The Python Standard Library by Example
The Python Standard Library by Example
The Developer’s Library Series from Addison-Wesley provides practicing programmers with unique, high-quality references and tutorials on the latest programming languages and technologies they use in their daily work. All books in the Developer’s Library are written by expert technology practitioners who are exceptionally skilled at organizing and presenting information in a way that’s useful for other programmers.

Developer’s Library books cover a wide range of topics, from open-source programming languages and databases, Linux programming, Microsoft, and Java, to Web development, social networking platforms, Mac/iPhone programming, and Android programming.

Visit developers-library.com for a complete list of available products.
This book is dedicated to my wife, Theresa,
for everything she has done for me.
CONTENTS AT A GLANCE

Contents ix
Tables xxxi
Foreword xxxiii
Acknowledgments xxxvii
About the Author xxxix

INTRODUCTION 1

1 TEXT 3

2 DATA STRUCTURES 69

3 ALGORITHMS 129

4 DATES AND TIMES 173

5 MATHEMATICS 197

6 THE FILE SYSTEM 247

7 DATA PERSISTENCE AND EXCHANGE 333

8 DATA COMPRESSION AND ARCHIVING 421

9 CRYPTOGRAPHY 469
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>PROCESSES AND THREADS</td>
<td>481</td>
</tr>
<tr>
<td>11</td>
<td>NETWORKING</td>
<td>561</td>
</tr>
<tr>
<td>12</td>
<td>THE INTERNET</td>
<td>637</td>
</tr>
<tr>
<td>13</td>
<td>EMAIL</td>
<td>727</td>
</tr>
<tr>
<td>14</td>
<td>APPLICATION BUILDING BLOCKS</td>
<td>769</td>
</tr>
<tr>
<td>15</td>
<td>INTERNATIONALIZATION AND LOCALIZATION</td>
<td>899</td>
</tr>
<tr>
<td>16</td>
<td>DEVELOPER TOOLS</td>
<td>919</td>
</tr>
<tr>
<td>17</td>
<td>RUNTIME FEATURES</td>
<td>1045</td>
</tr>
<tr>
<td>18</td>
<td>LANGUAGE TOOLS</td>
<td>1169</td>
</tr>
<tr>
<td>19</td>
<td>MODULES AND PACKAGES</td>
<td>1235</td>
</tr>
</tbody>
</table>

*Index of Python Modules*  1259

*Index*  1261
CONTENTS

Tables xxxi
Foreword xxxiii
Acknowledgments xxxvii
About the Author xxxix

INTRODUCTION 1

1 TEXT 3
  1.1 string—Text Constants and Templates 4
    1.1.1 Functions 4
    1.1.2 Templates 5
    1.1.3 Advanced Templates 7
  1.2 textwrap—Formatting Text Paragraphs 9
    1.2.1 Example Data 9
    1.2.2 Filling Paragraphs 10
    1.2.3 Removing Existing Indentation 10
    1.2.4 Combining Dedent and Fill 11
    1.2.5 Hanging Indents 12
  1.3 re—Regular Expressions 13
    1.3.1 Finding Patterns in Text 14
    1.3.2 Compiling Expressions 14
    1.3.3 Multiple Matches 15
    1.3.4 Pattern Syntax 16
    1.3.5 Constraining the Search 28
    1.3.6 Dissecting Matches with Groups 30
2 DATA STRUCTURES

2.1 collections—Container Data Types
   2.1.1 Counter
   2.1.2 defaultdict
   2.1.3 Deque
   2.1.4 namedtuple
   2.1.5 OrderedDict

2.2 array—Sequence of Fixed-Type Data
   2.2.1 Initialization
   2.2.2 Manipulating Arrays
   2.2.3 Arrays and Files
   2.2.4 Alternate Byte Ordering

2.3 heapq—Heap Sort Algorithm
   2.3.1 Example Data
   2.3.2 Creating a Heap
   2.3.3 Accessing Contents of a Heap
   2.3.4 Data Extremes from a Heap

2.4 bisect—Maintain Lists in Sorted Order
   2.4.1 Inserting in Sorted Order
   2.4.2 Handling Duplicates

2.5 Queue—Thread-Safe FIFO Implementation
   2.5.1 Basic FIFO Queue
   2.5.2 LIFO Queue
   2.5.3 Priority Queue
   2.5.4 Building a Threaded Podcast Client

2.6 struct—Binary Data Structures
   2.6.1 Functions vs. Struct Class
   2.6.2 Packing and Unpacking
2.6.3 Endianness 103
2.6.4 Buffers 105
2.7 weakref—Impermanent References to Objects 106
  2.7.1 References 107
  2.7.2 Reference Callbacks 108
  2.7.3 Proxies 108
  2.7.4 Cyclic References 109
  2.7.5 Caching Objects 114
2.8 copy—Duplicate Objects 117
  2.8.1 Shallow Copies 118
  2.8.2 Deep Copies 118
  2.8.3 Customizing Copy Behavior 119
  2.8.4 Recursion in Deep Copy 120
2.9 pprint—Pretty-Print Data Structures 123
  2.9.1 Printing 123
  2.9.2 Formatting 124
  2.9.3 Arbitrary Classes 125
  2.9.4 Recursion 125
  2.9.5 Limiting Nested Output 126
  2.9.6 Controlling Output Width 126

3 ALGORITHMS 129
  3.1 functools—Tools for Manipulating Functions 129
    3.1.1 Decorators 130
    3.1.2 Comparison 138
  3.2 itertools—Iterator Functions 141
    3.2.1 Merging and Splitting Iterators 142
    3.2.2 Converting Inputs 145
    3.2.3 Producing New Values 146
    3.2.4 Filtering 148
    3.2.5 Grouping Data 151
  3.3 operator—Functional Interface to Built-in Operators 153
    3.3.1 Logical Operations 154
    3.3.2 Comparison Operators 154
    3.3.3 Arithmetic Operators 155
    3.3.4 Sequence Operators 157
    3.3.5 In-Place Operators 158
    3.3.6 Attribute and Item “Getters” 159
    3.3.7 Combining Operators and Custom Classes 161
5.3.2 Seeding 212
5.3.3 Saving State 213
5.3.4 Random Integers 214
5.3.5 Picking Random Items 215
5.3.6 Permutations 216
5.3.7 Sampling 218
5.3.8 Multiple Simultaneous Generators 219
5.3.9 SystemRandom 221
5.3.10 Nonuniform Distributions 222
5.4 math—Mathematical Functions 223
5.4.1 Special Constants 223
5.4.2 Testing for Exceptional Values 224
5.4.3 Converting to Integers 226
5.4.4 Alternate Representations 227
5.4.5 Positive and Negative Signs 229
5.4.6 Commonly Used Calculations 230
5.4.7 Exponents and Logarithms 234
5.4.8 Angles 238
5.4.9 Trigonometry 240
5.4.10 Hyperbolic Functions 243
5.4.11 Special Functions 244

