or "themes" in some GUI toolkits:

```
class ThemedButton(Button):                      # config my style too
    def __init__(self, parent=None, **configs):    # used for each instance
        Button.__init__(self, parent, **configs)     # see chapter 8 for options
        self.pack()
        self.config(fg='red', bg='black', font=('courier', 12), relief=RAISED, bd=5)

B1 = ThemedButton(text='spam', command=onSpam)   # normal button widget objects
B2 = ThemedButton(text='eggs')                   # but same appearance by inheritance
B2.pack(expand=YES, fill=BOTH)
```

This code is something of a preview; see file *gui5b-themed.py* in the examples package for a complete version, and watch for more on its widget configuration options in [Chapter 8](#). But it illustrates the application of common appearance by subclassing widgets directly—every button created from its class looks the same, and will pick up any future changes in its configurations automatically.

Widget subclasses are a programmer’s tool, of course, but we can also make such configurations accessible to a GUI’s users. In larger programs later in the book (e.g., PyEdit, PyClock, and PyMailGUI), we’ll sometimes achieve a similar effect by importing configurations from modules and applying them to widgets as they are built. If such external settings are used by a customized widget subclass like our `ThemedButton` above, they will again apply to all its instances and subclasses (for reference, the full version of the following code is in file *gui5b-themed-user.py*):

```
from user_preferences import bcolor, bfont, bsize # get user settings

class ThemedButton(Button):
    def __init__(self, parent=None, **configs):
        Button.__init__(self, parent, **configs)
        self.pack()
        self.config(bg=bcolor, font=(bfont, bsize))

ThemedButton(text='spam', command=onSpam) # normal button widget objects
ThemedButton(text='eggs', command=onEggs) # all inherit user preferences

class MyButton(ThemedButton): # subclasses inherit prefs too
    def __init__(self, parent=None, **configs):
        ThemedButton.__init__(self, parent, **configs)
        self.config(text='subclass')

MyButton(command=onSpam)
```

Again, more on widget configuration in the next chapter; the big picture to take away here is that customizing widget classes with *subclasses* allows us to tailor both their behavior and their appearance for an entire set of widgets. The next example provides yet another way to arrange for specialization—as customizable and attachable widget packages, usually known as *components*.

Reusable GUI Components with Classes

Larger GUI interfaces are often built up as subclasses of `Frame`, with callback handlers implemented as methods. This structure gives us a natural place to store information between events: instance attributes record state. It also allows us to both specialize GUIs by overriding their methods in new subclasses and attach them to larger GUI structures to reuse them as general components. For instance, a GUI text editor implemented as a `Frame` subclass can be attached to and configured by any number of other GUIs; if done well, we can plug such a text editor into any user interface that needs text editing tools.

We'll meet such a text editor component in [Chapter 11](#). For now, [Example 7-20](#) illustrates the concept in a simple way. The script `gui6.py` produces the window in [Figure 7-20](#).

Example 7-20. PP4E\Gui\Intro\gui6.py

```
from tkinter import *

class Hello(Frame):
    def __init__(self, parent=None):                      # an extended Frame
        Frame.__init__(self, parent)                      # do superclass init
        self.pack()
        self.data = 42
        self.make_widgets()                             # attach widgets to self

    def make_widgets(self):
        widget = Button(self, text='Hello frame world!', command=self.message)
        widget.pack(side=LEFT)

    def message(self):
        self.data += 1
        print('Hello frame world %s!' % self.data)

if __name__ == '__main__': Hello().mainloop()
```

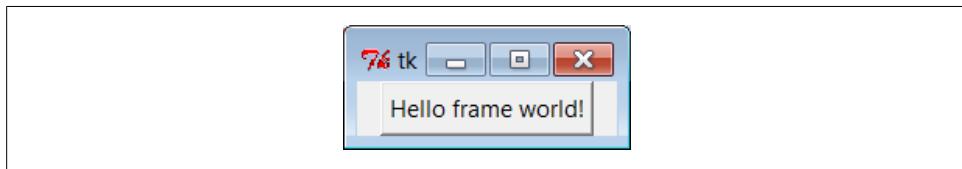


Figure 7-20. A custom Frame in action

This example pops up a single-button window. When pressed, the button triggers the `self.message` bound method to print to `stdout` again. Here is the output after pressing this button four times; notice how `self.data` (a simple counter here) retains its state between presses:

```
C:\...\PP4E\Gui\Intro> python gui6.py
Hello frame world 43!
Hello frame world 44!
Hello frame world 45!
Hello frame world 46!
```

This may seem like a roundabout way to show a `Button` (we did it in fewer lines in Examples [7-10](#), [7-11](#), and [7-12](#)). But the `Hello` class provides an enclosing organizational *structure* for building GUIs. In the examples prior to the last section, we made GUIs using a function-like approach: we called widget constructors as though they were functions and hooked widgets together manually by passing in parents to widget construction calls. There was no notion of an enclosing context, apart from the global

scope of the module file containing the widget calls. This works for simple GUIs but can make for brittle code when building up larger GUI structures.

But by subclassing `Frame` as we've done here, the class becomes an enclosing context for the GUI:

- Widgets are added by attaching objects to `self`, an instance of a `Frame` container subclass (e.g., `Button`).
- Callback handlers are registered as bound methods of `self`, and so are routed back to code in the class (e.g., `self.message`).
- State information is retained between events by assigning to attributes of `self`, visible to all callback methods in the class (e.g., `self.data`).
- It's easy to make multiple copies of such a GUI component, even within the same process, because each class instance is a distinct namespace.
- Classes naturally support customization by inheritance and by composition attachment.

In a sense, entire GUIs become specialized `Frame` objects with extensions for an application. Classes can also provide protocols for building widgets (e.g., the `make_widgets` method here), handle standard configuration chores (like setting window manager options), and so on. In short, `Frame` subclasses provide a simple way to organize collections of other widget-class objects.

Attaching Class Components

Perhaps more importantly, subclasses of `Frame` are true widgets: they can be further extended and customized by subclassing and can be attached to enclosing widgets. For instance, to attach the entire package of widgets that a class builds to something else, simply create an instance of the class with a real parent widget passed in. To illustrate, running the script in [Example 7-21](#) creates the window shown in [Figure 7-21](#).

Example 7-21. PP4E\Gui\Intro\gui6b.py

```
from sys import exit
from tkinter import *                      # get Tk widget classes
from gui6 import Hello                      # get the subframe class

parent = Frame(None)                       # make a container widget
parent.pack()                             # attach Hello instead of running it
Hello(parent).pack(side=RIGHT)

Button(parent, text='Attach', command=exit).pack(side=LEFT)
parent.mainloop()
```

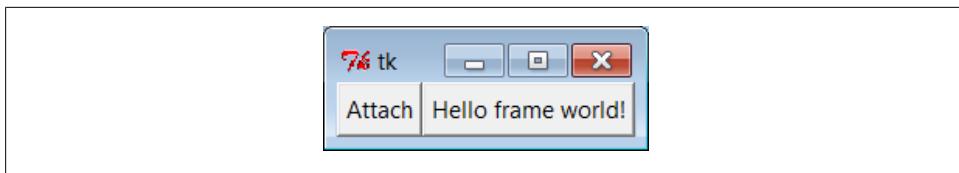


Figure 7-21. An attached class component on the right

This script just adds `Hello`'s button to the right side of `parent`—a container `Frame`. In fact, the button on the right in this window represents an embedded component: its button really represents an attached Python class object. Pressing the embedded class's button on the right prints a message as before; pressing the new button exits the GUI by a `sys.exit` call:

```
C:\...\PP4E\Gui\Intro> python gui6b.py
Hello frame world 43!
Hello frame world 44!
Hello frame world 45!
Hello frame world 46!
```

In more complex GUIs, we might instead attach large `Frame` subclasses to other container components and develop each independently. For instance, [Example 7-22](#) is yet another specialized `Frame` itself, but it attaches an instance of the original `Hello` class in a more object-oriented fashion. When run as a top-level program, it creates a window identical to the one shown in [Figure 7-21](#).

Example 7-22. PP4E\Gui\Intro\gui6c.py

```
from tkinter import *                      # get Tk widget classes
from gui6 import Hello                      # get the subframe class

class HelloContainer(Frame):
    def __init__(self, parent=None):
        Frame.__init__(self, parent)
        self.pack()
        self.makeWidgets()

    def makeWidgets(self):
        Hello(self).pack(side=RIGHT)      # attach a Hello to me
        Button(self, text='Attach', command=self.quit).pack(side=LEFT)

if __name__ == '__main__': HelloContainer().mainloop()
```

This looks and works exactly like `gui6b` but registers the added button's callback handler as `self.quit`, which is just the standard `quit` widget method this class inherits from `Frame`. The window this time represents two Python classes at work—the embedded component's widgets on the right (the original `Hello` button) and the container's widgets on the left.

Naturally, this is a simple example (we attached only a single button here, after all). But in more practical user interfaces, the set of widget class objects attached in this way

can be much larger. If you imagine replacing the `Hello` call in this script with a call to attach an already coded and fully debugged calculator object, you'll begin to better understand the power of this paradigm. If we code all of our GUI components as classes, they automatically become a library of reusable widgets, which we can combine in other applications as often as we like.

Extending Class Components

When GUIs are built with classes, there are a variety of ways to reuse their code in other displays. To extend `Hello` instead of attaching it, we just override some of its methods in a new subclass (which itself becomes a specialized `Frame` widget). This technique is shown in [Example 7-23](#).

Example 7-23. PP4E\Gui\Intro\gui6d.py

```
from tkinter import *
from gui6 import Hello

class HelloExtender(Hello):
    def make_widgets(self):
        Hello.make_widgets(self) # extend method here
        Button(self, text='Extend', command=self.quit).pack(side=RIGHT)

    def message(self):
        print('hello', self.data) # redefine method here

if __name__ == '__main__': HelloExtender().mainloop()
```

This subclass's `make_widgets` method here first builds the superclass's widgets and then adds a second Extend button on the right, as shown in [Figure 7-22](#).

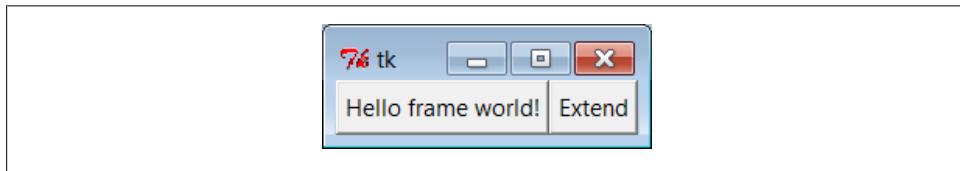


Figure 7-22. A customized class's widgets, on the left

Because it redefines the `message` method, pressing the original superclass's button on the left now prints a different string to `stdout` (when searching up from `self`, the `message` attribute is found first in this subclass, not in the superclass):

```
C:\...\PP4E\Gui\Intro> python gui6d.py
hello 42
hello 42
hello 42
hello 42
```

But pressing the new Extend button on the right, which is added by this subclass, exits immediately, since the `quit` method (inherited from `Hello`, which inherits it from

`Frame`) is the added button's callback handler. The net effect is that this class customizes the original to add a new button and change `message`'s behavior.

Although this example is simple, it demonstrates a technique that can be powerful in practice: to change a GUI's behavior, we can write a new class that customizes its parts rather than changing the existing GUI code in place. The main code need be debugged only once and can be customized with subclasses as unique needs arise.

The moral of this story is that tkinter GUIs can be coded without ever writing a single new class, but using classes to structure your GUI code makes it much more reusable in the long run. If done well, you can both attach already debugged components to new interfaces and specialize their behavior in new external subclasses as needed for custom requirements. Either way, the initial upfront investment to use classes is bound to save coding time in the end.

Standalone Container Classes

Before we move on, I want to point out that it's possible to reap most of the class-based component benefits previously mentioned by creating standalone classes not derived from tkinter `Frames` or other widgets. For instance, the class in [Example 7-24](#) generates the window shown in [Figure 7-23](#).

Example 7-24. PP4E\Gui\Intro\gui7.py

```
from tkinter import *

class HelloPackage:                      # not a widget subclass
    def __init__(self, parent=None):
        self.top = Frame(parent)           # embed a Frame
        self.top.pack()
        self.data = 0
        self.make_widgets()               # attach widgets to self.top

    def make_widgets(self):
        Button(self.top, text='Bye', command=self.top.quit).pack(side=LEFT)
        Button(self.top, text='Hye', command=self.message).pack(side=RIGHT)

    def message(self):
        self.data += 1
        print('Hello number', self.data)

if __name__ == '__main__': HelloPackage().top.mainloop()
```

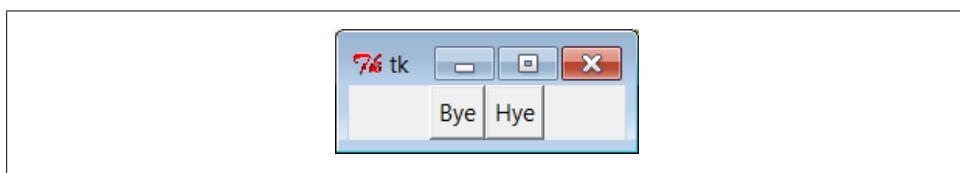


Figure 7-23. A standalone class package in action

When run, the Hye button here prints to `stdout` and the Bye button closes and exits the GUI, much as before:

```
C:\...\PP4E\Gui\Intro> python gui7.py
Hello number 1
Hello number 2
Hello number 3
Hello number 4
```

Also as before, `self.data` retains state between events, and callbacks are routed to the `self.message` method within this class. Unlike before, the `HelloPackage` class is not itself a kind of `Frame` widget. In fact, it's not a kind of anything—it serves only as a generator of namespaces for storing away real widget objects and state. Because of that, widgets are attached to a `self.top` (an embedded `Frame`), not to `self`. Moreover, all references to the object as a widget must descend to the embedded frame, as in the `top.mainloop` call to start the GUI at the end of the script.

This makes for a bit more coding within the class, but it avoids potential name clashes with both attributes added to `self` by the tkinter framework and existing tkinter widget methods. For instance, if you define a `config` method in your class, it will hide the `config` call exported by tkinter. With the standalone class package in this example, you get only the methods and instance attributes that your class defines.

In practice, tkinter doesn't use very many names, so this is not generally a big concern.[‡] It can happen, of course; but frankly, I've never seen a real tkinter name clash in widget subclasses in some 18 years of Python coding. Moreover, using standalone classes is not without other downsides. Although they can generally be attached and subclassed as before, they are not quite plug-and-play compatible with real widget objects. For instance, the configuration calls made in [Example 7-21](#) for the `Frame` subclass fail in [Example 7-25](#).

Example 7-25. PP4E\Gui\Intro\gui7b.py

```
from tkinter import *
from gui7 import HelloPackage      # or get from gui7c--__getattr__ added

frm = Frame()
frm.pack()
Label(frm, text='hello').pack()

part = HelloPackage(frm)
```

[‡] If you study the `tkinter` module's source code (today, mostly in file `__init__.py` in `Lib\tkinter`), you'll notice that many of the attribute names it creates start with a single underscore to make them unique from yours; others do not because they are potentially useful outside of the tkinter implementation (e.g., `self.master`, `self.children`). Curiously, at this writing most of tkinter still does not use the Python "pseudoprivate attributes" trick of prefixing attribute names with two leading underscores to automatically add the enclosing class's name and thus localize them to the creating class. If tkinter is ever rewritten to employ this feature, name clashes will be much less likely in widget subclasses. Most of the attributes of widget classes, though, are methods intended for use in client scripts; the single underscore names are accessible too, but are less likely to clash with most names of your own.

```
part.pack(side=RIGHT)           # FAILS!--need part.top.pack(side=RIGHT)
frm.mainloop()
```

This won't quite work, because `part` isn't really a widget. To treat it as such, you must descend to `part.top` before making GUI configurations and hope that the name `top` is never changed by the class's developer. In other words, it exposes some of the class's internals. The class could make this better by defining a method that always routes unknown attribute fetches to the embedded `Frame`, as in [Example 7-26](#).

Example 7-26. PP4E\Gui\Intro\gui7c.py

```
import gui7
from tkinter import *

class HelloPackage(gui7.HelloPackage):
    def __getattr__(self, name):
        return getattr(self.top, name)          # pass off to a real widget

if __name__ == '__main__': HelloPackage().mainloop()  # invokes __getattr__!
```

As is, this script simply creates [Figure 7-23](#) again; changing [Example 7-25](#) to import this extended `HelloPackage` from `gui7c`, though, produces the correctly-working window in [Figure 7-24](#).