6 THE FILE SYSTEM 247
6.1 os.path—Platform-Independent Manipulation of Filenames 248
6.1.1 Parsing Paths 248
6.1.2 Building Paths 252
6.1.3 Normalizing Paths 253
6.1.4 File Times 254
6.1.5 Testing Files 255
6.1.6 Traversing a Directory Tree 256
6.2 glob—Filename Pattern Matching 257
6.2.1 Example Data 258
6.2.2 Wildcards 258
6.2.3 Single Character Wildcard 259
6.2.4 Character Ranges 260
6.3 linecache—Read Text Files Efficiently 261
6.3.1 Test Data 261
6.3.2 Reading Specific Lines 262
6.3.3 Handling Blank Lines 263
<table>
<thead>
<tr>
<th>Section</th>
<th>Module</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.3.4</td>
<td>Error Handling</td>
<td>Error Handling</td>
<td>263</td>
</tr>
<tr>
<td>6.3.5</td>
<td>Reading Python Source Files</td>
<td>Reading Python Source Files</td>
<td>264</td>
</tr>
<tr>
<td>6.4</td>
<td>tempfile—Temporary File System Objects</td>
<td>Temporary File System Objects</td>
<td>265</td>
</tr>
<tr>
<td>6.4.1</td>
<td>Temporary Files</td>
<td>Temporary Files</td>
<td>265</td>
</tr>
<tr>
<td>6.4.2</td>
<td>Named Files</td>
<td>Named Files</td>
<td>268</td>
</tr>
<tr>
<td>6.4.3</td>
<td>Temporary Directories</td>
<td>Temporary Directories</td>
<td>268</td>
</tr>
<tr>
<td>6.4.4</td>
<td>Predicting Names</td>
<td>Predicting Names</td>
<td>269</td>
</tr>
<tr>
<td>6.4.5</td>
<td>Temporary File Location</td>
<td>Temporary File Location</td>
<td>270</td>
</tr>
<tr>
<td>6.5</td>
<td>shutil—High-Level File Operations</td>
<td>High-Level File Operations</td>
<td>271</td>
</tr>
<tr>
<td>6.5.1</td>
<td>Copying Files</td>
<td>Copying Files</td>
<td>271</td>
</tr>
<tr>
<td>6.5.2</td>
<td>Copying File Metadata</td>
<td>Copying File Metadata</td>
<td>274</td>
</tr>
<tr>
<td>6.5.3</td>
<td>Working with Directory Trees</td>
<td>Working with Directory Trees</td>
<td>276</td>
</tr>
<tr>
<td>6.6</td>
<td>mmap—Memory-Map Files</td>
<td>Memory-Map Files</td>
<td>279</td>
</tr>
<tr>
<td>6.6.1</td>
<td>Reading</td>
<td>Reading</td>
<td>279</td>
</tr>
<tr>
<td>6.6.2</td>
<td>Writing</td>
<td>Writing</td>
<td>280</td>
</tr>
<tr>
<td>6.6.3</td>
<td>Regular Expressions</td>
<td>Regular Expressions</td>
<td>283</td>
</tr>
<tr>
<td>6.7</td>
<td>codecs—String Encoding and Decoding</td>
<td>String Encoding and Decoding</td>
<td>284</td>
</tr>
<tr>
<td>6.7.1</td>
<td>Unicode Primer</td>
<td>Unicode Primer</td>
<td>284</td>
</tr>
<tr>
<td>6.7.2</td>
<td>Working with Files</td>
<td>Working with Files</td>
<td>287</td>
</tr>
<tr>
<td>6.7.3</td>
<td>Byte Order</td>
<td>Byte Order</td>
<td>289</td>
</tr>
<tr>
<td>6.7.4</td>
<td>Error Handling</td>
<td>Error Handling</td>
<td>291</td>
</tr>
<tr>
<td>6.7.5</td>
<td>Standard Input and Output Streams</td>
<td>Standard Input and Output Streams</td>
<td>295</td>
</tr>
<tr>
<td>6.7.6</td>
<td>Encoding Translation</td>
<td>Encoding Translation</td>
<td>298</td>
</tr>
<tr>
<td>6.7.7</td>
<td>Non-Unicode Encodings</td>
<td>Non-Unicode Encodings</td>
<td>300</td>
</tr>
<tr>
<td>6.7.8</td>
<td>Incremental Encoding</td>
<td>Incremental Encoding</td>
<td>301</td>
</tr>
<tr>
<td>6.7.9</td>
<td>Unicode Data and Network Communication</td>
<td>Unicode Data and Network Communication</td>
<td>303</td>
</tr>
<tr>
<td>6.7.10</td>
<td>Defining a Custom Encoding</td>
<td>Defining a Custom Encoding</td>
<td>307</td>
</tr>
<tr>
<td>6.8</td>
<td>StringIO—Text Buffers with a File-like API</td>
<td>Text Buffers with a File-like API</td>
<td>314</td>
</tr>
<tr>
<td>6.8.1</td>
<td>Examples</td>
<td>Examples</td>
<td>314</td>
</tr>
<tr>
<td>6.9</td>
<td>fnmatch—UNIX-Style Glob Pattern Matching</td>
<td>UNIX-Style Glob Pattern Matching</td>
<td>315</td>
</tr>
<tr>
<td>6.9.1</td>
<td>Simple Matching</td>
<td>Simple Matching</td>
<td>315</td>
</tr>
<tr>
<td>6.9.2</td>
<td>Filtering</td>
<td>Filtering</td>
<td>317</td>
</tr>
<tr>
<td>6.9.3</td>
<td>Translating Patterns</td>
<td>Translating Patterns</td>
<td>318</td>
</tr>
<tr>
<td>6.10</td>
<td>dircache—Cache Directory Listings</td>
<td>Cache Directory Listings</td>
<td>319</td>
</tr>
<tr>
<td>6.10.1</td>
<td>Listing Directory Contents</td>
<td>Listing Directory Contents</td>
<td>319</td>
</tr>
<tr>
<td>6.10.2</td>
<td>Annotated Listings</td>
<td>Annotated Listings</td>
<td>321</td>
</tr>
<tr>
<td>6.11</td>
<td>filecmp—Compare Files</td>
<td>Compare Files</td>
<td>322</td>
</tr>
<tr>
<td>6.11.1</td>
<td>Example Data</td>
<td>Example Data</td>
<td>323</td>
</tr>
<tr>
<td>6.11.2</td>
<td>Comparing Files</td>
<td>Comparing Files</td>
<td>325</td>
</tr>
</tbody>
</table>
6.11.3 Comparing Directories 327
6.11.4 Using Differences in a Program 328

7 DATA PERSISTENCE AND EXCHANGE 333
7.1 pickle—Object Serialization 334
7.1.1 Importing 335
7.1.2 Encoding and Decoding Data in Strings 335
7.1.3 Working with Streams 336
7.1.4 Problems Reconstructing Objects 338
7.1.5 Unpicklable Objects 340
7.1.6 Circular References 340
7.2 shelf—Persistent Storage of Objects 343
7.2.1 Creating a New Shelf 343
7.2.2 Writeback 344
7.2.3 Specific Shelf Types 346
7.3 anydbm—DBM-Style Databases 347
7.3.1 Database Types 347
7.3.2 Creating a New Database 348
7.3.3 Opening an Existing Database 349
7.3.4 Error Cases 349
7.4 whichdb—Identify DBM-Style Database Formats 350
7.5 sqlite3—Embedded Relational Database 351
7.5.1 Creating a Database 352
7.5.2 Retrieving Data 355
7.5.3 Query Metadata 357
7.5.4 Row Objects 358
7.5.5 Using Variables with Queries 359
7.5.6 Bulk Loading 362
7.5.7 Defining New Column Types 363
7.5.8 Determining Types for Columns 366
7.5.9 Transactions 368
7.5.10 Isolation Levels 372
7.5.11 In-Memory Databases 376
7.5.12 Exporting the Contents of a Database 376
7.5.13 Using Python Functions in SQL 378
7.5.14 Custom Aggregation 380
7.5.15 Custom Sorting 381
7.5.16 Threading and Connection Sharing 383
7.5.17 Restricting Access to Data 384
7.6  xml.etree.ElementTree—XML Manipulation API 387
  7.6.1  Parsing an XML Document 387
  7.6.2  Traversing the Parsed Tree 388
  7.6.3  Finding Nodes in a Document 390
  7.6.4  Parsed Node Attributes 391
  7.6.5  Watching Events While Parsing 393
  7.6.6  Creating a Custom Tree Builder 396
  7.6.7  Parsing Strings 398
  7.6.8  Building Documents with Element Nodes 400
  7.6.9  Pretty-Printing XML 401
  7.6.10 Setting Element Properties 403
  7.6.11 Building Trees from Lists of Nodes 405
  7.6.12 Serializing XML to a Stream 408

7.7  csv—Comma-Separated Value Files 411
  7.7.1  Reading 411
  7.7.2  Writing 412
  7.7.3  Dialects 413
  7.7.4  Using Field Names 418

8  DATA COMPRESSION AND ARCHIVING 421
  8.1  zlib—GNU zlib Compression 421
    8.1.1  Working with Data in Memory 422
    8.1.2  Incremental Compression and Decompression 423
    8.1.3  Mixed Content Streams 424
    8.1.4  Checksums 425
    8.1.5  Compressing Network Data 426
  8.2  gzip—Read and Write GNU Zip Files 430
    8.2.1  Writing Compressed Files 431
    8.2.2  Reading Compressed Data 433
    8.2.3  Working with Streams 434
  8.3  bz2—bzip2 Compression 436
    8.3.1  One-Shot Operations in Memory 436
    8.3.2  Incremental Compression and Decompression 438
    8.3.3  Mixed Content Streams 439
    8.3.4  Writing Compressed Files 440
    8.3.5  Reading Compressed Files 442
    8.3.6  Compressing Network Data 443
  8.4  tarfile—Tar Archive Access 448
    8.4.1  Testing Tar Files 448
## 10.2 signal—Asynchronous System Events

- 10.2.1 Receiving Signals
- 10.2.2 Retrieving Registered Handlers
- 10.2.3 Sending Signals
- 10.2.4 Alarms
- 10.2.5 Ignoring Signals
- 10.2.6 Signals and Threads

## 10.3 threading—Manage Concurrent Operations

- 10.3.1 Thread Objects
- 10.3.2 Determining the Current Thread
- 10.3.3 Daemon vs. Non-Daemon Threads
- 10.3.4 Enumerating All Threads
- 10.3.5 Subclassing Thread
- 10.3.6 Timer Threads
- 10.3.7 Signaling between Threads
- 10.3.8 Controlling Access to Resources
- 10.3.9 Synchronizing Threads
- 10.3.10 Limiting Concurrent Access to Resources
- 10.3.11 Thread-Specific Data

## 10.4 multiprocessing—Manage Processes like Threads

- 10.4.1 Multiprocessing Basics
- 10.4.2 Importable Target Functions
- 10.4.3 Determining the Current Process
- 10.4.4 Daemon Processes
- 10.4.5 Waiting for Processes
- 10.4.6 Terminating Processes
- 10.4.7 Process Exit Status
- 10.4.8 Logging
- 10.4.9 Subclassing Process
- 10.4.10 Passing Messages to Processes
- 10.4.11 Signaling between Processes
- 10.4.12 Controlling Access to Resources
- 10.4.13 Synchronizing Operations
- 10.4.14 Controlling Concurrent Access to Resources
- 10.4.15 Managing Shared State
- 10.4.16 Shared Namespaces
- 10.4.17 Process Pools
- 10.4.18 Implementing MapReduce
11 NETWORKING

11.1 socket—Network Communication
   11.1.1 Addressing, Protocol Families, and Socket Types 562
   11.1.2 TCP/IP Client and Server 572
   11.1.3 User Datagram Client and Server 580
   11.1.4 UNIX Domain Sockets 583
   11.1.5 Multicast 587
   11.1.6 Sending Binary Data 591
   11.1.7 Nonblocking Communication and Timeouts 593

11.2 select—Wait for I/O Efficiently
   11.2.1 Using select() 595
   11.2.2 Nonblocking I/O with Timeouts 601
   11.2.3 Using poll() 603
   11.2.4 Platform-Specific Options 608

11.3 SocketServer—Creating Network Servers
   11.3.1 Server Types 609
   11.3.2 Server Objects 609
   11.3.3 Implementing a Server 610
   11.3.4 Request Handlers 610
   11.3.5 Echo Example 610
   11.3.6 Threading and Forking 616

11.4 asyncore—Asynchronous I/O
   11.4.1 Servers 619
   11.4.2 Clients 621
   11.4.3 The Event Loop 623
   11.4.4 Working with Other Event Loops 625
   11.4.5 Working with Files 628

11.5 asynchat—Asynchronous Protocol Handler
   11.5.1 Message Terminators 629
   11.5.2 Server and Handler 630
   11.5.3 Client 632
   11.5.4 Putting It All Together 634

12 THE INTERNET

12.1 urlparse—Split URLs into Components
   12.1.1 Parsing 638
   12.1.2 Unparsing 641
   12.1.3 Joining 642
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.9</td>
<td>json—JavaScript Object Notation</td>
<td>690</td>
</tr>
<tr>
<td>12.9.1</td>
<td>Encoding and Decoding Simple Data Types</td>
<td>690</td>
</tr>
<tr>
<td>12.9.2</td>
<td>Human-Consumable vs. Compact Output</td>
<td>692</td>
</tr>
<tr>
<td>12.9.3</td>
<td>Encoding Dictionaries</td>
<td>694</td>
</tr>
<tr>
<td>12.9.4</td>
<td>Working with Custom Types</td>
<td>695</td>
</tr>
<tr>
<td>12.9.5</td>
<td>Encoder and Decoder Classes</td>
<td>697</td>
</tr>
<tr>
<td>12.9.6</td>
<td>Working with Streams and Files</td>
<td>700</td>
</tr>
<tr>
<td>12.9.7</td>
<td>Mixed Data Streams</td>
<td>701</td>
</tr>
<tr>
<td>12.10</td>
<td>xmlrpclib—Client Library for XML-RPC</td>
<td>702</td>
</tr>
<tr>
<td>12.10.1</td>
<td>Connecting to a Server</td>
<td>704</td>
</tr>
<tr>
<td>12.10.2</td>
<td>Data Types</td>
<td>706</td>
</tr>
<tr>
<td>12.10.3</td>
<td>Passing Objects</td>
<td>709</td>
</tr>
<tr>
<td>12.10.4</td>
<td>Binary Data</td>
<td>710</td>
</tr>
<tr>
<td>12.10.5</td>
<td>Exception Handling</td>
<td>712</td>
</tr>
<tr>
<td>12.10.6</td>
<td>Combining Calls into One Message</td>
<td>712</td>
</tr>
<tr>
<td>12.11</td>
<td>SimpleXMLRPCServer—An XML-RPC Server</td>
<td>714</td>
</tr>
<tr>
<td>12.11.1</td>
<td>A Simple Server</td>
<td>714</td>
</tr>
<tr>
<td>12.11.2</td>
<td>Alternate API Names</td>
<td>716</td>
</tr>
<tr>
<td>12.11.3</td>
<td>Dotted API Names</td>
<td>718</td>
</tr>
<tr>
<td>12.11.4</td>
<td>Arbitrary API Names</td>
<td>719</td>
</tr>
<tr>
<td>12.11.5</td>
<td>Exposing Methods of Objects</td>
<td>720</td>
</tr>
<tr>
<td>12.11.6</td>
<td>Dispatching Calls</td>
<td>722</td>
</tr>
<tr>
<td>12.11.7</td>
<td>Introspection API</td>
<td>724</td>
</tr>
</tbody>
</table>

### EMAIL

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>13.1</td>
<td>smtplib—Simple Mail Transfer Protocol Client</td>
<td>727</td>
</tr>
<tr>
<td>13.1.1</td>
<td>Sending an Email Message</td>
<td>728</td>
</tr>
<tr>
<td>13.1.2</td>
<td>Authentication and Encryption</td>
<td>730</td>
</tr>
<tr>
<td>13.1.3</td>
<td>Verifying an Email Address</td>
<td>732</td>
</tr>
<tr>
<td>13.2</td>
<td>smtpd—Sample Mail Servers</td>
<td>734</td>
</tr>
<tr>
<td>13.2.1</td>
<td>Mail Server Base Class</td>
<td>734</td>
</tr>
<tr>
<td>13.2.2</td>
<td>Debugging Server</td>
<td>737</td>
</tr>
<tr>
<td>13.2.3</td>
<td>Proxy Server</td>
<td>737</td>
</tr>
<tr>
<td>13.3</td>
<td>imaplib—IMAP4 Client Library</td>
<td>738</td>
</tr>
<tr>
<td>13.3.1</td>
<td>Variations</td>
<td>739</td>
</tr>
<tr>
<td>13.3.2</td>
<td>Connecting to a Server</td>
<td>739</td>
</tr>
<tr>
<td>13.3.3</td>
<td>Example Configuration</td>
<td>741</td>
</tr>
<tr>
<td>13.3.4</td>
<td>Listing Mailboxes</td>
<td>741</td>
</tr>
<tr>
<td>13.3.5</td>
<td>Mailbox Status</td>
<td>744</td>
</tr>
</tbody>
</table>
Contents

13.3.6 Selecting a Mailbox 745
13.3.7 Searching for Messages 746
13.3.8 Search Criteria 747
13.3.9 Fetching Messages 749
13.3.10 Whole Messages 752
13.3.11 Uploading Messages 753
13.3.12 Moving and Copying Messages 755
13.3.13 Deleting Messages 756

13.4 mailbox—Manipulate Email Archives 758
13.4.1 mbox 759
13.4.2 Maildir 762
13.4.3 Other Formats 768

14 APPLICATION BUILDING BLOCKS 769
14.1 getopt—Command-Line Option Parsing 770
14.1.1 Function Arguments 771
14.1.2 Short-Form Options 771
14.1.3 Long-Form Options 772
14.1.4 A Complete Example 772
14.1.5 Abbreviating Long-Form Options 775
14.1.6 GNU-Style Option Parsing 775
14.1.7 Ending Argument Processing 777