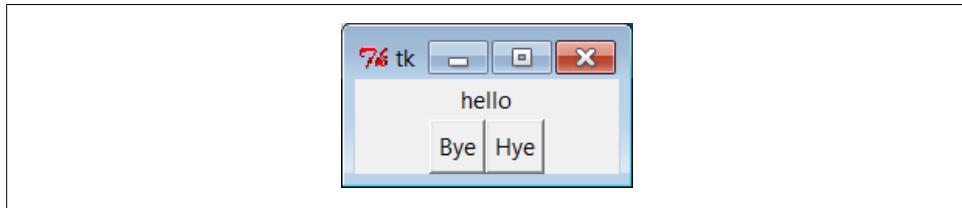


Figure 7-24. A standalone class package in action

Routing attribute fetches to nested widgets works this way, but that then requires even more extra coding in standalone package classes. As usual, though, the significance of all these trade-offs varies per application.

The End of the Tutorial

In this chapter, we learned the core concepts of Python/tkinter programming and met a handful of simple widget objects along the way—e.g., labels, buttons, frames, and the packer geometry manager. We've seen enough to construct simple interfaces, but we have really only scratched the surface of the tkinter widget set.

In the next two chapters, we will apply what we've learned here to study the rest of the tkinter library, and we'll learn how to use it to generate the kinds of interfaces you expect to see in realistic GUI programs. As a preview and roadmap, [Table 7-1](#) lists the kinds of widgets we'll meet there in roughly their order of appearance. Note that this

table lists only widget classes; along the way, we will also meet a few additional widget-related topics that don't appear in this table.

Table 7-1. tkinter widget classes

Widget class	Description
Label	A simple message area
Button	A simple labeled push-button widget
Frame	A container for attaching and arranging other widget objects
Toplevel, Tk	A new window managed by the window manager
Message	A multiline label
Entry	A simple single-line text-entry field
Checkbutton	A two-state button widget, typically used for multiple-choice selections
Radiobutton	A two-state button widget, typically used for single-choice selections
Scale	A slider widget with scalable positions
PhotoImage	An image object used for displaying full-color images on other widgets
BitmapImage	An image object used for displaying bitmap images on other widgets
Menu	A set of options associated with a Menubutton or top-level window
Menubutton	A button that opens a Menu of selectable options and submenus
Scrollbar	A control for scrolling other widgets (e.g., listbox, canvas, text)
Listbox	A list of selection names
Text	A multiline text browse/edit widget, with support for fonts, and so on
Canvas	A graphic drawing area, which supports lines, circles, photos, text, and so on

We've already met `Label`, `Button`, and `Frame` in this chapter's tutorial. To make the remaining topics easier to absorb, they are split over the next two chapters: [Chapter 8](#) covers the first widgets in this table up to but not including `Menu`, and [Chapter 9](#) presents widgets that are lower in this table.

Besides the widget classes in this table, there are additional classes and tools in the `tkinter` library, many of which we'll explore in the following two chapters as well:

Geometry management

`pack`, `grid`, `place`

tkinter linked variables

`StringVar`, `IntVar`, `DoubleVar`, `BooleanVar`

Advanced Tk widgets

`Spinbox`, `LabelFrame`, `PanedWindow`

Composite widgets

`Dialog`, `ScrolledText`, `OptionsMenu`

Scheduled callbacks

Widget `after`, `wait`, and `update` methods

Other tools

Standard dialogs, clipboard, `bind` and `Event`, widget configuration options, custom and modal dialogs, animation techniques

Most tkinter widgets are familiar user interface devices. Some are remarkably rich in functionality. For instance, the `Text` class implements a sophisticated multiline text widget that supports fonts, colors, and special effects and is powerful enough to implement a web browser's page display. The similarly feature-rich `Canvas` class provides extensive drawing tools powerful enough for visualization and other image-processing applications. Beyond this, tkinter extensions such as the `Pmw`, `Tix`, and `ttk` packages described at the start of this chapter add even richer widgets to a GUI programmer's toolbox.

Python/tkinter for Tcl/Tk Converts

At the start of this chapter, I mentioned that tkinter is Python's interface to the Tk GUI library, originally written for the Tcl language. To help readers migrating from Tcl to Python and to summarize some of the main topics we met in this chapter, this section contrasts Python's Tk interface with Tcl's. This mapping also helps make Tk references written for other languages more useful to Python developers.

In general terms, Tcl's command-string view of the world differs widely from Python's object-based approach to programming. In terms of Tk programming, though, the syntactic differences are fairly small. Here are some of the main distinctions in Python's tkinter interface:

Creation

Widgets are created as class instance objects by calling a widget class.

Masters (parents)

Parents are previously created objects that are passed to widget-class constructors.

Widget options

Options are constructor or `config` keyword arguments or indexed keys.

Operations

Widget operations (actions) become tkinter widget class object methods.

Callbacks

Callback handlers are any callable objects: function, method, `lambda`, and so on.

Extension

Widgets are extended using Python class inheritance mechanisms.

Composition

Interfaces are constructed by attaching objects, not by concatenating names.

Linked variables (next chapter)

Variables associated with widgets are tkinter class objects with methods.

In Python, widget creation commands (e.g., `button`) are Python class names that start with an uppercase letter (e.g., `Button`), two-word widget operations (e.g., `add command`) become a single method name with an underscore (e.g., `add_command`), and the “configure” method can be abbreviated as “config,” as in Tcl. In [Chapter 8](#), we will also see that tkinter “variables” associated with widgets take the form of class instance objects (e.g., `StringVar`, `IntVar`) with `get` and `set` methods, not simple Python or Tcl variable names. [Table 7-2](#) shows some of the primary language mappings in more concrete terms.

Table 7-2. Tk-to-tkinter mappings

Operation	Tcl/Tk	Python/tkinter
Creation	<code>Frame .panel</code>	<code>panel = Frame()</code>
Masters	<code>button .panel.quit</code>	<code>quit = Button(panel)</code>
Options	<code>button .panel.go -fg black</code>	<code>go = Button(panel, fg='black')</code>
Configure	<code>.panel.go config -bg red</code>	<code>go.config(bg='red') go['bg'] = 'red'</code>
Actions	<code>.popup invoke</code>	<code>popup.invoke()</code>
Packing	<code>pack .panel -side left -fill x</code>	<code>panel.pack(side=LEFT, fill=X)</code>

Some of these differences are more than just syntactic, of course. For instance, Python builds an internal widget object tree based on parent arguments passed to widget constructors, without ever requiring concatenated widget pathname strings. Once you’ve made a widget object, you can use it directly by object reference. Tcl coders can hide some dotted pathnames by manually storing them in variables, but that’s not quite the same as Python’s purely object-based model.

Once you’ve written a few Python/tkinter scripts, though, the coding distinctions in the Python object world will probably seem trivial. At the same time, Python’s support for object-oriented techniques adds an entirely new component to Tk development; you get the same widgets, plus Python’s support for code structure and reuse.

A tkinter Tour, Part 1

“Widgets and Gadgets and GUIs, Oh My!”

This chapter is a continuation of our look at GUI programming in Python. The previous chapter used simple widgets—buttons, labels, and the like—to demonstrate the fundamentals of Python/tkinter coding. That was simple by design: it’s easier to grasp the big GUI picture if widget interface details don’t get in the way. But now that we’ve seen the basics, this chapter and the next move on to present a tour of more advanced widget objects and tools available in the tkinter library.

As we’ll find, this is where GUI scripting starts getting both practical and fun. In these two chapters, we’ll meet classes that build the interface devices you expect to see in real programs—e.g., sliders, check buttons, menus, scrolled lists, dialogs, graphics, and so on. After these chapters, the last GUI chapter moves on to present larger GUIs that utilize the coding techniques and the interfaces shown in all prior GUI chapters. In these two chapters, though, examples are small and self-contained so that we can focus on widget details.

This Chapter’s Topics

Technically, we’ve already used a handful of simple widgets in [Chapter 7](#). So far we’ve met `Label`, `Button`, `Frame`, and `Tk`, and studied `pack` geometry management concepts along the way. Although all of these are basic, they represent tkinter interfaces in general and can be workhorses in typical GUIs. `Frame` containers, for instance, are the basis of hierarchical display layout.

In this and the following chapter, we’ll explore additional options for widgets we’ve already seen and move beyond the basics to cover the rest of the tkinter widget set. Here are some of the widgets and topics we’ll explore in this chapter:

- `Toplevel` and `Tk` widgets
- `Message` and `Entry` widgets
- `Checkbutton`, `Radiobutton`, and `Scale` widgets

- Images: `PhotoImage` and `BitmapImage` objects
- Widget and window configuration options
- Dialogs, both standard and custom
- Low-level event binding
- `tkinter` linked variable objects
- Using the Python Imaging Library (PIL) extension for other image types and operations

After this chapter, [Chapter 9](#) concludes the two-part tour by presenting the remainder of the `tkinter` library’s tool set: menus, text, canvases, animation, and more.

To make this tour interesting, I’ll also introduce a few notions of component reuse along the way. For instance, some later examples will be built using components written for prior examples. Although these two tour chapters introduce widget interfaces, this book is also about Python programming in general; as we’ll see, `tkinter` programming in Python can be much more than simply drawing circles and arrows.

Configuring Widget Appearance

So far, all the buttons and labels in examples have been rendered with a default look-and-feel that is standard for the underlying platform. With my machine’s color scheme, that usually means that they’re gray on Windows. `tkinter` widgets can be made to look arbitrarily different, though, using a handful of widget and packer options.

Because I generally can’t resist the temptation to customize widgets in examples, I want to cover this topic early on the tour. [Example 8-1](#) introduces some of the configuration options available in `tkinter`.

Example 8-1. PP4E\Gui\Tour\config-label.py

```
from tkinter import *
root = Tk()
labelfont = ('times', 20, 'bold')
widget = Label(root, text='Hello config world')
widget.config(bg='black', fg='yellow')           # family, size, style
                                                # yellow text on black label
widget.config(font=labelfont)                   # use a larger font
widget.config(height=3, width=20)                # initial size: lines,chars
widget.pack(expand=YES, fill=BOTH)
root.mainloop()
```

Remember, we can call a widget’s `config` method to reset its options at any time, instead of passing all of them to the object’s constructor. Here, we use it to set options that produce the window in [Figure 8-1](#).

This may not be completely obvious unless you run this script on a real computer (alas, I can’t show it in color here), but the label’s text shows up in yellow on a black



Figure 8-1. A custom label appearance

background, and with a font that's very different from what we've seen so far. In fact, this script customizes the label in a number of ways:

Color

By setting the `bg` option of the label widget here, its background is displayed in black; the `fg` option similarly changes the foreground (text) color of the widget to yellow. These color options work on most tkinter widgets and accept either a simple color name (e.g., '`blue`') or a hexadecimal string. Most of the color names you are familiar with are supported (unless you happen to work for Crayola). You can also pass a hexadecimal color identifier string to these options to be more specific; they start with a # and name a color by its red, green, and blue saturations, with an equal number of bits in the string for each. For instance, '`#ff0000`' specifies eight bits per color and defines pure red; "f" means four "1" bits in hexadecimal. We'll come back to this hex form when we meet the color selection dialog later in this chapter.

Size

The label is given a preset size in lines high and characters wide by setting its `height` and `width` attributes. You can use this setting to make the widget larger than the tkinter geometry manager would by default.

Font

This script specifies a custom font for the label's text by setting the label's `font` attribute to a three-item tuple giving the font family, size, and style (here: `Times`, 20-point, and bold). Font style can be `normal`, `bold`, `roman`, `italic`, `underline`, `overstrike`, or combinations of these (e.g., "bold italic"). tkinter guarantees that `Times`, `Courier`, and `Helvetica` font family names exist on all platforms, but others may work, too (e.g., `system` gives the system font on Windows). Font settings like this work on all widgets with text, such as labels, buttons, entry fields, listboxes, and `Text` (the latter of which can even display more than one font at once with "tags"). The `font` option still accepts older X-Windows-style font indicators—long strings with dashes and stars—but the newer tuple font indicator form is more platform independent.

Layout and expansion

Finally, the label is made generally expandable and stretched by setting the `pack expand` and `fill` options we met in the last chapter; the label grows as the window does. If you maximize this window, its black background fills the whole screen and the yellow message is centered in the middle; try it.

In this script, the net effect of all these settings is that this label looks radically different from the ones we've been making so far. It no longer follows the Windows standard look-and-feel, but such conformance isn't always important. For reference, `tkinter` provides additional ways to customize appearance that are not used by this script, but which may appear in others:

Border and relief

A `bd= N` widget option can be used to set border width, and a `relief= S` option can specify a border style; `S` can be `FLAT`, `SUNKEN`, `RAISED`, `GROOVE`, `SOLID`, or `RIDGE`—all constants exported by the `tkinter` module.

Cursor

A `cursor` option can be given to change the appearance of the mouse pointer when it moves over the widget. For instance, `cursor='gumby'` changes the pointer to a Gumby figure (the green kind). Other common cursor names used in this book include `watch`, `pencil`, `cross`, and `hand2`.

State

Some widgets also support the notion of a state, which impacts their appearance. For example, a `state=DISABLED` option will generally stipple (gray out) a widget on screen and make it unresponsive; `NORMAL` does not. Some widgets support a `READONLY` state as well, which displays normally but is unresponsive to changes.

Padding

Extra space can be added around many widgets (e.g., buttons, labels, and text) with the `padx= N` and `pady= N` options. Interestingly, you can set these options both in `pack` calls (where it adds empty space around the widget in general) and in a widget object itself (where it makes the widget larger).

To illustrate some of these extra settings, [Example 8-2](#) configures the custom button captured in [Figure 8-2](#) and changes the mouse pointer when it is positioned above it.

Example 8-2. PP4E\Gui\Tour\config-button.py

```
from tkinter import *
widget = Button(text='Spam', padx=10, pady=10)
widget.pack(padx=20, pady=20)
widget.config(cursor='gumby')
widget.config(bd=8, relief=RAISED)
widget.config(bg='dark green', fg='white')
widget.config(font=('helvetica', 20, 'underline italic'))
mainloop()
```

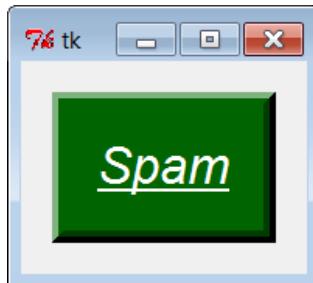


Figure 8-2. Config button at work

To see the effects generated by these two scripts' settings, try out a few changes on your computer. Most widgets can be given a custom appearance in the same way, and we'll see such options used repeatedly in this text. We'll also meet operational configurations, such as `focus` (for focusing input) and others. In fact, widgets can have dozens of options; most have reasonable defaults that produce a native look-and-feel on each windowing platform, and this is one reason for tkinter's simplicity. But tkinter lets you build more custom displays when you want to.



For more on ways to apply configuration options to provide common look-and-feel for your widgets, refer back to “[Customizing Widgets with Classes](#)” on page 400, especially its `ThemedButton` examples. Now that you know more about configuration, its examples' source code should more readily show how configurations applied in widget subclasses are automatically inherited by all instances and subclasses. The new `ttk` extension described in [Chapter 7](#) also provides additional ways to configure widgets with its notion of themes; see the preceding chapter for more details and resources on `ttk`.

Top-Level Windows

tkinter GUIs always have an application root window, whether you get it by default or create it explicitly by calling the `Tk` object constructor. This main root window is the one that opens when your program runs, and it is where you generally pack your most important and long-lived widgets. In addition, tkinter scripts can create any number of independent windows, generated and popped up on demand, by creating `Toplevel` widget objects.

Each `Toplevel` object created produces a new window on the display and automatically adds it to the program's GUI event-loop processing stream (you don't need to call the `mainloop` method of new windows to activate them). [Example 8-3](#) builds a root and two pop-up windows.

Example 8-3. PP4E\Gui\Tour\toplevel0.py

```
import sys
from tkinter import Toplevel, Button, Label

win1 = Toplevel()           # two independent windows
win2 = Toplevel()           # but part of same process

Button(win1, text='Spam', command=sys.exit).pack()
Button(win2, text='SPAM', command=sys.exit).pack()

Label(text='Popups').pack()   # on default Tk() root window
win1.mainloop()
```

The *toplevel0* script gets a root window by default (that's what the `Label` is attached to, since it doesn't specify a real parent), but it also creates two standalone `Toplevel` windows that appear and function independently of the root window, as seen in Figure 8-3.