14.2 optparse—Command-Line Option Parser 777
14.2.1 Creating an OptionParser 777
14.2.2 Short- and Long-Form Options 778
14.2.3 Comparing with getopt 779
14.2.4 Option Values 781
14.2.5 Option Actions 784
14.2.6 Help Messages 790

14.3 argparse—Command-Line Option and Argument Parsing 795
14.3.1 Comparing with optparse 796
14.3.2 Setting Up a Parser 796
14.3.3 Defining Arguments 796
14.3.4 Parsing a Command Line 796
14.3.5 Simple Examples 797
14.3.6 Automatically Generated Options 805
14.3.7 Parser Organization 807
14.3.8 Advanced Argument Processing 815
14.4 readline—The GNU Readline Library  
14.4.1 Configuring  
14.4.2 Completing Text  
14.4.3 Accessing the Completion Buffer  
14.4.4 Input History  
14.4.5 Hooks  
14.5 getpass—Secure Password Prompt  
14.5.1 Example  
14.5.2 Using getpass without a Terminal  
14.6 cmd—Line-Oriented Command Processors  
14.6.1 Processing Commands  
14.6.2 Command Arguments  
14.6.3 Live Help  
14.6.4 Auto-Completion  
14.6.5 Overriding Base Class Methods  
14.6.6 Configuring Cmd through Attributes  
14.6.7 Running Shell Commands  
14.6.8 Alternative Inputs  
14.6.9 Commands from sys.argv  
14.7 shlex—Parse Shell-Style Syntaxes  
14.7.1 Quoted Strings  
14.7.2 Embedded Comments  
14.7.3 Split  
14.7.4 Including Other Sources of Tokens  
14.7.5 Controlling the Parser  
14.7.6 Error Handling  
14.7.7 POSIX vs. Non-POSIX Parsing  
14.8 ConfigParser—Work with Configuration Files  
14.8.1 Configuration File Format  
14.8.2 Reading Configuration Files  
14.8.3 Accessing Configuration Settings  
14.8.4 Modifying Settings  
14.8.5 Saving Configuration Files  
14.8.6 Option Search Path  
14.8.7 Combining Values with Interpolation  
14.9 logging—Report Status, Error, and Informational Messages  
14.9.1 Logging in Applications vs. Libraries  
14.9.2 Logging to a File  
14.9.3 Rotating Log Files
14.9.4 Verbosity Levels 880
14.9.5 Naming Logger Instances 882

14.10 fileinput—Command-Line Filter Framework 883
14.10.1 Converting M3U Files to RSS 883
14.10.2 Progress Metadata 886
14.10.3 In-Place Filtering 887

14.11 atexit—Program Shutdown Callbacks 890
14.11.1 Examples 890
14.11.2 When Are atexit Functions Not Called? 891
14.11.3 Handling Exceptions 893

14.12 sched—Timed Event Scheduler 894
14.12.1 Running Events with a Delay 895
14.12.2 Overlapping Events 896
14.12.3 Event Priorities 897
14.12.4 Canceling Events 897

15 INTERNATIONALIZATION AND LOCALIZATION 899
15.1 gettext—Message Catalogs 899
15.1.1 Translation Workflow Overview 900
15.1.2 Creating Message Catalogs from Source Code 900
15.1.3 Finding Message Catalogs at Runtime 903
15.1.4 Plural Values 905
15.1.5 Application vs. Module Localization 907
15.1.6 Switching Translations 908

15.2 locale—Cultural Localization API 909
15.2.1 Probing the Current Locale 909
15.2.2 Currency 915
15.2.3 Formatting Numbers 916
15.2.4 Parsing Numbers 917
15.2.5 Dates and Times 917

16 DEVELOPER TOOLS 919
16.1 pydoc—Online Help for Modules 920
16.1.1 Plain-Text Help 920
16.1.2 HTML Help 920
16.1.3 Interactive Help 921

16.2 doctest—Testing through Documentation 921
16.2.1 Getting Started 922
16.2.2 Handling Unpredictable Output 924
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>16.2.3</td>
<td>Tracebacks</td>
<td>928</td>
</tr>
<tr>
<td>16.2.4</td>
<td>Working around Whitespace</td>
<td>930</td>
</tr>
<tr>
<td>16.2.5</td>
<td>Test Locations</td>
<td>936</td>
</tr>
<tr>
<td>16.2.6</td>
<td>External Documentation</td>
<td>939</td>
</tr>
<tr>
<td>16.2.7</td>
<td>Running Tests</td>
<td>942</td>
</tr>
<tr>
<td>16.2.8</td>
<td>Test Context</td>
<td>945</td>
</tr>
<tr>
<td>16.3</td>
<td>unittest—Automated Testing Framework</td>
<td>949</td>
</tr>
<tr>
<td>16.3.1</td>
<td>Basic Test Structure</td>
<td>949</td>
</tr>
<tr>
<td>16.3.2</td>
<td>Running Tests</td>
<td>949</td>
</tr>
<tr>
<td>16.3.3</td>
<td>Test Outcomes</td>
<td>950</td>
</tr>
<tr>
<td>16.3.4</td>
<td>Asserting Truth</td>
<td>952</td>
</tr>
<tr>
<td>16.3.5</td>
<td>Testing Equality</td>
<td>953</td>
</tr>
<tr>
<td>16.3.6</td>
<td>Almost Equal?</td>
<td>954</td>
</tr>
<tr>
<td>16.3.7</td>
<td>Testing for Exceptions</td>
<td>955</td>
</tr>
<tr>
<td>16.3.8</td>
<td>Test Fixtures</td>
<td>956</td>
</tr>
<tr>
<td>16.3.9</td>
<td>Test Suites</td>
<td>957</td>
</tr>
<tr>
<td>16.4</td>
<td>traceback—Exceptions and Stack Traces</td>
<td>958</td>
</tr>
<tr>
<td>16.4.1</td>
<td>Supporting Functions</td>
<td>958</td>
</tr>
<tr>
<td>16.4.2</td>
<td>Working with Exceptions</td>
<td>959</td>
</tr>
<tr>
<td>16.4.3</td>
<td>Working with the Stack</td>
<td>963</td>
</tr>
<tr>
<td>16.5</td>
<td>cgitb—Detailed Traceback Reports</td>
<td>965</td>
</tr>
<tr>
<td>16.5.1</td>
<td>Standard Traceback Dumps</td>
<td>966</td>
</tr>
<tr>
<td>16.5.2</td>
<td>Enabling Detailed Tracebacks</td>
<td>966</td>
</tr>
<tr>
<td>16.5.3</td>
<td>Local Variables in Tracebacks</td>
<td>968</td>
</tr>
<tr>
<td>16.5.4</td>
<td>Exception Properties</td>
<td>971</td>
</tr>
<tr>
<td>16.5.5</td>
<td>HTML Output</td>
<td>972</td>
</tr>
<tr>
<td>16.5.6</td>
<td>Logging Tracebacks</td>
<td>972</td>
</tr>
<tr>
<td>16.6</td>
<td>pdb—Interactive Debugger</td>
<td>975</td>
</tr>
<tr>
<td>16.6.1</td>
<td>Starting the Debugger</td>
<td>976</td>
</tr>
<tr>
<td>16.6.2</td>
<td>Controlling the Debugger</td>
<td>979</td>
</tr>
<tr>
<td>16.6.3</td>
<td>Breakpoints</td>
<td>990</td>
</tr>
<tr>
<td>16.6.4</td>
<td>Changing Execution Flow</td>
<td>1002</td>
</tr>
<tr>
<td>16.6.5</td>
<td>Customizing the Debugger with Aliases</td>
<td>1009</td>
</tr>
<tr>
<td>16.6.6</td>
<td>Saving Configuration Settings</td>
<td>1011</td>
</tr>
<tr>
<td>16.7</td>
<td>trace—Follow Program Flow</td>
<td>1012</td>
</tr>
<tr>
<td>16.7.1</td>
<td>Example Program</td>
<td>1013</td>
</tr>
<tr>
<td>16.7.2</td>
<td>Tracing Execution</td>
<td>1013</td>
</tr>
<tr>
<td>16.7.3</td>
<td>Code Coverage</td>
<td>1014</td>
</tr>
<tr>
<td>16.7.4</td>
<td>Calling Relationships</td>
<td>1017</td>
</tr>
</tbody>
</table>
### 16.7.5 Programming Interface
- Page 1018