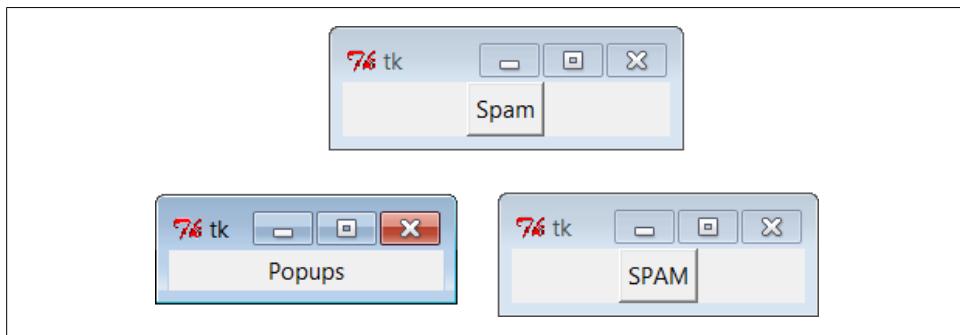


Figure 8-3. Two `Toplevel` windows and a root window

The two `Toplevel` windows on the right are full-fledged windows; they can be independently iconified, maximized, and so on. `Toplevels` are typically used to implement multiple-window displays and pop-up modal and nonmodal dialogs (more on dialogs in the next section). They stay up until they are explicitly destroyed or until the application that created them exits.

In fact, as coded here, pressing the `X` in the upper right corner of either of the `Toplevel` windows kills that window only. On the other hand, the entire program and all its remaining windows are closed if you press either of the created buttons or the main window's `X` (more on shutdown protocols in a moment).

It's important to know that although `Toplevels` are independently active windows, they are not separate processes; if your program exits, all of its windows are erased, including all `Toplevel` windows it may have created. We'll learn how to work around this rule later by launching independent GUI programs.

Toplevel and Tk Widgets

A `Toplevel` is roughly like a `Frame` that is split off into its own window and has additional methods that allow you to deal with top-level window properties. The `Tk` widget is roughly like a `Toplevel`, but it is used to represent the application root window. `Toplevel` windows have parents, but `Tk` windows do not—they are the true roots of the widget hierarchies we build when making tkinter GUIs.

We got a `Tk` root for free in [Example 8-3](#) because the `Label` had a default parent, designated by not having a widget in the first argument of its constructor call:

```
Label(text='Popups').pack()           # on default Tk() root window
```

Passing `None` to a widget constructor's first argument (or to its `master` keyword argument) has the same default-parent effect. In other scripts, we've made the `Tk` root more explicit by creating it directly, like this:

```
root = Tk()
Label(root, text='Popups').pack()      # on explicit Tk() root window
root.mainloop()
```

In fact, because tkinter GUIs are a hierarchy, by default you *always* get at least one `Tk` root window, whether it is named explicitly, as here, or not. Though not typical, there may be more than one `Tk` root if you make them manually, and a program ends if all its `Tk` windows are closed. The first `Tk` top-level window created—whether explicitly by your code, or automatically by Python when needed—is used as the default parent window of widgets and other windows if no parent is provided.

You should generally use the `Tk` root window to display top-level information of some sort. If you don't attach widgets to the root, it may show up as an odd empty window when you run your script (often because you used the default parent unintentionally in your code by omitting a widget's parent and didn't pack widgets attached to it). Technically, you can suppress the default root creation logic and make multiple root windows with the `Tk` widget, as in [Example 8-4](#).

Example 8-4. PP4E\Gui\Tour\toplevel1.py

```
import tkinter
from tkinter import Tk, Button
tkinter.NoDefaultRoot()

win1 = Tk()          # two independent root windows
win2 = Tk()

Button(win1, text='Spam', command=win1.destroy).pack()
Button(win2, text='SPAM', command=win2.destroy).pack()
win1.mainloop()
```

When run, this script displays the two pop-up windows of the screenshot in [Figure 8-3](#) only (there is no third root window). But it's more common to use the `Tk` root as a main window and create `Toplevel` widgets for an application's pop-up windows.

Notice how this GUI's windows use a window's `destroy` method to close just one window, instead of `sys.exit` to shut down the entire program; to see how this method really does its work, let's move on to window protocols.

Top-Level Window Protocols

Both Tk and `Toplevel` widgets export extra methods and features tailored for their top-level role, as illustrated in [Example 8-5](#).

Example 8-5. PP4E\Gui\Tour\toplevel2.py

```
"""
pop up three new windows, with style
destroy() kills one window, quit() kills all windows and app (ends mainloop);
top-level windows have title, icon, iconify/deiconify and protocol for wm events;
there always is an application root window, whether by default or created as an
explicit Tk() object; all top-level windows are containers, but they are never
packed/gridded; Toplevel is like Frame, but a new window, and can have a menu;
"""

from tkinter import *
root = Tk()                                     # explicit root

trees = [('The Larch!',           'light blue'),
         ('The Pine!',            'light green'),
         ('The Giant Redwood!', 'red')]

for (tree, color) in trees:
    win = Toplevel(root)                         # new window
    win.title('Sing...')                          # set border
    win.protocol('WM_DELETE_WINDOW', lambda:None) # ignore close
    win.iconbitmap('py-blue-trans-out.ico')       # not red Tk

    msg = Button(win, text=tree, command=win.destroy)      # kills one win
    msg.pack(expand=YES, fill=BOTH)
    msg.config(padx=10, pady=10, bd=10, relief=Raised)
    msg.config(bg='black', fg=color, font=('times', 30, 'bold italic'))

root.title('Lumberjack demo')
Label(root, text='Main window', width=30).pack()
Button(root, text='Quit All', command=root.quit).pack()      # kills all app
root.mainloop()
```

This program adds widgets to the Tk root window, immediately pops up three `Toplevel` windows with attached buttons, and uses special top-level protocols. When run, it generates the scene captured in living black-and-white in [Figure 8-4](#) (the buttons' text shows up blue, green, and red on a color display).



Figure 8-4. Three Toplevel windows with configurations

There are a few operational details worth noticing here, all of which are more obvious if you run this script on your machine:

Intercepting closes: protocol

Because the window manager close event has been intercepted by this script using the top-level widget `protocol` method, pressing the X in the top-right corner doesn't do anything in the three `Toplevel` pop ups. The name string `WM_DELETE_WINDOW` identifies the close operation. You can use this interface to disallow closes apart from the widgets your script creates. The function created by this script's `lambda:None` does nothing but return `None`.

Killing one window (and its children): destroy

Pressing the big black buttons in any one of the three pop ups kills that pop up only, because the pop up runs the widget `destroy` method. The other windows live on, much as you would expect of a pop-up dialog window. Technically, this call destroys the subject widget and any other widgets for which it is a parent. For windows, this includes all their content. For simpler widgets, the widget is erased.

Because `Toplevel` windows have parents, too, their relationships might matter on a `destroy`—destroying a window, even the automatic or first-made Tk root which is used as the default parent, also destroys all its child windows. Since Tk root windows have no parents, they are unaffected by destroys of other windows. Moreover, destroying the last Tk root window remaining (or the only Tk root created) effectively ends the program. `Toplevel` windows, however, are always destroyed with their parents, and their destruction doesn't impact other windows to which they are not ancestors. This makes them ideal for pop-up dialogs. Technically, a `Toplevel` can be a child of any type of widget and will be destroyed with it, though they are usually children of an automatic or explicit Tk.

Killing all windows: quit

To kill all the windows at once and end the GUI application (really, its active `mainloop` call), the root window's button runs the `quit` method instead. That is, pressing the root window's button ends the program. In general, the `quit` method immediately ends the entire application and closes all its windows. It can be called through any tkinter widget, not just through the top-level window; it's also available on frames, buttons, and so on. See the discussion of the `bind` method and its `<Destroy>` events later in this chapter for more on `quit` and `destroy`.

Window titles: title

As introduced in [Chapter 7](#), top-level window widgets (`Tk` and `Toplevel`) have a `title` method that lets you change the text displayed on the top border. Here, the window title text is set to the string '`Sing...`' in the pop-ups to override the default '`tk`'.

Window icons: iconbitmap

The `iconbitmap` method changes a top-level window's icon. It accepts an icon or bitmap file and uses it for the window's icon graphic when it is both minimized and open. On Windows, pass in the name of a `.ico` file (this example uses one in the current directory); it will replace the default red "Tk" icon that normally appears in the upper-lefthand corner of the window as well as in the Windows task-bar. On other platforms, you may need to use other icon file conventions if the icon calls in this book won't work for you (or simply comment-out the calls altogether if they cause scripts to fail); icons tend to be a platform-specific feature that is dependent upon the underlying window manager.

Geometry management

Top-level windows are containers for other widgets, much like a standalone `Frame`. Unlike frames, though, top-level window widgets are never themselves packed (or gridded, or placed). To embed widgets, this script passes its windows as parent arguments to label and button constructors.

It is also possible to fetch the maximum window size (the physical screen display size, as a `[width, height]` tuple) with the `maxsize()` method, as well as set the initial size of a window with the top-level `geometry("width x height + x + y")` method. It is generally easier and more user-friendly to let tkinter (or your users) work out window size for you, but display size may be used for tasks such as scaling images (see the discussion on PyPhoto in [Chapter 11](#) for an example).

In addition, top-level window widgets support other kinds of protocols that we will utilize later on in this tour:

State

The `iconify` and `withdraw` top-level window object methods allow scripts to hide and erase a window on the fly; `deiconify` redraws a hidden or erased window. The `state` method queries or changes a window's state; valid states passed in or returned include `iconic`, `withdrawn`, `zoomed` (full screen on Windows: use `geometry`

elsewhere), and `normal` (large enough for window content). The methods `lift` and `lower` raise and lower a window with respect to its siblings (`lift` is the Tk `raise` command, but avoids a Python reserved word). See the alarm scripts near the end of [Chapter 9](#) for usage.

Menus

Each top-level window can have its own window menus too; both the `Tk` and the `Toplevel` widgets have a `menu` option used to associate a horizontal menu bar of pull-down option lists. This menu bar looks as it should on each platform on which your scripts are run. We'll explore menus early in [Chapter 9](#).

Most top-level window-manager-related methods can also be named with a “`wm_`” at the front; for instance, `state` and `protocol` can also be called `wm_state` and `wm_protocol`.

Notice that the script in [Example 8-3](#) passes its `Toplevel` constructor calls an explicit parent widget—the Tk root window (that is, `Toplevel(root)`). `Toplevels` can be associated with a parent just as other widgets can, even though they are not visually embedded in their parents. I coded the script this way to avoid what seems like an odd feature; if coded instead like this:

```
win = Toplevel() # new window
```

and if no Tk root yet exists, this call actually generates a default Tk root window to serve as the `Toplevel`'s parent, just like any other widget call without a parent argument. The problem is that this makes the position of the following line crucial:

```
root = Tk() # explicit root
```

If this line shows up above the `Toplevel` calls, it creates the single root window as expected. But if you move this line below the `Toplevel` calls, tkinter creates a default Tk root window that is different from the one created by the script's explicit `Tk` call. You wind up with two Tk roots just as in [Example 8-4](#). Move the `Tk` call below the `Toplevel` calls and rerun it to see what I mean. You'll get a fourth window that is completely empty! As a rule of thumb, to avoid such oddities, make your Tk root windows early on and make them explicit.

All of the top-level protocol interfaces are available only on top-level window widgets, but you can often access them by going through other widgets' `master` attributes—links to the widget parents. For example, to set the title of a window in which a frame is contained, say something like this:

```
theframe.master.title('Spam demo') # master is the container window
```

Naturally, you should do so only if you're sure that the frame will be used in only one kind of window. General-purpose attachable components coded as classes, for instance, should leave window property settings to their client applications.

Top-level widgets have additional tools, some of which we may not meet in this book. For instance, under Unix window managers, you can also set the name used on the window's icon (`iconname`). Because some icon options may be useful when scripts run

on Unix only, see other Tk and tkinter resources for more details on this topic. For now, the next scheduled stop on this tour explores one of the more common uses of top-level windows.

Dialogs

Dialogs are windows popped up by a script to provide or request additional information. They come in two flavors, modal and nonmodal:

Modal

These dialogs block the rest of the interface until the dialog window is dismissed; users must reply to the dialog before the program continues.

Nonmodal

These dialogs can remain on-screen indefinitely without interfering with other windows in the interface; they can usually accept inputs at any time.

Regardless of their modality, dialogs are generally implemented with the `Toplevel` window object we met in the prior section, whether you make the `Toplevel` or not. There are essentially three ways to present pop-up dialogs to users with tkinter—by using common dialog calls, by using the now-dated `Dialog` object, and by creating custom dialog windows with `Toplevels` and other kinds of widgets. Let's explore the basics of all three schemes.

Standard (Common) Dialogs

Because standard dialog calls are simpler, let's start here first. tkinter comes with a collection of precoded dialog windows that implement many of the most common pop ups programs generate—file selection dialogs, error and warning pop ups, and question and answer prompts. They are called *standard dialogs* (and sometimes *common dialogs*) because they are part of the tkinter library, and they use platform-specific library calls to look like they should on each platform. A tkinter file open dialog, for instance, looks like any other on Windows.

All standard dialog calls are modal (they don't return until the dialog box is dismissed by the user), and they block the program's main window while they are displayed. Scripts can customize these dialogs' windows by passing message text, titles, and the like. Since they are so simple to use, let's jump right into [Example 8-6](#) (coded as a `.pyw` file here to avoid a shell pop up when clicked in Windows).

Example 8-6. PP4E\Gui\Tour\dlg1.pyw

```
from tkinter import *
from tkinter.messagebox import *

def callback():
    if askyesno('Verify', 'Do you really want to quit?'):
        showwarning('Yes', 'Quit not yet implemented')
```

```

else:
    showinfo('No', 'Quit has been cancelled')

errmsg = 'Sorry, no Spam allowed!'
Button(text='Quit', command=callback).pack(fill=X)
Button(text='Spam', command=(lambda: showerror('Spam', errmsg))).pack(fill=X)
mainloop()

```

A lambda anonymous function is used here to wrap the call to `showerror` so that it is passed two hardcoded arguments (remember, button-press callbacks get no arguments from tkinter itself). When run, this script creates the main window in [Figure 8-5](#).

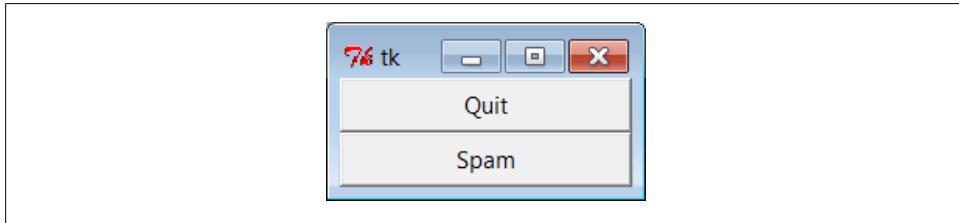


Figure 8-5. dlg1 main window: buttons to trigger pop ups

When you press this window's Quit button, the dialog in [Figure 8-6](#) is popped up by calling the standard `askyesno` function in the tkinter package's `messagebox` module. This looks different on Unix and Macintosh systems, but it looks like you'd expect when run on Windows (and in fact varies its appearance even across different versions and configurations of Windows—using my default Window 7 setup, it looks slightly different than it did on Windows XP in the prior edition).

The dialog in [Figure 8-6](#) blocks the program until the user clicks one of its buttons; if the dialog's Yes button is clicked (or the Enter key is pressed), the dialog call returns with a true value and the script pops up the standard dialog in [Figure 8-7](#) by calling `showwarning`.

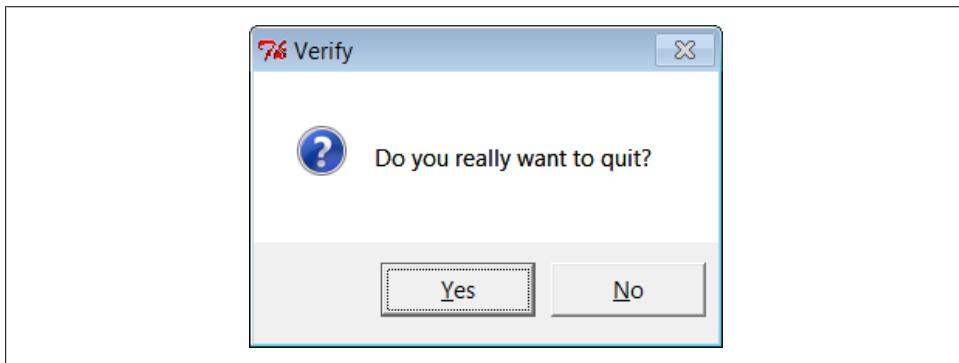


Figure 8-6. dlg1 askyesno dialog (Windows 7)

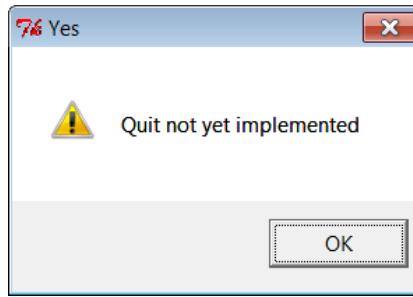


Figure 8-7. *dlg1 showwarning dialog*

There is nothing the user can do with [Figure 8-7](#)'s dialog but press OK. If No is clicked in [Figure 8-6](#)'s quit verification dialog, a `showinfo` call creates the pop up in [Figure 8-8](#) instead. Finally, if the Spam button is clicked in the main window, the standard dialog captured in [Figure 8-9](#) is generated with the standard `showerror` call.