### 16.7.6 Saving Result Data
- Page 1020

### 16.7.7 Options
- Page 1022

#### 16.8 profile and pstats—Performance Analysis
- Page 1022
  - 16.8.1 Running the Profiler
    - Page 1023
  - 16.8.2 Running in a Context
    - Page 1026
  - 16.8.3 pstats: Saving and Working with Statistics
    - Page 1027
  - 16.8.4 Limiting Report Contents
    - Page 1028
  - 16.8.5 Caller / Callee Graphs
    - Page 1029

#### 16.9 timeit—Time the Execution of Small Bits of Python Code
- Page 1031
  - 16.9.1 Module Contents
    - Page 1031
  - 16.9.2 Basic Example
    - Page 1032
  - 16.9.3 Storing Values in a Dictionary
    - Page 1033
  - 16.9.4 From the Command Line
    - Page 1035

#### 16.10 compileall—Byte-Compile Source Files
- Page 1037
  - 16.10.1 Compiling One Directory
    - Page 1037
  - 16.10.2 Compiling sys.path
    - Page 1038
  - 16.10.3 From the Command Line
    - Page 1039

#### 16.11 pyclbr—Class Browser
- Page 1039
  - 16.11.1 Scanning for Classes
    - Page 1041
  - 16.11.2 Scanning for Functions
    - Page 1042

### 17 RUNTIME FEATURES
- Page 1045

#### 17.1 site—Site-Wide Configuration
- Page 1046
  - 17.1.1 Import Path
    - Page 1046
  - 17.1.2 User Directories
    - Page 1047
  - 17.1.3 Path Configuration Files
    - Page 1049
  - 17.1.4 Customizing Site Configuration
    - Page 1051
  - 17.1.5 Customizing User Configuration
    - Page 1053
  - 17.1.6 Disabling the site Module
    - Page 1054

#### 17.2 sys—System-Specific Configuration
- Page 1055
  - 17.2.1 Interpreter Settings
    - Page 1055
  - 17.2.2 Runtime Environment
    - Page 1062
  - 17.2.3 Memory Management and Limits
    - Page 1065
  - 17.2.4 Exception Handling
    - Page 1071
  - 17.2.5 Low-Level Thread Support
    - Page 1074
  - 17.2.6 Modules and Imports
    - Page 1080
  - 17.2.7 Tracing a Program as It Runs
    - Page 1101
17.3 os—Portable Access to Operating System Specific Features
17.3.1 Process Owner 1108
17.3.2 Process Environment 1111
17.3.3 Process Working Directory 1112
17.3.4 Pipes 1112
17.3.5 File Descriptors 1116
17.3.6 File System Permissions 1116
17.3.7 Directories 1118
17.3.8 Symbolic Links 1119
17.3.9 Walking a Directory Tree 1120
17.3.10 Running External Commands 1121
17.3.11 Creating Processes with os.fork() 1122
17.3.12 Waiting for a Child 1125
17.3.13 Spawn 1127
17.3.14 File System Permissions 1127

17.4 platform—System Version Information 1129
17.4.1 Interpreter 1129
17.4.2 Platform 1130
17.4.3 Operating System and Hardware Info 1131
17.4.4 Executable Architecture 1133

17.5 resource—System Resource Management 1134
17.5.1 Current Usage 1134
17.5.2 Resource Limits 1135

17.6 gc—Garbage Collector 1138
17.6.1 Tracing References 1138
17.6.2 Forcing Garbage Collection 1141
17.6.3 Finding References to Objects that Cannot Be Collected 1146
17.6.4 Collection Thresholds and Generations 1148
17.6.5 Debugging 1151

17.7 sysconfig—Interpreter Compile-Time Configuration 1160
17.7.1 Configuration Variables 1160
17.7.2 Installation Paths 1163
17.7.3 Python Version and Platform 1167

18 LANGUAGE TOOLS 1169
18.1 warnings—Nonfatal Alerts 1170
18.1.1 Categories and Filtering 1170
18.1.2 Generating Warnings 1171
18.1.3 Filtering with Patterns 1172
18.1.4 Repeated Warnings 1174
18.1.5 Alternate Message Delivery Functions 1175
18.1.6 Formatting 1176
18.1.7 Stack Level in Warnings 1177
18.2 abc—Abstract Base Classes 1178
18.2.1 Why Use Abstract Base Classes? 1178
18.2.2 How Abstract Base Classes Work 1178
18.2.3 Registering a Concrete Class 1179
18.2.4 Implementation through Subclassing 1179
18.2.5 Concrete Methods in ABCs 1181
18.2.6 Abstract Properties 1182
18.3 dis—Python Bytecode Disassembler 1186
18.3.1 Basic Disassembly 1187
18.3.2 Disassembling Functions 1187
18.3.3 Classes 1189
18.3.4 Using Disassembly to Debug 1190
18.3.5 Performance Analysis of Loops 1192
18.3.6 Compiler Optimizations 1198
18.4 inspect—Inspect Live Objects 1200
18.4.1 Example Module 1200
18.4.2 Module Information 1201
18.4.3 Inspecting Modules 1203
18.4.4 Inspecting Classes 1204
18.4.5 Documentation Strings 1206
18.4.6 Retrieving Source 1207
18.4.7 Method and Function Arguments 1209
18.4.8 Class Hierarchies 1210
18.4.9 Method Resolution Order 1212
18.4.10 The Stack and Frames 1213
18.5 exceptions—Built-in Exception Classes 1216
18.5.1 Base Classes 1216
18.5.2 Raised Exceptions 1217
18.5.3 Warning Categories 1233

19 MODULES AND PACKAGES 1235
19.1 imp—Python’s Import Mechanism 1235
19.1.1 Example Package 1236
19.1.2 Module Types 1236
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>19.1.3</td>
<td>Finding Modules</td>
<td>1237</td>
</tr>
<tr>
<td>19.1.4</td>
<td>Loading Modules</td>
<td>1238</td>
</tr>
<tr>
<td>19.2</td>
<td>zipimport—Load Python Code from ZIP Archives</td>
<td>1240</td>
</tr>
<tr>
<td>19.2.1</td>
<td>Example</td>
<td>1240</td>
</tr>
<tr>
<td>19.2.2</td>
<td>Finding a Module</td>
<td>1241</td>
</tr>
<tr>
<td>19.2.3</td>
<td>Accessing Code</td>
<td>1242</td>
</tr>
<tr>
<td>19.2.4</td>
<td>Source</td>
<td>1243</td>
</tr>
<tr>
<td>19.2.5</td>
<td>Packages</td>
<td>1244</td>
</tr>
<tr>
<td>19.2.6</td>
<td>Data</td>
<td>1244</td>
</tr>
<tr>
<td>19.3</td>
<td>pkgutil—Package Utilities</td>
<td>1247</td>
</tr>
<tr>
<td>19.3.1</td>
<td>Package Import Paths</td>
<td>1247</td>
</tr>
<tr>
<td>19.3.2</td>
<td>Development Versions of Packages</td>
<td>1249</td>
</tr>
<tr>
<td>19.3.3</td>
<td>Managing Paths with PKG Files</td>
<td>1251</td>
</tr>
<tr>
<td>19.3.4</td>
<td>Nested Packages</td>
<td>1253</td>
</tr>
<tr>
<td>19.3.5</td>
<td>Package Data</td>
<td>1255</td>
</tr>
</tbody>
</table>

*Index of Python Modules* 1259

*Index* 1261
This page intentionally left blank
TABLES

1.1 Regular Expression Escape Codes 24
1.2 Regular Expression Anchoring Codes 27
1.3 Regular Expression Flag Abbreviations 45

2.1 Byte Order Specifiers for struct 104

6.1 Codec Error Handling Modes 292

7.1 The “project” Table 353
7.2 The “task” Table 353
7.3 CSV Dialect Parameters 415

10.1 Multiprocessing Exit Codes 537

11.1 Event Flags for poll() 604

13.1 IMAP 4 Mailbox Status Conditions 744

14.1 Flags for Variable Argument Definitions in argparse 815
14.2 Logging Levels 881

16.1 Test Case Outcomes 950

17.1 CPython Command-Line Option Flags 1057
17.2 Event Hooks for settrace() 1101
17.3 Platform Information Functions 1132
17.4 Path Names Used in sysconfig 1164

18.1 Warning Filter Actions 1171

xxx
This page intentionally left blank
It’s Thanksgiving Day, 2010. For those outside of the United States, and for many of those within it, it might just seem like a holiday where people eat a ton of food, watch some football, and otherwise hang out.

For me, and many others, it’s a time to take a look back and think about the things that have enriched our lives and give thanks for them. Sure, we should be doing that every day, but having a single day that’s focused on just saying thanks sometimes makes us think a bit more broadly and a bit more deeply.

I’m sitting here writing the foreward to this book, something I’m very thankful for having the opportunity to do—but I’m not just thinking about the content of the book, or the author, who is a fantastic community member. I’m thinking about the subject matter itself—Python—and specifically, its standard library.

Every version of Python shipped today contains hundreds of modules spanning many years, many developers, many subjects, and many tasks. It contains modules for everything from sending and receiving email, to GUI development, to a built-in HTTP server. By itself, the standard library is a massive work. Without the people who have maintained it throughout the years, and the hundreds of people who have submitted patches, documentation, and feedback, it would not be what it is today.

It’s an astounding accomplishment, and something that has been the critical component in the rise of Python’s popularity as a language and ecosystem. Without the standard library, without the “batteries included” motto of the core team and others, Python would never have come as far. It has been downloaded by hundreds of thousands of people and companies, and has been installed on millions of servers, desktops, and other devices.

Without the standard library, Python would still be a fantastic language, built on solid concepts of teaching, learning, and readability. It might have gotten far enough
on its own, based on those merits. But the standard library turns it from an interesting experiment into a powerful and effective tool.

Every day, developers across the world build tools and entire applications based on nothing but the core language and the standard library. You not only get the ability to conceptualize what a car is (the language), but you also get enough parts and tools to put together a basic car yourself. It might not be the perfect car, but it gets you from A to B, and that’s incredibly empowering and rewarding. Time and time again, I speak to people who look at me proudly and say, “Look what I built with nothing except what came with Python!”

It is not, however, a fait accompli. The standard library has its warts. Given its size and breadth, and its age, it’s no real surprise that some of the modules have varying levels of quality, API clarity, and coverage. Some of the modules have suffered “feature creep,” or have failed to keep up with modern advances in the areas they cover. Python continues to evolve, grow, and improve over time through the help and hard work of many, many unpaid volunteers.

Some argue, though, that due to the shortcomings and because the standard library doesn’t necessarily comprise the “best of breed” solutions for the areas its modules cover (“best of” is a continually moving and adapting target, after all), that it should be killed or sent out to pasture, despite continual improvement. These people miss the fact that not only is the standard library a critical piece of what makes Python continually successful, but also, despite its warts, it is still an excellent resource.

But I’ve intentionally ignored one giant area: documentation. The standard library’s documentation is good and is constantly improving and evolving. Given the size and breadth of the standard library, the documentation is amazing for what it is. It’s awesome that we have hundreds of pages of documentation contributed by hundreds of developers and users. The documentation is used every single day by hundreds of thousands of people to create things—things as simple as one-off scripts and as complex as the software that controls giant robotic arms.

The documentation is why we are here, though. All good documentation and code starts with an idea—a kernel of a concept about what something is, or will be. Outward from that kernel come the characters (the APIs) and the storyline (the modules). In the case of code, sometimes it starts with a simple idea: “I want to parse a string and look for a date.” But when you reach the end—when you’re looking at the few hundred unit tests, functions, and other bits you’ve made—you sit back and realize you’ve built something much, much more vast than originally intended. The same goes for documentation, especially the documentation of code.

The examples are the most critical component in the documentation of code, in my estimation. You can write a narrative about a piece of an API until it spans entire books, and you can describe the loosely coupled interface with pretty words and thoughtful use
cases. But it all falls flat if a user approaching it for the first time can’t glue those pretty
words, thoughtful use cases, and API signatures together into something that makes
sense and solves their problems.

Examples are the gateway by which people make the critical connections—those
logical jumps from an abstract concept into something concrete. It’s one thing to
“know” the ideas and API; it’s another to see it used. It helps jump the void when
you’re not only trying to learn something, but also trying to improve existing things.

Which brings us back to Python. Doug Hellmann, the author of this book, started
a blog in 2007 called the Python Module of the Week. In the blog, he walked through
various modules of the standard library, taking an example-first approach to showing
how each one worked and why. From the first day I read it, it had a place right next to
the core Python documentation. His writing has become an indispensable resource for
me and many other people in the Python community.

Doug’s writings fill a critical gap in the Python documentation I see today: the
need for examples. Showing how and why something works in a functional, simple
manner is no easy task. And, as we’ve seen, it’s a critical and valuable body of work
that helps people every single day. People send me emails with alarming regularity
saying things like, “Did you see this post by Doug? This is awesome!” or “Why isn’t
this in the core documentation? It helped me understand how things really work!”

When I heard Doug was going to take the time to further flesh out his existing
work, to turn it into a book I could keep on my desk to dog-ear and wear out from near
constant use, I was more than a little excited. Doug is a fantastic technical writer with
a great eye for detail. Having an entire book dedicated to real examples of how over a
hundred modules in the standard library work, written by him, blows my mind.

You see, I’m thankful for Python. I’m thankful for the standard library—warts and
all. I’m thankful for the massive, vibrant, yet sometimes dysfunctional community we
have. I’m thankful for the tireless work of the core development team, past, present
and future. I’m thankful for the resources, the time, and the effort so many community
members—of which Doug Hellmann is an exemplary example—have put into making
this community and ecosystem such an amazing place.

Lastly, I’m thankful for this book. Its author will continue to be well respected and
the book well used in the years to come.

— Jesse Noller
Python Core Developer
PSF Board Member
Principal Engineer, Nasuni Corporation
ACKNOWLEDGMENTS

This book would not have come into being without the contributions and support of many people.

I was first introduced to Python around 1997 by Dick Wall, while we were working together on GIS software at ERDAS. I remember being simultaneously happy that I had found a new tool language that was so easy to use, and sad that the company did not let us use it for “real work.” I have used Python extensively at all of my subsequent jobs, and I have Dick to thank for the many happy hours I have spent working on software since then.

The Python core development team has created a robust ecosystem of language, tools, and libraries that continue to grow in popularity and find new application areas. Without the amazing investment in time and resources they have given us, we would all still be spending our time reinventing wheel after wheel.

As described in the Introduction, the material in this book started out as a series of blog posts. Each of those posts has been reviewed and commented on by members of the Python community, with corrections, suggestions, and questions that led to changes in the version you find here. Thank you all for reading along week after week, and contributing your time and attention.

The technical reviewers for the book—Matt Culbreth, Katie Cunningham, Jeff McNeil, and Keyton Weissinger—spent many hours looking for issues with the example code and accompanying explanations. The result is stronger than I could have produced on my own. I also received advice from Jesse Noller on the multiprocessing module and Brett Cannon on creating custom importers.

A special thanks goes to the editors and production staff at Pearson for all their hard work and assistance in helping me realize my vision for this book.
Finally, I want to thank my wife, Theresa Flynn, who has always given me excellent writing advice and was a constant source of encouragement throughout the entire process of creating this book. I doubt she knew what she was getting herself into when she told me, “You know, at some point, you have to sit down and start writing it.” It’s your turn.
Doug Hellmann is currently a senior developer with Racemi, Inc., and communications director of the Python Software Foundation. He has been programming in Python since version 1.4 and has worked on a variety of UNIX and non-UNIX platforms for projects in fields such as mapping, medical news publishing, banking, and data center automation. After a year as a regular columnist for Python Magazine, he served as editor-in-chief from 2008–2009. Since 2007, Doug has published the popular Python Module of the Week series on his blog. He lives in Athens, Georgia.
This page intentionally left blank
Distributed with every copy of Python, the standard library contains hundreds of
modules that provide tools for interacting with the operating system, interpreter, and
Internet. All of them are tested and ready to be used to jump start the development of
your applications. This book presents selected examples demonstrating how to use the
most commonly used features of the modules that give Python its “batteries included”
slogan, taken from the popular Python Module of the Week (PyMOTW) blog series.

This Book’s Target Audience

The audience for this book is an intermediate Python programmer, so although all the
source code is presented with discussion, only a few cases include line-by-line expla-
nations. Every section focuses on the features of the modules, illustrated by the source
code and output from fully independent example programs. Each feature is presented as
concisely as possible, so the reader can focus on the module or function being demon-
strated without being distracted by the supporting code.

An experienced programmer familiar with other languages may be able to learn
Python from this book, but it is not intended to be an introduction to the language. Some
prior experience writing Python programs will be useful when studying the examples.