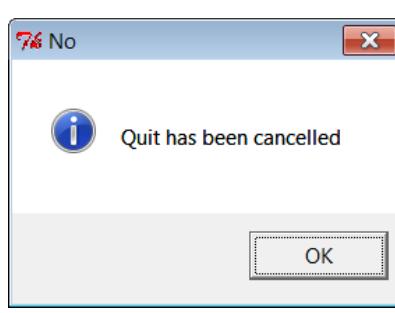


Figure 8-8. *dlg1 showinfo dialog*



Figure 8-9. *dlg1 showerror dialog*

All of this makes for a lot of window pop ups, of course, and you need to be careful not to rely on these dialogs too much (it's generally better to use input fields in long-lived

windows than to distract the user with pop ups). But where appropriate, such pop ups save coding time and provide a nice native look-and-feel.

A “smart” and reusable Quit button

Let’s put some of these canned dialogs to better use. [Example 8-7](#) implements an attachable Quit button that uses standard dialogs to verify the quit request. Because it’s a class, it can be attached and reused in any application that needs a verifying Quit button. Because it uses standard dialogs, it looks as it should on each GUI platform.

Example 8-7. PP4E\Gui\Tour\quitter.py

```
"""
a Quit button that verifies exit requests;
to reuse, attach an instance to other GUIs, and re-pack as desired
"""

from tkinter import *                      # get widget classes
from tkinter.messagebox import askokcancel    # get canned std dialog

class Quitter(Frame):                      # subclass our GUI
    def __init__(self, parent=None):          # constructor method
        Frame.__init__(self, parent)
        self.pack()
        widget = Button(self, text='Quit', command=self.quit)
        widget.pack(side=LEFT, expand=YES, fill=BOTH)

    def quit(self):
        ans = askokcancel('Verify exit', "Really quit?")
        if ans: Frame.quit(self)

if __name__ == '__main__': Quitter().mainloop()
```

This module is mostly meant to be used elsewhere, but it puts up the button it implements when run standalone. [Figure 8-10](#) shows the Quit button itself in the upper left, and the `askokcancel` verification dialog that pops up when Quit is pressed.

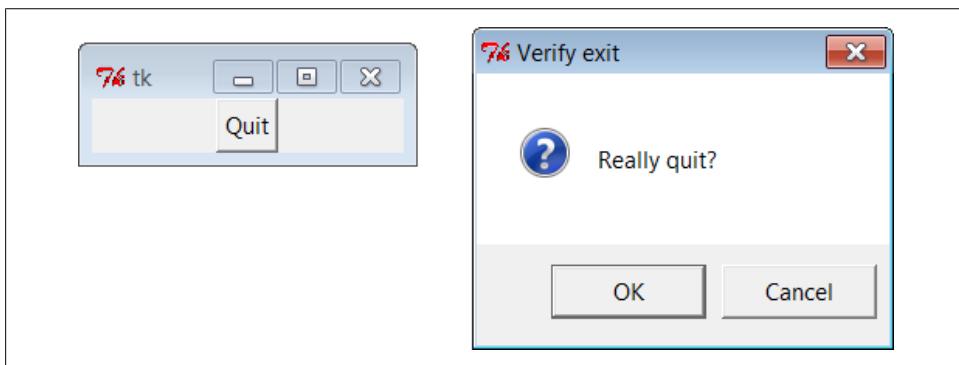


Figure 8-10. Quitter, with `askokcancel` dialog

If you press OK here, `Quitter` runs the `Frame` `quit` method to end the GUI to which this button is attached (really, the `mainloop` call). But to really understand how such a spring-loaded button can be useful, we need to move on and study a client GUI in the next section.

A dialog demo launcher bar

So far, we've seen a handful of standard dialogs, but there are quite a few more. Instead of just throwing these up in dull screenshots, though, let's write a Python demo script to generate them on demand. Here's one way to do it. First of all, in [Example 8-8](#) we write a module to define a table that maps a demo name to a standard dialog call (and we use lambda to wrap the call if we need to pass extra arguments to the dialog function).

Example 8-8. PP4E\Gui\Tour\dialogTable.py

```
# define a name:callback demos table

from tkinter.filedialog import askopenfilename      # get standard dialogs
from tkinter.colorchooser import askcolor           # they live in Lib\tkinter
from tkinter.messagebox import askquestion, showerror
from tkinter.simpledialog import askfloat

demos = {
    'Open': askopenfilename,
    'Color': askcolor,
    'Query': lambda: askquestion('Warning', 'You typed "rm *\nConfirm?'),
    'Error': lambda: showerror('Error!', "He's dead, Jim"),
    'Input': lambda: askfloat('Entry', 'Enter credit card number')
}
```

I put this table in a module so that it might be reused as the basis of other demo scripts later (dialogs are more fun than printing to `stdout`). Next, we'll write a Python script, shown in [Example 8-9](#), which simply generates buttons for all of this table's entries—use its keys as button labels and its values as button callback handlers.

Example 8-9. PP4E\Gui\Tour\demoDlg.py

```
"create a bar of simple buttons that launch dialog demos"

from tkinter import *                  # get base widget set
from dialogTable import demos          # button callback handlers
from quitter import Quitter            # attach a quit object to me

class Demo(Frame):
    def __init__(self, parent=None, **options):
        Frame.__init__(self, parent, **options)
        self.pack()
        Label(self, text="Basic demos").pack()
        for (key, value) in demos.items():
            Button(self, text=key, command=value).pack(side=TOP, fill=BOTH)
        Quitter(self).pack(side=TOP, fill=BOTH)
```

```
if __name__ == '__main__': Demo().mainloop()
```

This script creates the window shown in [Figure 8-11](#) when run as a standalone program; it's a bar of demo buttons that simply route control back to the values of the table in the module `dialogTable` when pressed.

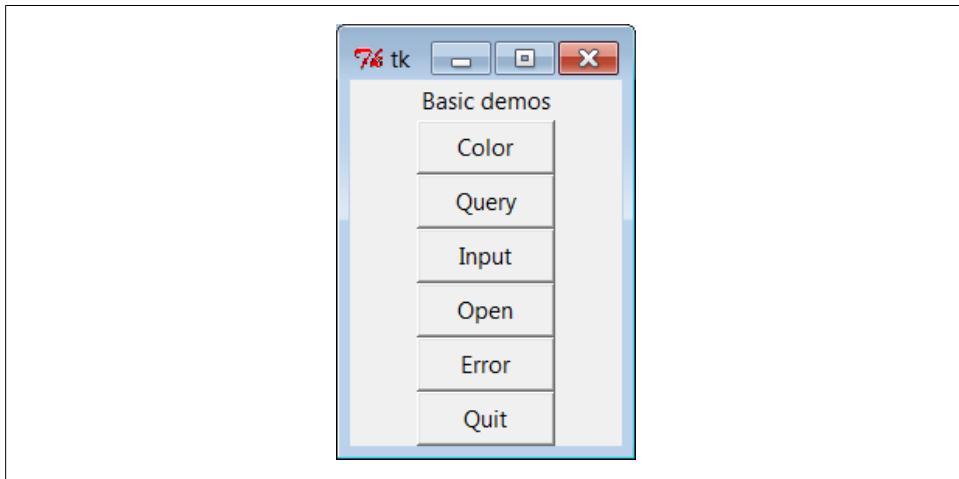


Figure 8-11. demoDlg main window

Notice that because this script is driven by the contents of the `dialogTable` module's dictionary, we can change the set of demo buttons displayed by changing just `dialogTable` (we don't need to change any executable code in `demoDlg`). Also note that the `Quit` button here is an attached instance of the `Quitter` class of the prior section whose frame is repacked to stretch like the other buttons as needed here—it's at least one bit of code that you never have to write again.

This script's class also takes care to pass any `**options` constructor configuration keyword arguments on to its `Frame` superclass. Though not used here, this allows callers to pass in configuration options at creation time (`Demo(o=v)`), instead of configuring after the fact (`d.config(o=v)`). This isn't strictly required, but it makes the demo class work just like a normal tkinter frame widget (which is what subclassing makes it, after all). We'll see how this can be used to good effect later.

We've already seen some of the dialogs triggered by this demo bar window's other buttons, so I'll just step through the new ones here. Pressing the main window's `Query` button, for example, generates the standard pop up in [Figure 8-12](#).

This `askquestion` dialog looks like the `askyesno` we saw earlier, but actually it returns either string "yes" or "no" (`askyesno` and `askokcancel` return `True` or `False` instead—trivial but true). Pressing the demo bar's `Input` button generates the standard `askfloat` dialog box shown in [Figure 8-13](#).

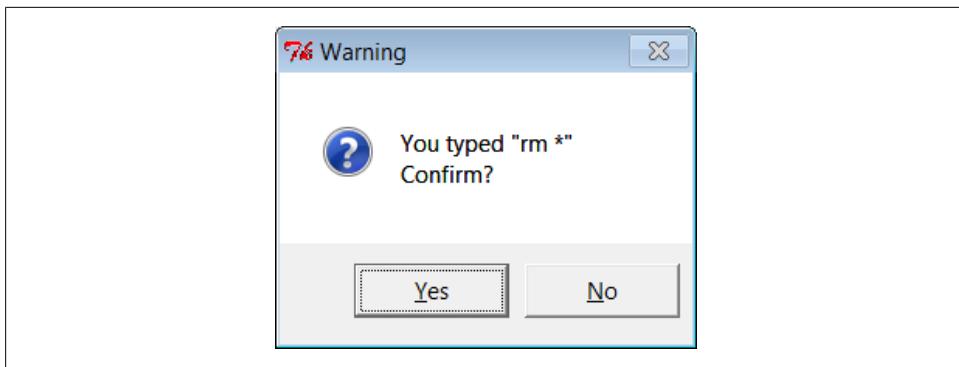


Figure 8-12. `demoDlg` query, `askquestion` dialog

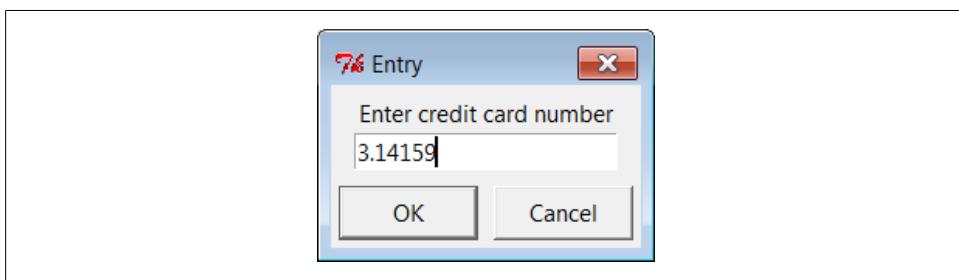


Figure 8-13. `demoDlg` input, `askfloat` dialog

This dialog automatically checks the input for valid floating-point syntax before it returns, and it is representative of a collection of single-value input dialogs (`askinteger` and `askstring` prompt for integer and string inputs, too). It returns the input as a floating-point number object (not as a string) when the OK button or Enter key is pressed, or the Python `None` object if the user clicks Cancel. Its two relatives return the input as integer and string objects instead.

When the demo bar's Open button is pressed, we get the standard file open dialog made by calling `askopenfilename` and captured in [Figure 8-14](#). This is Windows 7's look-and-feel; it can look radically different on Macs, Linux, and older versions of Windows, but appropriately so.

A similar dialog for selecting a save-as filename is produced by calling `asksaveasfilename` (see the Text widget section in [Chapter 9](#) for a first example). Both file dialogs let the user navigate through the filesystem to select a subject filename, which is returned with its full directory pathname when Open is pressed; an empty string comes back if Cancel is pressed instead. Both also have additional protocols not demonstrated by this example:

- They can be passed a `filetypes` keyword argument—a set of name patterns used to select files, which appear in the pull-down list near the bottom of the dialog.

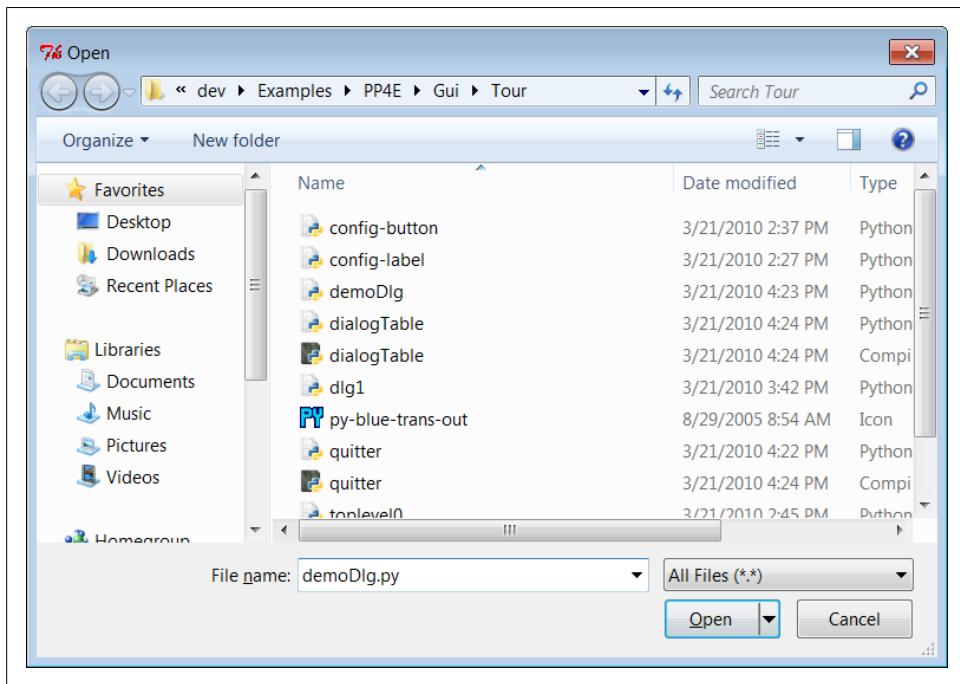


Figure 8-14. *demoDlg* open, `askopenfilename` dialog

- They can be passed an `initialdir` (start directory), `initialfile` (for “File name”), `title` (for the dialog window), `defaultextension` (appended if the selection has none), and `parent` (to appear as an embedded child instead of a pop-up dialog).
- They can be made to remember the last directory selected by using exported objects instead of these function calls—a hook we’ll make use of in later longer-lived examples.

Another common dialog call in the `tkinter filedialog` module, `askdirectory`, can be used to pop up a dialog that allows users to choose a directory rather than a file. It presents a tree view that users can navigate to pick the desired directory, and it accepts keyword arguments including `initialdir` and `title`. The corresponding `Directory` object remembers the last directory selected and starts there the next time the dialog is shown.

We’ll use most of these interfaces later in the book, especially for the file dialogs in the PyEdit example in [Chapter 11](#), but feel free to flip ahead for more details now. The directory selection dialog will show up in the PyPhoto example in [Chapter 11](#) and the PyMailGUI example in [Chapter 14](#); again, skip ahead for code and screenshots.

Finally, the demo bar’s Color button triggers a standard `askcolor` call, which generates the standard color selection dialog shown in [Figure 8-15](#).

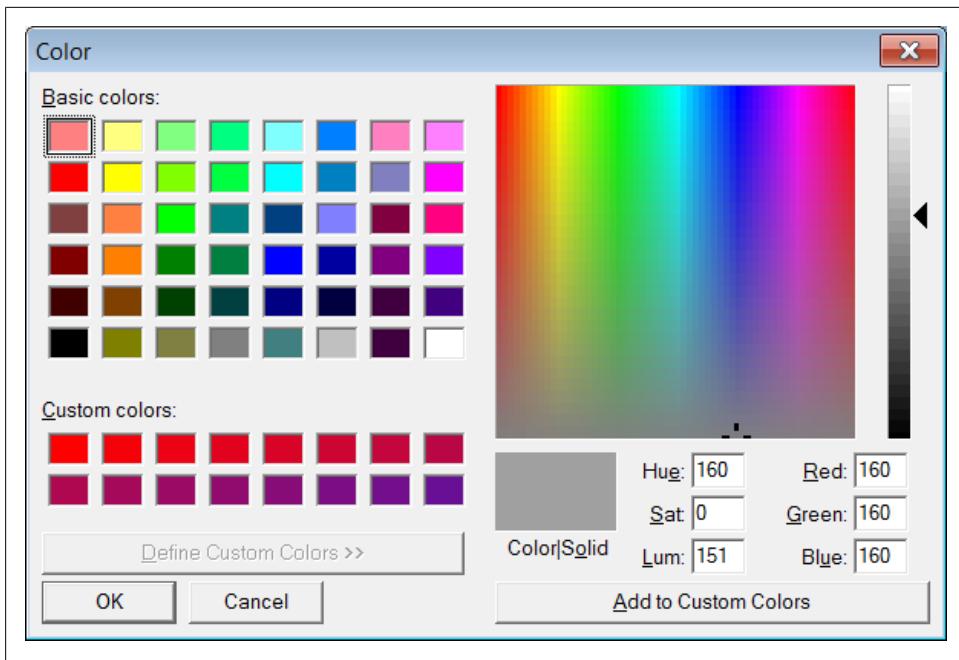


Figure 8-15. *demoDlg* color, *askcolor* dialog

If you press its OK button, it returns a data structure that identifies the selected color, which can be used in all color contexts in tkinter. It includes RGB values and a hexadecimal color string (e.g., ((160, 160, 160), '#a0a0a0')). More on how this tuple can be useful in a moment. If you press Cancel, the script gets back a tuple containing two nones (None of the Python variety, that is).