Several sections, such as the description of network programming with sockets or
hmac encryption, require domain-specific knowledge. The basic information needed to
explain the examples is included here, but the range of topics covered by the modules
in the standard library makes it impossible to cover every topic comprehensively in
a single volume. The discussion of each module is followed by a list of suggested
sources for more information and further reading. These include online resources, RFC
standards documents, and related books.

Although the current transition to Python 3 is well underway, Python 2 is still
likely to be the primary version of Python used in production environments for years
to come because of the large amount of legacy Python 2 source code available and
the slow transition rate to Python 3. All the source code for the examples has been
updated from the original online versions and tested with Python 2.7, the final release
of the 2.x series. Many of the example programs can be readily adapted to work with
Python 3, but others cover modules that have been renamed or deprecated.

**How This Book Is Organized**

The modules are grouped into chapters to make it easy to find an individual module for
reference and browse by subject for more leisurely exploration. The book supplements
the comprehensive reference guide available on http://docs.python.org, providing fully
functional example programs to demonstrate the features described there.

**Downloading the Example Code**

The original versions of the articles, errata for the book, and the sample code are avail-
able on the author’s web site (http://www.doughellmann.com/books/byexample).
Chapter 2

DATA STRUCTURES

Python includes several standard programming data structures, such as `list`, `tuple`, `dict`, and `set`, as part of its built-in types. Many applications do not require other structures, but when they do, the standard library provides powerful and well-tested versions that are ready to use.

The `collections` module includes implementations of several data structures that extend those found in other modules. For example, `Deque` is a double-ended queue that allows the addition or removal of items from either end. The `defaultdict` is a dictionary that responds with a default value if a key is missing, while `OrderedDict` remembers the sequence in which items are added to it. And `namedtuple` extends the normal `tuple` to give each member item an attribute name in addition to a numeric index.

For large amounts of data, an `array` may make more efficient use of memory than a `list`. Since the `array` is limited to a single data type, it can use a more compact memory representation than a general purpose `list`. At the same time, `arrays` can be manipulated using many of the same methods as a `list`, so it may be possible to replace `lists` with `arrays` in an application without a lot of other changes.

Sorting items in a sequence is a fundamental aspect of data manipulation. Python’s `list` includes a `sort()` method, but sometimes it is more efficient to maintain a list in sorted order without resorting it each time its contents are changed. The functions in `heapq` modify the contents of a list while preserving the sort order of the list with low overhead.

Another option for building sorted lists or arrays is `bisect`. It uses a binary search to find the insertion point for new items and is an alternative to repeatedly sorting a list that changes frequently.
Although the built-in list can simulate a queue using the insert() and pop() methods, it is not thread-safe. For true ordered communication between threads, use the Queue module. multiprocessing includes a version of a Queue that works between processes, making it easier to convert a multithreaded program to use processes instead.

struct is useful for decoding data from another application, perhaps coming from a binary file or stream of data, into Python’s native types for easier manipulation.

This chapter covers two modules related to memory management. For highly interconnected data structures, such as graphs and trees, use weakref to maintain references while still allowing the garbage collector to clean up objects after they are no longer needed. The functions in copy are used for duplicating data structures and their contents, including recursive copies with deepcopy().

Debugging data structures can be time consuming, especially when wading through printed output of large sequences or dictionaries. Use pprint to create easy-to-read representations that can be printed to the console or written to a log file for easier debugging.

And, finally, if the available types do not meet the requirements, subclass one of the native types and customize it, or build a new container type using one of the abstract base classes defined in collections as a starting point.

2.1 collections—Container Data Types

Purpose Container data types.

Python Version 2.4 and later

The collections module includes container data types beyond the built-in types list, dict, and tuple.

2.1.1 Counter

A Counter is a container that tracks how many times equivalent values are added. It can be used to implement the same algorithms for which other languages commonly use bag or multiset data structures.

Initializing

Counter supports three forms of initialization. Its constructor can be called with a sequence of items, a dictionary containing keys and counts, or using keyword arguments mapping string names to counts.
import collections

print collections.Counter(['a', 'b', 'c', 'a', 'b', 'b'])
print collections.Counter({'a':2, 'b':3, 'c':1})
print collections.Counter(a=2, b=3, c=1)

The results of all three forms of initialization are the same.

$ python collections_counter_init.py
Counter({'b': 3, 'a': 2, 'c': 1})
Counter({'b': 3, 'a': 2, 'c': 1})
Counter({'b': 3, 'a': 2, 'c': 1})

An empty Counter can be constructed with no arguments and populated via the update() method.

import collections
c = collections.Counter()
print 'Initial :', c

c.update('abcdaab')
print 'Sequence:', c

c.update({'a':1, 'd':5})
print 'Dict :', c

The count values are increased based on the new data, rather than replaced. In this example, the count for a goes from 3 to 4.

$ python collections_counter_update.py
Initial : Counter()
Sequence: Counter({'a': 3, 'b': 2, 'c': 1, 'd': 1})
Dict : Counter({'d': 6, 'a': 4, 'b': 2, 'c': 1})

Accessing Counts

Once a Counter is populated, its values can be retrieved using the dictionary API.
import collections

c = collections.Counter('abcdab')

for letter in 'abcde':
    print '%s : %d' % (letter, c[letter])

Counter does not raise KeyError for unknown items. If a value has not been seen in the input (as with e in this example), its count is 0.

$ python collections_counter_get_values.py

a : 3
b : 2
c : 1
d : 1
e : 0

The elements() method returns an iterator that produces all items known to the Counter.

import collections

c = collections.Counter('extremely')
c['z'] = 0
print c
print list(c.elements())

The order of elements is not guaranteed, and items with counts less than or equal to zero are not included.

$ python collections_counter_elements.py

Counter({'e': 3, 'm': 1, 'l': 1, 'r': 1, 't': 1, 'y': 1, 'x': 1, 'z': 0})
['e', 'e', 'e', 'm', 'l', 'r', 't', 'y', 'x']

Use most_common() to produce a sequence of the n most frequently encountered input values and their respective counts.
import collections

c = collections.Counter()
with open('/usr/share/dict/words', 'rt') as f:
    for line in f:
        c.update(line.rstrip().lower())

print 'Most common:'
for letter, count in c.most_common(3):
    print '%s: %d' % (letter, count)

This example counts the letters appearing in all words in the system dictionary to produce a frequency distribution, and then prints the three most common letters. Leaving out the argument to most_common() produces a list of all the items, in order of frequency.

$ python collections_counter_most_common.py

Most common:
e: 234803
i: 200613
a: 198938

Arithmetic

Counter instances support arithmetic and set operations for aggregating results.

import collections

c1 = collections.Counter(['a', 'b', 'c', 'a', 'b', 'b'])
c2 = collections.Counter('alphabet')

print 'C1:', c1
print 'C2:', c2

print 'Combined counts:'
print c1 + c2

print 'Subtraction:'
print c1 - c2
print '\nIntersection (taking positive minimums):'
print c1 & c2

print '\nUnion (taking maximums):'
print c1 | c2

Each time a new Counter is produced through an operation, any items with zero or negative counts are discarded. The count for a is the same in c1 and c2, so subtraction leaves it at zero.

$ python collections_counter_arithmetic.py

C1: Counter({'b': 3, 'a': 2, 'c': 1})
C2: Counter({'a': 2, 'b': 1, 'e': 1, 'h': 1, 'l': 1, 'p': 1, 't': 1})

Combined counts:
Counter({'a': 4, 'b': 4, 'c': 1, 'e': 1, 'h': 1, 'l': 1, 'p': 1, 't': 1})

Subtraction:
Counter({'b': 2, 'c': 1})

Intersection (taking positive minimums):
Counter({'a': 2, 'b': 1})

Union (taking maximums):
Counter({'b': 3, 'a': 2, 'c': 1, 'e': 1, 'h': 1, 'l': 1, 'p': 1, 't': 1})

2.1.2 defaultdict

The standard dictionary includes the method setdefault() for retrieving a value and establishing a default if the value does not exist. By contrast, defaultdict lets the caller specify the default up front when the container is initialized.

import collections
def default_factory():
    return 'default value'

d = collections.defaultdict(default_factory, foo='bar')
print 'd:', d
print 'foo =>', d['foo']
print 'bar =>', d['bar']

This method works well, as long as it is appropriate for all keys to have the same default. It can be especially useful if the default is a type used for aggregating or accumulating values, such as a list, set, or even int. The standard library documentation includes several examples of using defaultdict this way.