Printing dialog results and passing callback data with lambdas

The dialog demo launcher bar displays standard dialogs and can be made to display others by simply changing the `dialogTable` module it imports. As coded, though, it really shows only dialogs; it would also be nice to see their return values so that we know how to use them in scripts. [Example 8-10](#) adds printing of standard dialog results to the `stdout` standard output stream.

Example 8-10. PP4E\Gui\Tour\demoDlg-print.py

```
"""
similar, but show return values of dialog calls; the lambda saves data from
the local scope to be passed to the handler (button press handlers normally
get no arguments, and enclosing scope references don't work for loop variables)
and works just like a nested def statement: def func(key=key): self.printit(key)
"""
```

```

from tkinter import *           # get base widget set
from dialogTable import demos   # button callback handlers
from quitter import Quitter     # attach a quit object to me

class Demo(Frame):
    def __init__(self, parent=None):
        Frame.__init__(self, parent)
        self.pack()
        Label(self, text="Basic demos").pack()
        for key in demos:
            func = (lambda key=key: self.printit(key))
            Button(self, text=key, command=func).pack(side=TOP, fill=BOTH)
        Quitter(self).pack(side=TOP, fill=BOTH)

    def printit(self, name):
        print(name, 'returns =>', demos[name]())      # fetch, call, print

if __name__ == '__main__':
    Demo().mainloop()

```

This script builds the same main button-bar window, but notice that the callback handler is an anonymous function made with a lambda now, not a direct reference to dialog calls in the imported `dialogTable` dictionary:

```

# use enclosing scope lookup
func = (lambda key=key: self.printit(key))

```

We talked about this in the prior chapter's tutorial, but this is the first time we've actually used `lambda` like this, so let's get the facts straight. Because button-press callbacks are run with no arguments, if we need to pass *extra data* to the handler, it must be wrapped in an object that remembers that extra data and passes it along, by deferring the call to the actual handler. Here, a button press runs the function generated by the `lambda`, an indirect call layer that retains information from the enclosing scope. The net effect is that the real handler, `printit`, receives an extra required `name` argument giving the demo associated with the button pressed, even though this argument wasn't passed back from `tkinter` itself. In effect, the `lambda` remembers and passes on state information.

Notice, though, that this `lambda` function's body references both `self` and `key` in the enclosing method's local scope. In all recent Pythons, the reference to `self` just works because of the enclosing function scope lookup rules, but we need to pass `key` in explicitly with *a default argument* or else it will be the same in all the generated `lambda` functions—the value it has after the last loop iteration. As we learned in [Chapter 7](#), enclosing scope references are resolved when the nested function is called, but defaults are resolved when the nested function is created. Because `self` won't change after the function is made, we can rely on the scope lookup rules for that name, but not for loop variables like `key`.

In earlier Pythons, default arguments were required to pass all values in from enclosing scopes explicitly, using either of these two techniques:

```
# use simple defaults
func = (lambda self=self, name=key: self.printit(name))

# use a bound method default
func = (lambda handler=self.printit, name=key: handler(name))
```

Today, we can get away with the simpler enclosing -scope reference technique for `self`, though we still need a default for the `key` loop variable (and you may still see the default forms in older Python code).

Note that the parentheses around the lambdas are not required here; I add them as a personal style preference just to set the lambda off from its surrounding code (your mileage may vary). Also notice that the lambda does the same work as a nested `def` statement here; in practice, though, the lambda could appear within the call to `Button` itself because it is an expression and it need not be assigned to a name. The following two forms are equivalent:

```
for (key, value) in demos.items():
    func = (lambda key=key: self.printit(key))           # can be nested in Button()

for (key, value) in demos.items():
    def func(key=key): self.printit(key)                 # but def statement cannot
```

You can also use a callable class object here that retains state as instance attributes (see the tutorial's `__call__` example in [Chapter 7](#) for hints). But as a rule of thumb, if you want a lambda's result to use any names from the enclosing scope when later called, either simply name them and let Python save their values for future use, or pass them in with defaults to save the values they have at lambda function creation time. The latter scheme is required only if the variable used may change before the callback occurs.

When run, this script creates the same window ([Figure 8-11](#)) but also prints dialog return values to standard output; here is the output after clicking all the demo buttons in the main window and picking both Cancel/No and then OK/Yes buttons in each dialog:

```
C:\...\PP4E\Gui\Tour> python demoDlg-print.py
Color returns => (None, None)
Color returns => ((128.5, 128.5, 255.99609375), '#8080ff')
Query returns => no
Query returns => yes
Input returns => None
Input returns => 3.14159
Open returns =>
Open returns => C:/Users/mark/Stuff/Books/4E/PP4E/dev/Examples/PP4E/Launcher.py
Error returns => ok
```

Now that I've shown you these dialog results, I want to next show you how one of them can actually be useful.

Letting users select colors on the fly

The standard color selection dialog isn't just another pretty face—scripts can pass the hexadecimal color string it returns to the `bg` and `fg` widget color configuration options we met earlier. That is, `bg` and `fg` accept both a color name (e.g., `blue`) and an `askcolor` hex RGB result string that starts with a `#` (e.g., the `#8080ff` in the last output line of the prior section).

This adds another dimension of customization to tkinter GUIs: instead of hardcoding colors in your GUI products, you can provide a button that pops up color selectors that let users choose color preferences on the fly. Simply pass the color string to widget `config` methods in callback handlers, as in [Example 8-11](#).

Example 8-11. PP4E\Gui\Tour\setcolor.py

```
from tkinter import *
from tkinter.colorchooser import askcolor

def setBgColor():
    (triple, hexstr) = askcolor()
    if hexstr:
        print(hexstr)
        push.config(bg=hexstr)

root = Tk()
push = Button(root, text='Set Background Color', command=setBgColor)
push.config(height=3, font=('times', 20, 'bold'))
push.pack(expand=YES, fill=BOTH)
root.mainloop()
```

This script creates the window in [Figure 8-16](#) when launched (its button's background is a sort of green, but you'll have to trust me on this). Pressing the button pops up the color selection dialog shown earlier; the color you pick in that dialog becomes the background color of this button after you press OK.

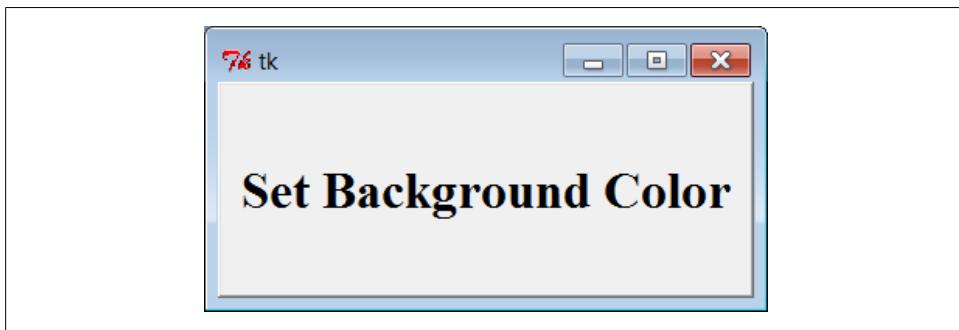


Figure 8-16. setcolor main window

Color strings are also printed to the `stdout` stream (the console window); run this on your computer to experiment with available color settings:

```
C:\...\PP4E\Gui\Tour> python setcolor.py
#0080c0
#408080
#77d5df
```

Other standard dialog calls

We've seen most of the standard dialogs and we'll use these pop ups in examples throughout the rest of this book. But for more details on other calls and options available, either consult other tkinter documentation or browse the source code of the modules used at the top of the `dialogTable` module in [Example 8-8](#); all are simple Python files installed in the `tkinter` subdirectory of the Python source library on your machine (e.g., in `C:\Python31\Lib` on Windows). And keep this demo bar example filed away for future reference; we'll reuse it later in the tour for callback actions when we meet other button-like widgets.

The Old-Style Dialog Module

In older Python code, you may see dialogs occasionally coded with the standard tkinter `dialog` module. This is a bit dated now, and it uses an X Windows look-and-feel; but just in case you run across such code in your Python maintenance excursions, [Example 8-12](#) gives you a feel for the interface.

Example 8-12. PP4E\Gui\Tour\dlg-old.py

```
from tkinter import *
from tkinter.dialog import Dialog

class OldDialogDemo(Frame):
    def __init__(self, master=None):
        Frame.__init__(self, master)
        Pack.config(self) # same as self.pack()
        Button(self, text='Pop1', command=self.dialog1).pack()
        Button(self, text='Pop2', command=self.dialog2).pack()

    def dialog1(self):
        ans = Dialog(self,
                     title   = 'Popup Fun!',
                     text    = 'An example of a popup-dialog '
                               'box, using older "Dialog.py".',
                     bitmap  = 'questhead',
                     default = 0, strings = ('Yes', 'No', 'Cancel'))
        if ans.num == 0: self.dialog2()

    def dialog2(self):
        Dialog(self, title   = 'HAL-9000',
               text    = "I'm afraid I can't let you do that, Dave...",
               bitmap  = 'hourglass',
```

```
default = 0, strings = ('spam', 'SPAM'))  
  
if __name__ == '__main__': OldDialogDemo().mainloop()
```

If you supply `Dialog` a tuple of button labels and a message, you get back the index of the button pressed (the leftmost is index zero). `Dialog` windows are modal: the rest of the application's windows are disabled until the `Dialog` receives a response from the user. When you press the Pop2 button in the main window created by this script, the second dialog pops up, as shown in [Figure 8-17](#).

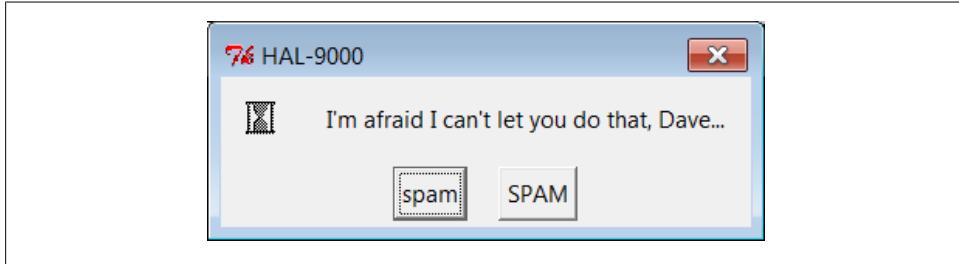


Figure 8-17. Old-style dialog

This is running on Windows, and as you can see, it is nothing like what you would expect on that platform for a question dialog. In fact, this dialog generates an X Windows look-and-feel, regardless of the underlying platform. Because of both `Dialog`'s appearance and the extra complexity required to program it, you are probably better off using the standard dialog calls of the prior section instead.

Custom Dialogs

The dialogs we've seen so far have a standard appearance and interaction. They are fine for many purposes, but often we need something a bit more custom. For example, forms that request multiple field inputs (e.g., name, age, shoe size) aren't directly addressed by the common dialog library. We could pop up one single-input dialog in turn for each requested field, but that isn't exactly user friendly.

Custom dialogs support arbitrary interfaces, but they are also the most complicated to program. Even so, there's not much to it—simply create a pop-up window as a `Toplevel` with attached widgets, and arrange a callback handler to fetch user inputs entered in the dialog (if any) and to destroy the window. To make such a custom dialog modal, we also need to wait for a reply by giving the window input focus, making other windows inactive, and waiting for an event. [Example 8-13](#) illustrates the basics.

Example 8-13. PP4E\Gui\Tour\dlg-custom.py

```
import sys  
from tkinter import *\br/>makemodal = (len(sys.argv) > 1)
```

```

def dialog():
    win = Toplevel()                      # make a new window
    Label(win, text='Hard drive reformatted!').pack() # add a few widgets
    Button(win, text='OK', command=win.destroy).pack() # set destroy callback
    if makemodal:
        win.focus_set()                  # take over input focus,
        win.grab_set()                  # disable other windows while I'm open,
        win.wait_window()               # and wait here until win destroyed
    print('dialog exit')                 # else returns right away

root = Tk()
Button(root, text='popup', command=dialog).pack()
root.mainloop()

```

This script is set up to create a pop-up dialog window in either modal or nonmodal mode, depending on its `makemodal` global variable. If it is run with no command-line arguments, it picks nonmodal style, captured in [Figure 8-18](#).

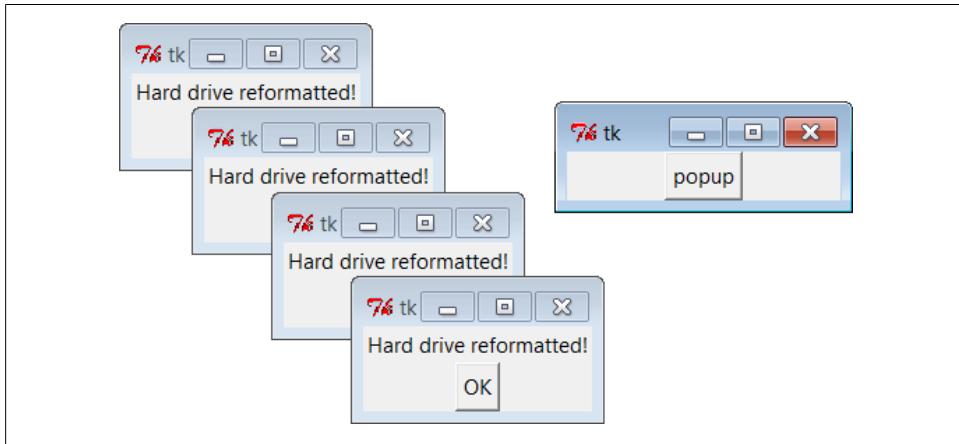


Figure 8-18. Nonmodal custom dialogs at work

The window in the upper right is the root window here; pressing its “popup” button creates a new pop-up dialog window. Because dialogs are nonmodal in this mode, the root window remains active after a dialog is popped up. In fact, nonmodal dialogs never block other windows, so you can keep pressing the root’s button to generate as many copies of the pop-up window as will fit on your screen. Any or all of the pop ups can be killed by pressing their OK buttons, without killing other windows in this display.

Making custom dialogs modal

Now, when the script is run with a command-line argument (e.g., `python dlg-custom.py 1`), it makes its pop ups modal instead. Because modal dialogs grab all of the interface’s attention, the main window becomes inactive in this mode until the pop up is killed; you can’t even click on it to reactivate it while the dialog is

open. Because of that, you can never make more than one copy of the pop up on-screen at once, as shown in [Figure 8-19](#).

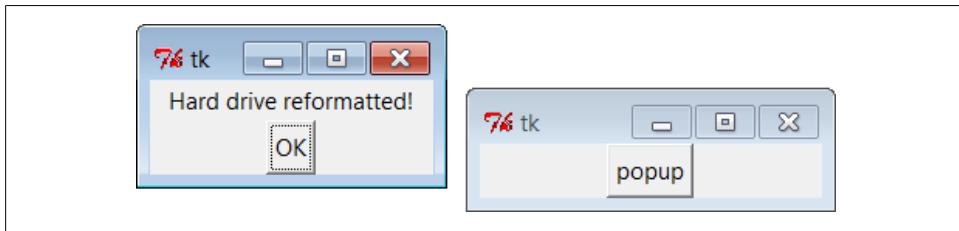


Figure 8-19. A modal custom dialog at work

In fact, the call to the `dialog` function in this script doesn't return until the dialog window on the left is dismissed by pressing its OK button. The net effect is that modal dialogs impose a function call-like model on an otherwise event-driven programming model; user inputs can be processed right away, not in a callback handler triggered at some arbitrary point in the future.

Forcing such a linear control flow on a GUI takes a bit of extra work, though. The secret to locking other windows and waiting for a reply boils down to three lines of code, which are a general pattern repeated in most custom modal dialogs.

`win.focus_set()`

Makes the window take over the application's input focus, as if it had been clicked with the mouse to make it the active window. This method is also known by the synonym `focus`, and it's also common to set the focus on an input widget within the dialog (e.g., an `Entry`) rather than on the entire window.

`win.grab_set()`

Disables all other windows in the application until this one is destroyed. The user cannot interact with other windows in the program while a grab is set.

`win.wait_window()`

Pauses the caller until the `win` widget is destroyed, but keeps the main event-processing loop (`mainloop`) active during the pause. That means that the GUI at large remains active during the wait; its windows redraw themselves if covered and uncovered, for example. When the window is destroyed with the `destroy` method, it is erased from the screen, the application grab is automatically released, and this method call finally returns.

Because the script waits for a window destroy event, it must also arrange for a callback handler to destroy the window in response to interaction with widgets in the dialog window (the only window active). This example's dialog is simply informational, so its OK button calls the window's `destroy` method. In user-input dialogs, we might instead install an Enter key-press callback handler that fetches data typed into an `Entry` widget and then calls `destroy` (see later in this chapter).

Other ways to be modal

Modal dialogs are typically implemented by waiting for a newly created pop-up window's `destroy` event, as in this example. But other schemes are viable too. For example, it's possible to create dialog windows ahead of time, and show and hide them as needed with the top-level window's `deiconify` and `withdraw` methods (see the alarm scripts near the end of [Chapter 9](#) for details). Given that window creation speed is generally fast enough as to appear instantaneous today, this is much less common than making and destroying a window from scratch on each interaction.

It's also possible to implement a modal state by waiting for a tkinter variable to change its value, instead of waiting for a window to be destroyed. See this chapter's later discussion of tkinter variables (which are class objects, not normal Python variables) and the `wait_variable` method discussed near the end of [Chapter 9](#) for more details. This scheme allows a long-lived dialog box's callback handler to signal a state change to a waiting main program, without having to destroy the dialog box.

Finally, if you call the `mainloop` method recursively, the call won't return until the widget `quit` method has been invoked. The `quit` method terminates a `mainloop` call, and so normally ends a GUI program. But it will simply exit a recursive `mainloop` level if one is active. Because of this, modal dialogs can also be written without `wait` method calls if you are careful. For instance, [Example 8-14](#) works the same way as the modal mode of `dlg-custom`.

Example 8-14. PP4E\Gui\Tour\dlg-recursive.py

```
from tkinter import *

def dialog():
    win = Toplevel()
    Label(win, text='Hard drive reformatted!').pack()          # make a new window
    Button(win, text='OK', command=win.quit).pack()            # add a few widgets
    win.protocol('WM_DELETE_WINDOW', win.quit)                 # set quit callback
                                                               # quit on wm close too!

    win.focus_set()                                         # take over input focus,
    win.grab_set()                                         # disable other windows while I'm open,
    win.mainloop()                                          # and start a nested event loop to wait
    win.destroy()
    print('dialog exit')

root = Tk()
Button(root, text='popup', command=dialog).pack()
root.mainloop()
```

If you go this route, be sure to call `quit` rather than `destroy` in dialog callback handlers (`destroy` doesn't terminate the `mainloop` level), and be sure to use `protocol` to make the window border close button call `quit` too (or else it won't end the recursive `mainloop` level call and may generate odd error messages when your program finally exits). Because of this extra complexity, you're probably better off using `wait_window` or `wait_variable`, not recursive `mainloop` calls.

We'll see how to build form-like dialogs with labels and input fields later in this chapter when we meet `Entry`, and again when we study the `grid` manager in [Chapter 9](#). For more custom dialog examples, see `ShellGui` ([Chapter 10](#)), `PyMailGUI` ([Chapter 14](#)), `PyCalc` ([Chapter 19](#)), and the nonmodal `form.py` ([Chapter 12](#)). Here, we're moving on to learn more about events that will prove to be useful currency at later tour destinations.

Binding Events

We met the `bind` widget method in the prior chapter, when we used it to catch button presses in the tutorial. Because `bind` is commonly used in conjunction with other widgets (e.g., to catch return key presses for input boxes), we're going to make a stop early in the tour here as well. [Example 8-15](#) illustrates more `bind` event protocols.

Example 8-15. PP4E\Gui\Tour\bind.py

```
from tkinter import *

def showPosEvent(event):
    print('Widget=%s X=%s Y=%s' % (event.widget, event.x, event.y))

def showAllEvent(event):
    print(event)
    for attr in dir(event):
        if not attr.startswith('__'):
            print(attr, '=>', getattr(event, attr))

def onKeyPress(event):
    print('Got key press:', event.char)

def onArrowKey(event):
    print('Got up arrow key press')

def onReturnKey(event):
    print('Got return key press')

def onLeftClick(event):
    print('Got left mouse button click:', end=' ')
    showPosEvent(event)

def onRightClick(event):
    print('Got right mouse button click:', end=' ')
    showPosEvent(event)

def onMiddleClick(event):
    print('Got middle mouse button click:', end=' ')
    showPosEvent(event)
    showAllEvent(event)

def onLeftDrag(event):
    print('Got left mouse button drag:', end=' ')
    showPosEvent(event)
```

```

def onDoubleLeftClick(event):
    print('Got double left mouse click', end=' ')
    showPosEvent(event)
    tkroot.quit()

tkroot = Tk()
labelfont = ('courier', 20, 'bold')
widget = Label(tkroot, text='Hello bind world')           # family, size, style
widget.config(bg='red', font=labelfont)                   # red background, large font
widget.config(height=5, width=20)                         # initial size: lines,chars
widget.pack(expand=YES, fill=BOTH)

widget.bind('<Button-1>', onLeftClick)                  # mouse button clicks
widget.bind('<Button-3>', onRightClick)
widget.bind('<Button-2>', onMiddleClick)
widget.bind('<Double-1>', onDoubleLeftClick)
widget.bind('<B1-Motion>', onLeftDrag)

widget.bind('<KeyPress>', onKeyPress)                    # all keyboard presses
widget.bind('<Up>', onArrowKey)
widget.bind('<Return>', onReturnKey)
widget.focus()                                           # or bind keypress to tkroot
tkroot.title('Click Me')
tkroot.mainloop()

```

Most of this file consists of callback handler functions triggered when bound events occur. As we learned in [Chapter 7](#), this type of callback receives an event object argument that gives details about the event that fired. Technically, this argument is an instance of the tkinter `Event` class, and its details are attributes; most of the callbacks simply trace events by displaying relevant event attributes.

When run, this script makes the window shown in [Figure 8-20](#); it's mostly intended just as a surface for clicking and pressing event triggers.



Figure 8-20. A bind window for the clicking

The black-and-white medium of the book you're holding won't really do justice to this script. When run live, it uses the configuration options shown earlier to make the

window show up as black on red, with a large Courier font. You'll have to take my word for it (or run this on your own).

But the main point of this example is to demonstrate other kinds of event binding protocols at work. We saw a script that intercepted left and double-left mouse clicks with the widget `bind` method in [Chapter 7](#), using event names `<Button-1>` and `<Double-1>`; the script here demonstrates other kinds of events that are commonly caught with `bind`:

`<KeyPress>`

To catch the press of a single key on the keyboard, register a handler for the `<KeyPress>` event identifier; this is a lower-level way to input data in GUI programs than the `Entry` widget covered in the next section. The key pressed is returned in ASCII string form in the event object passed to the callback handler (`event.char`). Other attributes in the event structure identify the key pressed in lower-level detail. Key presses can be intercepted by the top-level root window widget or by a widget that has been assigned keyboard focus with the `focus` method used by this script.

`<B1-Motion>`

This script also catches mouse motion while a button is held down: the registered `<B1-Motion>` event handler is called every time the mouse is moved while the left button is pressed and receives the current X/Y coordinates of the mouse pointer in its event argument (`event.x`, `event.y`). Such information can be used to implement object moves, drag-and-drop, pixel-level painting, and so on (e.g., see the PyDraw examples in [Chapter 11](#)).

`<Button-3>, <Button-2>`

This script also catches right and middle mouse button clicks (known as buttons 3 and 2). To make the middle button 2 click work on a two-button mouse, try clicking both buttons at the same time; if that doesn't work, check your mouse setting in your properties interface (the Control Panel on Windows).

`<Return>, <Up>`

To catch more specific kinds of key presses, this script registers for the Return/Enter and up-arrow key press events; these events would otherwise be routed to the general `<KeyPress>` handler and require event analysis.

Here is what shows up in the `stdout` output stream after a left click, right click, left click and drag, a few key presses, a Return and up-arrow press, and a final double-left click to exit. When you press the left mouse button and drag it around on the display, you'll get lots of drag event messages; one is printed for every move during the drag (and one Python callback is run for each):

```
C:\...\PP4E\Gui\Tour> python bind.py
Got left mouse button click: Widget=.25763696 X=376 Y=53
Got right mouse button click: Widget=.25763696 X=36 Y=60
Got left mouse button click: Widget=.25763696 X=144 Y=43
Got left mouse button drag: Widget=.25763696 X=144 Y=45
Got left mouse button drag: Widget=.25763696 X=144 Y=47
```

```
Got left mouse button drag: Widget=.25763696 X=145 Y=50
Got left mouse button drag: Widget=.25763696 X=146 Y=51
Got left mouse button drag: Widget=.25763696 X=149 Y=53
Got key press: s
Got key press: p
Got key press: a
Got key press: m
Got key press: 1
Got key press: -
Got key press: 2
Got key press: .
Got return key press
Got up arrow key press
Got left mouse button click: Widget=.25763696 X=300 Y=68
Got double left mouse click Widget=.25763696 X=300 Y=68
```

For mouse-related events, callbacks print the X and Y coordinates of the mouse pointer, in the event object passed in. Coordinates are usually measured in pixels from the upper-left corner (0,0), but are relative to the widget being clicked. Here's what is printed for a left, middle, and double-left click. Notice that the middle-click callback dumps the entire argument—all of the `Event` object's attributes (less internal names that begin with “`_`” which includes the `__doc__` string, and default operator overloading methods inherited from the implied `object` superclass in Python 3.X). Different event types set different event attributes; most key presses put something in `char`, for instance:

```
C:\...\PP4E\Gui\Tour> python bind.py
Got left mouse button click: Widget=.25632624 X=6 Y=6
Got middle mouse button click: Widget=.25632624 X=212 Y=95
<tkinter.Event object at 0x018CA210>
char => ???
delta => 0
height => ???
keycode => ???
keysym => ???
keysym_num => ???
num => 2
send_event => False
serial => 17
state => 0
time => 549707945
type => 4
widget => .25632624
width => ???
x => 212
x_root => 311
y => 95
y_root => 221
Got left mouse button click: Widget=.25632624 X=400 Y=183
Got double left mouse click Widget=.25632624 X=400 Y=183
```

Other bind Events

Besides those illustrated in this example, a tkinter script can register to catch additional kinds of bindable events. For example:

- <ButtonRelease> fires when a button is released (<ButtonPress> is run when the button first goes down).
- <Motion> is triggered when a mouse pointer is moved.
- <Enter> and <Leave> handlers intercept mouse entry and exit in a window's display area (useful for automatically highlighting a widget).
- <Configure> is invoked when the window is resized, repositioned, and so on (e.g., the event object's `width` and `height` give the new window size). We'll make use of this to resize the display on window resizes in the PyClock example of [Chapter 11](#).
- <Destroy> is invoked when the window widget is destroyed (and differs from the `protocol` mechanism for window manager close button presses). Since this interacts with widget `quit` and `destroy` methods, I'll say more about the event later in this section.
- <FocusIn> and <FocusOut> are run as the widget gains and loses focus.
- <Map> and <Unmap> are run when a window is opened and iconified.
- <Escape>, <BackSpace>, and <Tab> catch other special key presses.
- <Down>, <Left>, and <Right> catch other arrow key presses.

This is not a complete list, and event names can be written with a somewhat sophisticated syntax of their own. For instance:

- *Modifiers* can be added to event identifiers to make them even more specific; for instance, <B1-Motion> means moving the mouse with the left button pressed, and <KeyPress-a> refers to pressing the “a” key only.
- *Synonyms* can be used for some common event names; for instance, <Button Press-1>, <Button-1>, and <1> mean a left mouse button press, and <KeyPress-a> and <Key-a> mean the “a” key. All forms are case sensitive: use <Key-Escape>, not <KEY-ESCAPE>.
- *Virtual* event identifiers can be defined within double bracket pairs (e.g., <<Paste Text>>) to refer to a selection of one or more event sequences.

In the interest of space, though, we'll defer to other Tk and tkinter reference sources for an exhaustive list of details on this front. Alternatively, changing some of the settings in the example script and rerunning can help clarify some event behavior, too; this is Python, after all.

More on <Destroy> events and the quit and destroy methods

Before we move on, one event merits a few extra words: the <Destroy> event (whose name is case significant) is run when a widget is being destroyed, as a result of both

script method calls and window closures in general, including those at program exit. If you bind this on a window, it will be triggered once for each widget in the window; the callback’s event argument `widget` attribute gives the widget being destroyed, and you can check this to detect a particular widget’s destruction. If you bind this on a specific widget instead, it will be triggered once for that widget’s destruction only.

It’s important to know that a widget is in a “half dead” state (Tk’s terminology) when this event is triggered—it still exists, but most operations on it fail. Because of that, the `<Destroy>` event is not intended for GUI activity in general; for instance, checking a text widget’s changed state or fetching its content in a `<Destroy>` handler can both fail with exceptions. In addition, this event’s handler cannot cancel the destruction in general and resume the GUI; if you wish to intercept and verify or suppress window closes when a user clicks on a window’s X button, use `WM_DELETE_WINDOW` in top-level windows’ `protocol` methods as described earlier in this chapter.

You should also know that running a tkinter widget’s `quit` method does not trigger any `<Destroy>` events on exit, and even leads to a fatal Python error on program exit in 3.X if any `<Destroy>` event handlers are registered. Because of this, programs that bind this event for non-GUI window exit actions should usually call `destroy` instead of `quit` to close, and rely on the fact that a program exits when the last remaining or only Tk root window (default or explicit) is destroyed as described earlier. This precludes using `quit` for immediate shutdowns, though you can still run `sys.exit` for brute-force exits.

A script can also perform program exit actions in code run after the `mainloop` call returns, but the GUI is gone completely at this point, and this code is not associated with any particular widget. Watch for more on this event when we study the PyEdit example program in [Chapter 11](#); at the risk of spoiling the end of this story, we’ll find it unusable for verifying changed text saves.

Message and Entry

The `Message` and `Entry` widgets allow for display and input of simple text. Both are essentially functional subsets of the `Text` widget we’ll meet later; `Text` can do everything `Message` and `Entry` can, but not vice versa.

Message

The `Message` widget is simply a place to display text. Although the standard `showinfo` dialog we met earlier is perhaps a better way to display pop-up messages, `Message` splits up long strings automatically and flexibly and can be embedded inside container widgets any time you need to add some read-only text to a display. Moreover, this widget sports more than a dozen configuration options that let you customize its appearance. [Example 8-16](#) and [Figure 8-21](#) illustrate `Message` basics, and demonstrates how `Message` reacts to horizontal stretching with `fill` and `expand`; see [Chapter 7](#) for more on resizing and Tk or tkinter references for other options `Message` supports.

Example 8-16. PP4E\Gui\tour\message.py

```
from tkinter import *
msg = Message(text="Oh by the way, which one's Pink?")
msg.config(bg='pink', font=('times', 16, 'italic'))
msg.pack(fill=X, expand=YES)
mainloop()
```

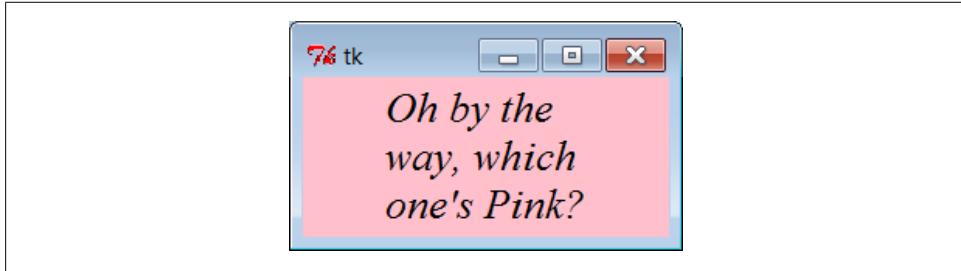


Figure 8-21. A Message widget at work

Entry

The `Entry` widget is a simple, single-line text input field. It is typically used for input fields in form-like dialogs and anywhere else you need the user to type a value into a field of a larger display. `Entry` also supports advanced concepts such as scrolling, key bindings for editing, and text selections, but it's simple to use in practice. [Example 8-17](#) builds the input window shown in [Figure 8-22](#).

Example 8-17. PP4E\Gui\tour\entry1.py

```
from tkinter import *
from quitter import Quitter

def fetch():
    print('Input => "%s"' % ent.get())          # get text

root = Tk()
ent = Entry(root)
ent.insert(0, 'Type words here')                  # set text
ent.pack(side=TOP, fill=X)                         # grow horiz

ent.focus()                                         # save a click
ent.bind('<Return>', (lambda event: fetch()))    # on enter key
btn = Button(root, text='Fetch', command=fetch)     # and on button
btn.pack(side=LEFT)
Quitter(root).pack(side=RIGHT)
root.mainloop()
```

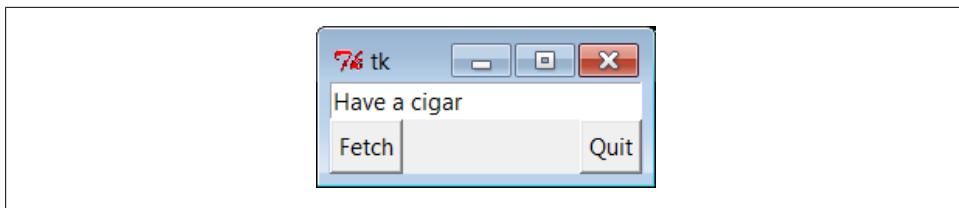


Figure 8-22. `entry1` caught in the act

On startup, the `entry1` script fills the input field in this GUI with the text “Type words here” by calling the widget’s `insert` method. Because both the Fetch button and the Enter key are set to trigger the script’s `fetch` callback function, either user event gets and displays the current text in the input field, using the widget’s `get` method:

```
C:\...\PP4E\Gui\Tour> python entry1.py
Input => "Type words here"
Input => "Have a cigar"
```

We met the `<Return>` event earlier when we studied `bind`; unlike button presses, these lower-level callbacks get an event argument, so the script uses a lambda wrapper to ignore it. This script also packs the entry field with `fill=X` to make it expand horizontally with the window (try it out), and it calls the widget `focus` method to give the entry field input focus when the window first appears. Manually setting the focus like this saves the user from having to click the input field before typing. Our smart Quit button we wrote earlier is attached here again as well (it verifies exit).

Programming Entry widgets

Generally speaking, the values typed into and displayed by `Entry` widgets are set and fetched with either tied “variable” objects (described later in this chapter) or `Entry` widget method calls such as this one:

```
ent.insert(0, 'some text')      # set value
value = ent.get()              # fetch value (a string)
```

The first parameter to the `insert` method gives the position where the text is to be inserted. Here, “0” means the front because offsets start at zero, and integer `0` and string `'0'` mean the same thing (Tkinter method arguments are always converted to strings if needed). If the `Entry` widget might already contain text, you also generally need to delete its contents before setting it to a new value, or else new text will simply be added to the text already present:

```
ent.delete(0, END)            # first, delete from start to end
ent.insert(0, 'some text')     # then set value
```

The name `END` here is a preassigned Tkinter constant denoting the end of the widget; we’ll revisit it in [Chapter 9](#) when we meet the full-blown and multiple-line `Text` widget (`Entry`’s more powerful cousin). Since the widget is empty after the deletion, this statement sequence is equivalent to the prior one:

```

ent.delete('0', END)           # delete from start to end
ent.insert(END, 'some text')    # add at end of empty text

```

Either way, if you don't delete the text first, new text that is inserted is simply added. If you want to see how, try changing the `fetch` function in [Example 8-17](#) to look like this—an "x" is added at the beginning and end of the input field on each button or key press:

```

def fetch():
    print('Input => "%s"' % ent.get())      # get text
    ent.insert(END, 'x')                      # to clear: ent.delete('0', END)
    ent.insert(0, 'x')                       # new text simply added

```

In later examples, we'll also see the `Entry` widget's `state='disabled'` option, which makes it read only, as well as its `show='*'` option, which makes it display each character as a * (useful for password-type inputs). Try this out on your own by changing and running this script for a quick look. `Entry` supports other options we'll skip here, too; see later examples and other resources for additional details.

Laying Out Input Forms

As mentioned, `Entry` widgets are often used to get field values in form-like displays. We're going to create such displays often in this book, but to show you how this works in simpler terms, [Example 8-18](#) combines labels, entries, and frames to achieve the multiple-input display captured in [Figure 8-23](#).

Example 8-18. PP4E\Gui\Tour\entry2.py

```

"""
use Entry widgets directly
lay out by rows with fixed-width labels: this and grid are best for forms
"""

from tkinter import *
from quitter import Quitter
fields = 'Name', 'Job', 'Pay'

def fetch(entries):
    for entry in entries:
        print('Input => "%s"' % entry.get())      # get text

def makeform(root, fields):
    entries = []
    for field in fields:
        row = Frame(root)                         # make a new row
        lab = Label(row, width=5, text=field)       # add label, entry
        ent = Entry(row)                          # pack row on top
        row.pack(side=TOP, fill=X)                 # grow horizontal
        lab.pack(side=LEFT)
        ent.pack(side=RIGHT, expand=YES, fill=X)
        entries.append(ent)
    return entries

```

```

if __name__ == '__main__':
    root = Tk()
    ents = makeform(root, fields)
    root.bind('<Return>', (lambda event: fetch(ents)))
    Button(root, text='Fetch',
           command= (lambda: fetch(ents))).pack(side=LEFT)
    Quitter(root).pack(side=RIGHT)
    root.mainloop()

```

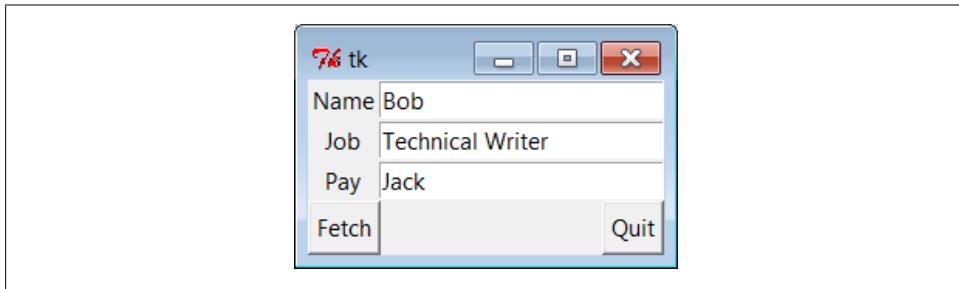


Figure 8-23. *entry2* (and *entry3*) form displays

The input fields here are just simple `Entry` widgets. The script builds an explicit list of these widgets to be used to fetch their values later. Every time you press this window's Fetch button, it grabs the current values in all the input fields and prints them to the standard output stream:

```

C:\...\PP4E\Gui\Tour> python entry2.py
Input => "Bob"
Input => "Technical Writer"
Input => "Jack"

```

You get the same field dump if you press the Enter key anytime this window has the focus on your screen; this event has been bound to the whole root window this time, not to a single input field.

Most of the art in form layout has to do with arranging widgets in a hierarchy. This script builds each label/entry row as a new `Frame` attached to the window's current `TOP`; fixed-width labels are attached to the `LEFT` of their row, and entries to the `RIGHT`. Because each row is a distinct `Frame`, its contents are insulated from other packing going on in this window. The script also arranges for just the entry fields to grow vertically on a resize, as in [Figure 8-24](#).

Going modal again

Later on this tour, we'll see how to make similar form layouts with the `grid` geometry manager, where we arrange by row and column numbers instead of frames. But now that we have a handle on form layout, let's see how to apply the modal dialog techniques we met earlier to a more complex input display.

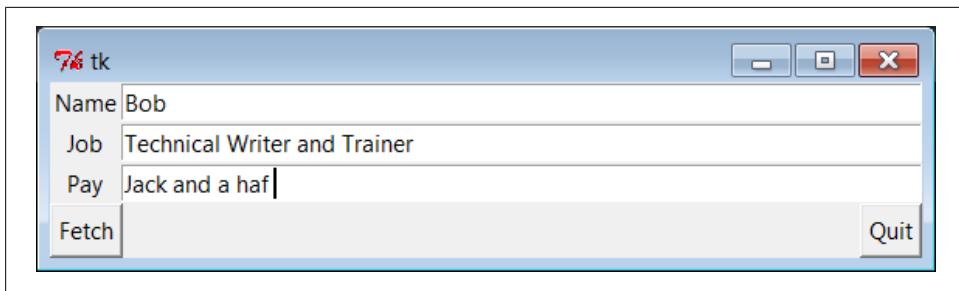


Figure 8-24. `entry2` (and `entry3`) expansion at work

[Example 8-19](#) uses the prior example's `makeform` and `fetch` functions to generate a form and prints its contents, much as before. Here, though, the input fields are attached to a new `Toplevel` pop-up window created on demand, and an OK button is added to the new window to trigger a window destroy event that erases the pop up. As we learned earlier, the `wait_window` call pauses until the destroy happens.

Example 8-19. PP4E\Gui\Tour\entry2-modal.py

```
# make form dialog modal; must fetch before destroy with entries

from tkinter import *
from entry2 import makeform, fetch, fields

def show(entries, popup):
    fetch(entries)                      # must fetch before window destroyed!
    popup.destroy()                     # fails with msgs if stmt order is reversed

def ask():
    popup = Toplevel()                 # show form in modal dialog window
    ents = makeform(popup, fields)
    Button(popup, text='OK', command=(lambda: show(ents, popup))).pack()
    popup.grab_set()
    popup.focus_set()
    popup.wait_window()               # wait for destroy here

root = Tk()
Button(root, text='Dialog', command=ask).pack()
root.mainloop()
```

When you run this code, pressing the button in this program's main window creates the blocking form input dialog in [Figure 8-25](#), as expected.

But a subtle danger is lurking in this modal dialog code: because it fetches user inputs from `Entry` widgets embedded in the popped-up display, it must fetch those inputs *before* destroying the pop-up window in the OK press callback handler. It turns out that a `destroy` call really does destroy all the child widgets of the window destroyed; trying to fetch values from a destroyed `Entry` not only doesn't work, but also generates a traceback with error messages in the console window. Try reversing the statement order in the `show` function to see for yourself.

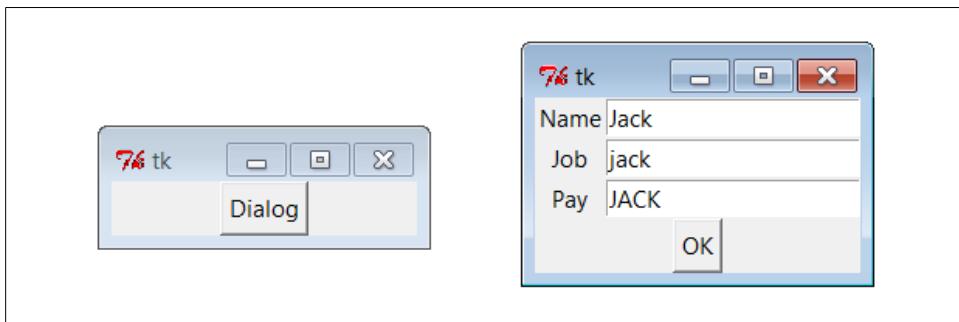


Figure 8-25. *entry2-modal* (and *entry3-modal*) displays

To avoid this problem, we can either be careful to fetch before destroying, or use `tkinter` variables, the subject of the next section.

tkinter “Variables” and Form Layout Alternatives

Entry widgets (among others) support the notion of an associated variable—changing the associated variable changes the text displayed in the `Entry`, and changing the text in the `Entry` changes the value of the variable. These aren’t normal Python variable names, though. Variables tied to widgets are instances of variable classes in the `tkinter` module library. These classes are named `StringVar`, `IntVar`, `DoubleVar`, and `BooleanVar`; you pick one based on the context in which it is to be used. For example, a `StringVar` class instance can be associated with an `Entry` field, as demonstrated in Example 8-20.

Example 8-20. PP4E\Gui\Tour\entry3.py

```
"""
use StringVar variables
lay out by columns: this might not align horizontally everywhere (see entry2)
"""

from tkinter import *
from quitter import Quitter
fields = 'Name', 'Job', 'Pay'

def fetch(variables):
    for variable in variables:
        print('Input => "%s"' % variable.get())      # get from var
```

```

def makeform(root, fields):
    form = Frame(root)                                # make outer frame
    left = Frame(form)                               # make two columns
    rite = Frame(form)
    form.pack(fill=X)
    left.pack(side=LEFT)
    rite.pack(side=RIGHT, expand=YES, fill=X)        # grow horizontal

    variables = []
    for field in fields:
        lab = Label(left, width=5, text=field)       # add to columns
        ent = Entry(rite)
        lab.pack(side=TOP)
        ent.pack(side=TOP, fill=X)                   # grow horizontal
        var = StringVar()
        ent.config(textvariable=var)                 # link field to var
        var.set('enter here')
        variables.append(var)
    return variables

if __name__ == '__main__':
    root = Tk()
    vars = makeform(root, fields)
    Button(root, text='Fetch', command=(lambda: fetch(vars))).pack(side=LEFT)
    Quitter(root).pack(side=RIGHT)
    root.bind('<Return>', (lambda event: fetch(vars)))
    root.mainloop()

```

Except for the fact that this script initializes input fields with the string 'enter here', it makes a window virtually identical in appearance and function to that created by the script `entry2` (see Figures 8-23 and 8-24). For illustration purposes, the window is laid out differently—as a `Frame` containing two nested subframes used to build the left and right columns of the form area—but the end result is the same when it is displayed on screen (for some GUIs on some platforms, at least: see the note at the end of this section for a discussion of why layout by rows instead of columns is generally preferred).

The main thing to notice here, though, is the use of `StringVar` variables. Instead of using a list of `Entry` widgets to fetch input values, this version keeps a list of `StringVar` objects that have been associated with the `Entry` widgets, like this:

```

ent = Entry(rite)
var = StringVar()
ent.config(textvariable=var)                      # link field to var

```

Once you've tied variables in this way, changing and fetching the variable's value:

```

var.set('text here')
value = var.get()

```

will really change and fetch the corresponding display's input field value.* The variable object `get` method returns as a string for `StringVar`, an integer for `IntVar`, and a floating-point number for `DoubleVar`.

Of course, we've already seen that it's easy to set and fetch text in `Entry` fields directly, without adding extra code to use variables. So, why the bother about variable objects? For one thing, it clears up that nasty fetch-after-destroy peril we met in the prior section. Because `StringVars` live on after the `Entry` widgets they are tied to have been destroyed, it's OK to fetch input values from them long after a modal dialog has been dismissed, as shown in [Example 8-21](#).

Example 8-21. PP4E\Gui\Tour\entry3-modal.py

```
# can fetch values after destroy with stringvars

from tkinter import *
from entry3 import makeform, fetch, fields

def show(variables, popup):
    popup.destroy()                      # order doesn't matter here
    fetch(variables)                     # variables live on after window destroyed

def ask():
    popup = Toplevel()                  # show form in modal dialog window
    vars = makeform(popup, fields)
    Button(popup, text='OK', command=(lambda: show(vars, popup))).pack()
    popup.grab_set()
    popup.focus_set()
    popup.wait_window()                # wait for destroy here

root = Tk()
Button(root, text='Dialog', command=ask).pack()
root.mainloop()
```

This version is the same as the original (shown in [Example 8-19](#) and [Figure 8-25](#)), but `show` now destroys the pop up before inputs are fetched through `StringVars` in the list created by `makeform`. In other words, variables are a bit more robust in some contexts because they are not part of a real display tree. For example, they are also commonly associated with check buttons, radio boxes, and scales in order to provide access to current settings and link multiple widgets together. Almost coincidentally, that's the topic of the next section.

* Historic anecdote: In a now-defunct `tkinter` release shipped with Python 1.3, you could also set and fetch variable values by calling them like functions, with and without an argument (e.g., `var(value)` and `var()`). Today, you call variable `set` and `get` methods instead. For unknown reasons, the function call form stopped working years ago, but you may still see it in older Python code (and in first editions of at least one O'Reilly Python book). If a fix made in the name of aesthetics breaks working code, is it really a fix?



We laid out input forms two ways in this section: by *row* frames with fixed-width labels (`entry2`), and by *column* frames (`entry3`). In [Chapter 9](#) we'll see a third form technique: layouts using the `grid` geometry manager. Of these, gridding, and the rows with fixed-width labels of `entry2` tend to work best across all platforms.

Laying out by column frames as in `entry3` works only on platforms where the height of each label exactly matches the height of each entry field. Because the two are not associated directly, they might not line up properly on some platforms. When I tried running some forms that looked fine on Windows XP on a Linux machine, labels and their corresponding entries did not line up horizontally.

Even the simple window produced by `entry3` looks slightly askew on closer inspection. It only appears the same as `entry2` on some platforms because of the small number of inputs and size defaults. On my Windows 7 netbook, the labels and entries start to become horizontally mismatched if you add 3 or 4 additional inputs to `entry3`'s `fields` tuple.

If you care about portability, lay out your forms either with the packed row frames and fixed/maximum-width labels of `entry2`, or by gridding widgets by row and column numbers instead of packing them. We'll see more on such forms in the next chapter. And in [Chapter 12](#), we'll write a form-construction tool that hides the layout details from its clients altogether (including its use case client in [Chapter 13](#)).

Checkbutton, Radiobutton, and Scale

This section introduces three widget types: the `Checkbutton` (a multiple-choice input widget), the `Radiobutton` (a single-choice device), and the `Scale` (sometimes known as a “slider”). All are variations on a theme and are somewhat related to simple buttons, so we'll explore them as a group here. To make these widgets more fun to play with, we'll reuse the `dialogTable` module shown in [Example 8-8](#) to provide callbacks for widget selections (callbacks pop up dialog boxes). Along the way, we'll also use the `tkinter` variables we just met to communicate with these widgets' state settings.

Checkbuttons

The `Checkbutton` and `Radiobutton` widgets are designed to be associated with `tkinter` variables: clicking the button changes the value of the variable, and setting the variable changes the state of the button to which it is linked. In fact, `tkinter` variables are central to the operation of these widgets:

- A collection of `Checkbuttons` implements a multiple-choice interface by assigning each button a variable of its own.
- A collection of `Radiobuttons` imposes a mutually exclusive single-choice model by giving each button a unique value and the same `tkinter` variable.

Both kinds of buttons provide both `command` and `variable` options. The `command` option lets you register a callback to be run immediately on button-press events, much like normal `Button` widgets. But by associating a tkinter variable with the `variable` option, you can also fetch or change widget state at any time by fetching or changing the value of the widget's associated variable.

Since it's a bit simpler, let's start with the tkinter `Checkbutton`. Example 8-22 creates the set of five captured in Figure 8-26. To make this more useful, it also adds a button that dumps the current state of all `Checkbutton`s and attaches an instance of the verifying `Quitter` button we built earlier in the tour.

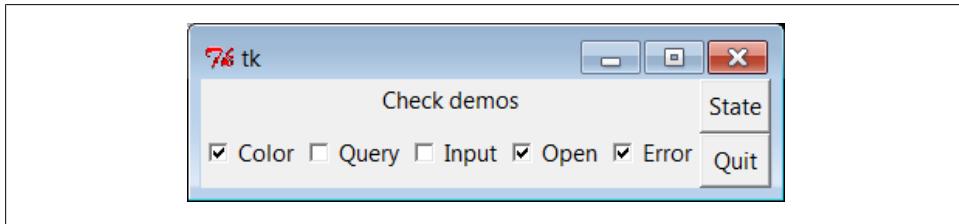


Figure 8-26. *demoCheck* in action

Example 8-22. PP4E\Gui\Tour\demoCheck.py

```
"create a bar of check buttons that run dialog demos"

from tkinter import *           # get base widget set
from dialogTable import demos    # get canned dialogs
from quitter import Quitter      # attach a quitter object to "me"

class Demo(Frame):
    def __init__(self, parent=None, **options):
        Frame.__init__(self, parent, **options)
        self.pack()
        self.tools()
        Label(self, text="Check demos").pack()
        self.vars = []
        for key in demos:
            var = IntVar()
            Checkbutton(self,
                         text=key,
                         variable=var,
                         command=demos[key]).pack(side=LEFT)
            self.vars.append(var)

    def report(self):
        for var in self.vars:
            print(var.get(), end=' ')  # current toggle settings: 1 or 0
        print()

    def tools(self):
        frm = Frame(self)
        frm.pack(side=RIGHT)
```

```
        Button(frm, text='State', command=self.report).pack(fill=X)
        Quitter(frm).pack(fill=X)

if __name__ == '__main__': Demo().mainloop()
```

In terms of program code, check buttons resemble normal buttons; they are even packed within a container widget. Operationally, though, they are a bit different. As you can probably tell from this figure (and can better tell by running this live), a check button works as a toggle—pressing one changes its state from off to on (from deselected to selected); or from on to off again. When a check button is selected, it has a checked display, and its associated `IntVar` variable has a value of 1; when deselected, its display is empty and its `IntVar` has a value of 0.

To simulate an enclosing application, the `State` button in this display triggers the script’s `report` method to display the current values of all five toggles on the `stdout` stream. Here is the output after a few clicks:

```
C:\...\PP4E\Gui\Tour> python demoCheck.py
0 0 0 0 0
1 0 0 0 0
1 0 1 0 0
1 0 1 1 0
1 0 0 1 0
1 0 0 1 1
```

Really, these are the values of the five tkinter variables associated with the `Checkbuttons` with `variable` options, but they give the buttons’ values when queried. This script associates `IntVar` variables with each `Checkbutton` in this display, since they are 0 or 1 binary indicators. `StringVars` will work here, too, although their `get` methods would return strings ‘0’ or ‘1’ (not integers) and their initial state would be an empty string (not the integer 0).

This widget’s `command` option lets you register a callback to be run each time the button is pressed. To illustrate, this script registers a standard dialog `demo` call as a handler for each of the `Checkbuttons`—pressing a button changes the toggle’s state but also pops up one of the dialog windows we visited earlier in this tour (regardless of its new state).

Interestingly, you can sometimes run the `report` method interactively, too—when working as follows in a shell window, widgets pop up as lines are typed and are fully active, even without calling `mainloop` (though this may not work in some interfaces like IDLE if you must call `mainloop` to display your GUI):

```
C:\...\PP4E\Gui\Tour> python
>>> from demoCheck import Demo
>>> d = Demo()
>>> d.report()
0 0 0 0 0
>>> d.report()
1 0 0 0 0
>>> d.report()
1 0 0 1 1
```

Check buttons and variables

When I first studied check buttons, my initial reaction was: why do we need tkinter variables here at all when we can register button-press callbacks? Linked variables may seem superfluous at first glance, but they simplify some GUI chores. Instead of asking you to accept this blindly, though, let me explain why.

Keep in mind that a `Checkbutton`'s `command` callback will be run on every press, whether the press toggles the check button to a selected or a deselected state. Because of that, if you want to run an action immediately when a check button is pressed, you will generally want to check the button's current value in the callback handler. Because there is no check button "get" method for fetching values, you usually need to interrogate an associated variable to see if the button is on or off.

Moreover, some GUIs simply let users set check buttons without running `command` callbacks at all and fetch button settings at some later point in the program. In such a scenario, variables serve to automatically keep track of button settings. The `demo` Check script's `report` method represents this latter approach.

Of course, you could manually keep track of each button's state in press callback handlers, too. [Example 8-23](#) keeps its own list of state toggles and updates it manually on `command` press callbacks.

Example 8-23. PP4E\Gui\Tour\demo-check-manual.py

```
# check buttons, the hard way (without variables)

from tkinter import *
states = []                                # change object not name
def onPress(i):                            # keep track of states
    states[i] = not states[i]                # changes False->True, True->False

root = Tk()
for i in range(10):
    chk = Checkbutton(root, text=str(i), command=(lambda i=i: onPress(i)) )
    chk.pack(side=LEFT)
    states.append(False)
root.mainloop()                           # show all states on exit
print(states)
```

The lambda here passes along the pressed button's index in the `states` list. Otherwise, we would need a separate callback function for each button. Here again, we need to use a *default argument* to pass the loop variable into the lambda, or the loop variable will be its value on the last loop iteration for all 10 of the generated functions (each press would update the tenth item in the list; see [Chapter 7](#) for background details on this). When run, this script makes the 10–check button display in [Figure 8-27](#).



Figure 8-27. Manual check button state window

Manually maintained state toggles are updated on every button press and are printed when the GUI exits (technically, when the `mainloop` call returns); it's a list of Boolean state values, which could also be integers 1 or 0 if we cared to exactly imitate the original:

```
C:\...\PP4E\Gui\Tour> python demo-check-manual.py
[False, False, True, False, True, False, False, True, False]
```

This works, and it isn't too horribly difficult to manage manually. But linked tkinter variables make this task noticeably easier, especially if you don't need to process check button states until some time in the future. This is illustrated in [Example 8-24](#).

Example 8-24. PP4E\Gui\Tour\demo-check-auto.py

```
# check buttons, the easy way

from tkinter import *
root = Tk()
states = []
for i in range(10):
    var = IntVar()
    chk = Checkbutton(root, text=str(i), variable=var)
    chk.pack(side=LEFT)
    states.append(var)
root.mainloop()                      # let tkinter keep track
print([var.get() for var in states])  # show all states on exit (or map/lambda)
```

This looks and works the same way, but there is no `command` button-press callback handler at all, because toggle state is tracked by tkinter automatically:

```
C:\...\PP4E\Gui\Tour> python demo-check-auto.py
[0, 0, 1, 1, 0, 0, 1, 0, 0, 1]
```

The point here is that you don't necessarily have to link variables with check buttons, but your GUI life will be simpler if you do. The list comprehension at the very end of this script, by the way, is equivalent to the following unbound method and lambda/bound-method `map` call forms:

```
print(list(map(IntVar.get, states)))
print(list(map(lambda var: var.get(), states)))
```

Though comprehensions are common in Python today, the form that seems clearest to you may very well depend upon your shoe size...

Radio Buttons

Radio buttons are toggles too, but they are generally used in groups: just like the mechanical station selector pushbuttons on radios of times gone by, pressing one Radio button widget in a group automatically deselects the one pressed last. In other words, at most, only one can be selected at one time. In tkinter, associating all radio buttons in a group with unique values and the same variable guarantees that, at most, only one can ever be selected at a given time.

Like check buttons and normal buttons, radio buttons support a `command` option for registering a callback to handle presses immediately. Like check buttons, radio buttons also have a `variable` attribute for associating single-selection buttons in a group and fetching the current selection at arbitrary times.

In addition, radio buttons have a `value` attribute that lets you tell tkinter what value the button's associated variable should have when the button is selected. Because more than one radio button is associated with the same variable, you need to be explicit about each button's value (it's not just a 1 or 0 toggle scenario). [Example 8-25](#) demonstrates radio button basics.

Example 8-25. PP4E\Gui\Tour\demoRadio.py

```
"create a group of radio buttons that launch dialog demos"

from tkinter import *                      # get base widget set
from dialogTable import demos               # button callback handlers
from quitter import Quitter                 # attach a quit object to "me"

class Demo(Frame):
    def __init__(self, parent=None, **options):
        Frame.__init__(self, parent, **options)
        self.pack()
        Label(self, text="Radio demos").pack(side=TOP)
        self.var = StringVar()
        for key in demos:
            Radiobutton(self, text=key,
                        command=self.onPress,
                        variable=self.var,
                        value=key).pack(anchor=NW)
        self.var.set(key) # select last to start
        Button(self, text='State', command=self.report).pack(fill=X)
        Quitter(self).pack(fill=X)

    def onPress(self):
        pick = self.var.get()
        print('you pressed', pick)
        print('result:', demos[pick]())

    def report(self):
        print(self.var.get())

if __name__ == '__main__': Demo().mainloop()
```

Figure 8-28 shows what this script generates when run. Pressing any of this window’s radio buttons triggers its `command` handler, pops up one of the standard dialog boxes we met earlier, and automatically deselects the button previously pressed. Like check buttons, radio buttons are packed; this script packs them to the top to arrange them vertically, and then anchors each on the northwest corner of its allocated space so that they align well.

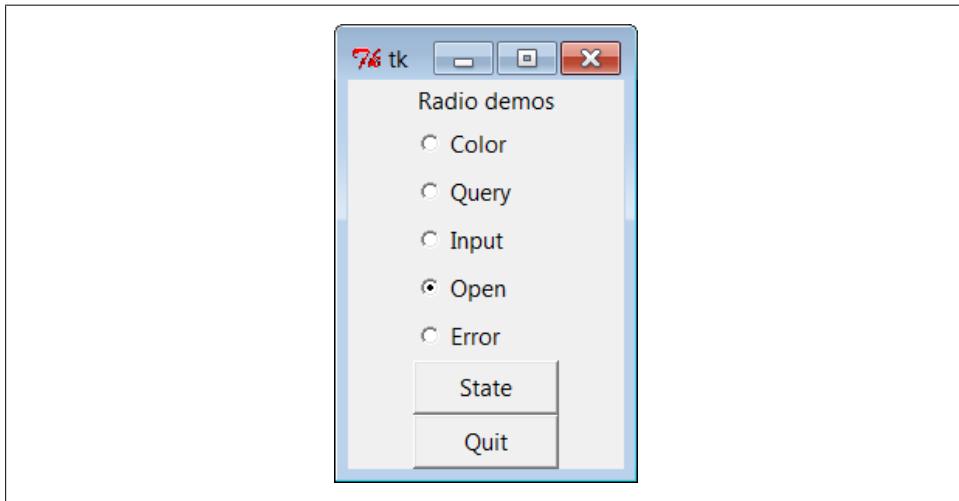


Figure 8-28. *demoRadio* in action

Like the check button demo script, this one also puts up a `State` button to run the class’s `report` method and to show the current radio state (the button selected). Unlike the check button demo, this script also prints the return values of dialog demo calls that are run as its buttons are pressed. Here is what the `stdout` stream looks like after a few presses and state dumps; states are shown in bold:

```
C:\...\PP4E\Gui\Tour> python demoRadio.py
you pressed Input
result: 3.14
Input
you pressed Open
result: C:/PP4thEd/Examples/PP4E/Gui/Tour/demoRadio.py
Open
you pressed Query
result: yes
Query
```

Radio buttons and variables

So, why variables here? For one thing, radio buttons also have no “get” widget method to fetch the selection in the future. More importantly, in radio button groups, the `value` and `variable` settings turn out to be the whole basis of single-choice behavior. In

fact, to make radio buttons work normally at all, it's crucial that they are all associated with the same tkinter variable and have distinct value settings. To truly understand why, though, you need to know a bit more about how radio buttons and variables do their stuff.

We've already seen that changing a widget changes its associated tkinter variable, and vice versa. But it's also true that changing a variable in any way automatically changes every widget it is associated with. In the world of radio buttons, pressing a button sets a shared variable, which in turn impacts other buttons associated with that variable. Assuming that all radio buttons have distinct values, this works as you expect it to work. When a button press changes the shared variable to the pressed button's value, all other buttons are deselected, simply because the variable has been changed to a value not their own.

This is true both when the user selects a button and changes the shared variable's value implicitly, but also when the variable's value is set manually by a script. For instance, when [Example 8-25](#) sets the shared variable to the last of the demo's names initially (with `self.var.set`), it selects that demo's button and deselects all the others in the process; this way, only one is selected at first. If the variable was instead set to a string that is not any demo's name (e.g., ' '), all buttons would be deselected at startup.

This ripple effect is a bit subtle, but it might help to know that within a group of radio buttons sharing the same variable, if you assign a set of buttons the same value, the entire set will be selected if any one of them is pressed. Consider [Example 8-26](#), which creates [Figure 8-29](#), for instance. All buttons start out deselected this time (by initializing the shared variable to none of their values), but because radio buttons 0, 3, 6, and 9 have value 0 (the remainder of division by 3), all are selected if any are selected.



Figure 8-29. Radio buttons gone bad?

[Example 8-26. PP4E\Gui\Tour\demo-radio-multi.py](#)

```
# see what happens when some buttons have same value

from tkinter import *
root = Tk()
var = StringVar()
for i in range(10):
    rad = Radiobutton(root, text=str(i), variable=var, value=str(i % 3))
    rad.pack(side=LEFT)
var.set(' ') # deselect all initially
root.mainloop()
```

If you press 1, 4, or 7 now, all three of these are selected, and any existing selections are cleared (they don't have the value "1"). That's not normally what you want—radio buttons are usually a single-choice group (check buttons handle multiple-choice inputs). If you want them to work as expected, be sure to give each radio button the same variable but a unique value across the entire group. In the `demoRadio` script, for instance, the name of the demo provides a naturally unique value for each button.

Radio buttons without variables

Strictly speaking, we could get by without tkinter variables here, too. [Example 8-27](#), for instance, implements a single-selection model without variables, by manually selecting and deselecting widgets in the group, in a callback handler of its own. On each press event, it issues `deselect` calls for every widget object in the group and `select` for the one pressed.

Example 8-27. PP4E\Gui\Tour\demo-radio-manual.py

```
"""
radio buttons, the hard way (without variables)
note that deselect for radio buttons simply sets the button's
associated value to a null string, so we either need to still
give buttons unique values, or use checkbuttons here instead;
"""

from tkinter import *
state = ''
buttons = []

def onPress(i):
    global state
    state = i
    for btn in buttons:
        btn.deselect()
    buttons[i].select()

root = Tk()
for i in range(10):
    rad = Radiobutton(root, text=str(i),
                      value=str(i), command=(lambda i=i: onPress(i)))
    rad.pack(side=LEFT)
    buttons.append(rad)

onPress(0)           # select first initially
root.mainloop()
print(state)         # show state on exit
```

This works. It creates a 10-radio button window that looks just like the one in [Figure 8-29](#) but implements a single-choice radio-style interface, with current state available in a global Python variable printed on script exit. By associating tkinter variables and unique values, though, you can let tkinter do all this work for you, as shown in [Example 8-28](#).