$ python collections_defaultdict.py

d: defaultdict(<function default_factory
    at 0x100d9ba28>, {'foo': 'bar'})
foo => bar
bar => default value

See Also:
defaultdict examples (http://docs.python.org/lib/defaultdict-examples.html)
Examples of using defaultdict from the standard library documentation.
Evolution of Default Dictionaries in Python
(http://jtauber.com/blog/2008/02/27/evolution_of_default_dictionaries_in_python/) Discussion from James Tauber of how defaultdict relates to other means of initializing dictionaries.

2.1.3 Deque

A double-ended queue, or deque, supports adding and removing elements from either end. The more commonly used structures, stacks, and queues are degenerate forms of deques where the inputs and outputs are restricted to a single end.

import collections

d = collections.deque('abcdefg')
print 'Deque:', d
print 'Length:', len(d)
print 'Left end:', d[0]
print 'Right end:', d[-1]

d.remove('c')
print 'remove(c):', d
Since deques are a type of sequence container, they support some of the same operations as `list`, such as examining the contents with `__getitem__()` , determining length, and removing elements from the middle by matching identity.

```
$ python collections_deque.py

Deque: deque([‘a’, ’b’, ’c’, ’d’, ’e’, ’f’, ’g’])
Length: 7
Left end: a
Right end: g
remove(c): deque([‘a’, ’b’, ’d’, ’e’, ’f’, ’g’])
```

### Populating

A deque can be populated from either end, termed “left” and “right” in the Python implementation.

```python
import collections

# Add to the right
d1 = collections.deque()
d1.extend(‘abcdefg’)  
print ‘extend :’, d1
print ‘append :’, d1

# Add to the left
d2 = collections.deque()
d2.extendleft(xrange(6))
print ‘extendleft:’, d2
print ‘appendleft:’, d2
```

The `extendleft()` function iterates over its input and performs the equivalent of an `appendleft()` for each item. The end result is that the deque contains the input sequence in reverse order.

```
$ python collections_deque_populating.py

```
2.1. collections—Container Data Types

extendleft: deque([5, 4, 3, 2, 1, 0])
appendleft: deque([6, 5, 4, 3, 2, 1, 0])

Consuming
Similarly, the elements of the deque can be consumed from both ends or either end, depending on the algorithm being applied.

```python
import collections

print 'From the right:'
d = collections.deque('abcdefg')
while True:
    try:
        print d.pop(),
    except IndexError:
        break

print

print '
From the left:'
d = collections.deque(xrange(6))
while True:
    try:
        print d.popleft(),
    except IndexError:
        break

print
```

Use `pop()` to remove an item from the right end of the deque and `popleft()` to take from the left end.

$ python collections_deque_consuming.py

From the right:
g f e d c b a

From the left:
0 1 2 3 4 5

Since deques are thread-safe, the contents can even be consumed from both ends at the same time from separate threads.
```python
import collections
import threading
import time

candle = collections.deque(xrange(5))

def burn(direction, nextSource):
    while True:
        try:
            next = nextSource()
        except IndexError:
            break
        else:
            print '%8s %s' % (direction, next)
            time.sleep(0.1)
        print '%8s done' % direction
    return

left = threading.Thread(target=burn, args=('Left', candle.popleft))
right = threading.Thread(target=burn, args=('Right', candle.pop))

left.start()
right.start()

left.join()
right.join()

The threads in this example alternate between each end, removing items until the
deqe is empty.

$ python collections_deque_both_ends.py

   Left: 0
   Right: 4
   Right: 3
     Left: 1
   Right: 2
    Left done
   Right done

Rotating

Another useful capability of the deque is to rotate it in either direction, to skip over
some items.
import collections

d = collections.deque(xrange(10))
print 'Normal :', d

d = collections.deque(xrange(10))
d.rotate(2)
print 'Right rotation:', d

d = collections.deque(xrange(10))
d.rotate(-2)
print 'Left rotation :', d

Rotating the deque to the right (using a positive rotation) takes items from the right end and moves them to the left end. Rotating to the left (with a negative value) takes items from the left end and moves them to the right end. It may help to visualize the items in the deque as being engraved along the edge of a dial.

$ python collections_deque_rotate.py

Normal : deque([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
Right rotation: deque([8, 9, 0, 1, 2, 3, 4, 5, 6, 7])
Left rotation : deque([2, 3, 4, 5, 6, 7, 8, 9, 0, 1])

See Also:
Deque (http://en.wikipedia.org/wiki/Deque) Wikipedia article that provides a dis-
cussion of the deque data structure.
Deque Recipes (http://docs.python.org/lib/deque-recipes.html) Examples of using
deques in algorithms from the standard library documentation.

2.1.4 namedtuple

The standard tuple uses numerical indexes to access its members.

bob = ('Bob', 30, 'male')
print 'Representation:', bob

jane = ('Jane', 29, 'female')
print '\nField by index:', jane[0]

print '\nFields by index:'
for p in [bob, jane]:
    print '$s is a $d year old $s' % p
This makes tuples convenient containers for simple uses.

```python
$ python collections_tuple.py

Representation: ('Bob', 30, 'male')

Field by index: Jane

Fields by index:
Bob is a 30 year old male
Jane is a 29 year old female
```

On the other hand, remembering which index should be used for each value can lead to errors, especially if the tuple has a lot of fields and is constructed far from where it is used. A namedtuple assigns names, as well as the numerical index, to each member.

**Defining**

namedtuple instances are just as memory efficient as regular tuples because they do not have per-instance dictionaries. Each kind of namedtuple is represented by its own class, created by using the namedtuple() factory function. The arguments are the name of the new class and a string containing the names of the elements.

```python
import collections

Person = collections.namedtuple('Person', 'name age gender')

print 'Type of Person:', type(Person)

bob = Person(name='Bob', age=30, gender='male')
print '\nRepresentation:', bob

jane = Person(name='Jane', age=29, gender='female')
print '\nField by name:', jane.name

print '\nFields by index:'
for p in [ bob, jane ]:
    print '%s is a %d year old %s' % p
```

As the example illustrates, it is possible to access the fields of the namedtuple by name using dotted notation (obj.attr) as well as using the positional indexes of standard tuples.
$ python collections_namedtuple_person.py

Type of Person: <type 'type'>

Representation: Person(name='Bob', age=30, gender='male')

Field by name: Jane

Fields by index:
Bob is a 30 year old male
Jane is a 29 year old female

Invalid Field Names

Field names are invalid if they are repeated or conflict with Python keywords.

```python
import collections
try:
    collections.namedtuple('Person', 'name class age gender')
except ValueError, err:
    print err

try:
    collections.namedtuple('Person', 'name age gender age')
except ValueError, err:
    print err
```

As the field names are parsed, invalid values cause ValueError exceptions.

$ python collections_namedtuple_bad_fields.py

Type names and field names cannot be a keyword: 'class'
Encountered duplicate field name: 'age'

If a namedtuple is being created based on values outside of the control of the program (such as to represent the rows returned by a database query, where the schema is not known in advance), set the rename option to True so the invalid fields are renamed.

```python
import collections

with_class = collections.namedtuple('Person', 'name class age gender', rename=True)
```
The new names for renamed fields depend on their index in the tuple, so the field with name class becomes _1 and the duplicate age field is changed to _3.

$ python collections_namedtuple_rename.py

('name', '_1', 'age', 'gender')
('name', 'age', 'gender', '_3')

2.1.5 OrderedDict

An OrderedDict is a dictionary subclass that remembers the order in which its contents are added.

```python
import collections

print 'Regular dictionary:'
d = {}
d['a'] = 'A'
d['b'] = 'B'
d['c'] = 'C'

for k, v in d.items():
    print k, v

print '
OrderedDict:'
d = collections.OrderedDict()
d['a'] = 'A'
d['b'] = 'B'
d['c'] = 'C'

for k, v in d.items():
    print k, v
```

A regular dict does not track the insertion order, and iterating over it produces the values in order based on how the keys are stored in the hash table. In an OrderedDict,
by contrast, the order in which the items are inserted is remembered and used when creating an iterator.

$ python collections_ordereddict_iter.py

Regular dictionary:
a A
c C
b B

OrderedDict:
a A
b B
c C

Equality
A regular dict looks at its contents when testing for equality. An OrderedDict also considers the order the items were added.

```python
import collections

print 'dict :',
d1 = {}
d1['a'] = 'A'
d1['b'] = 'B'
d1['c'] = 'C'

d2 = {}
d2['c'] = 'C'
d2['b'] = 'B'
d2['a'] = 'A'

print d1 == d2

print 'OrderedDict:',

d1 = collections.OrderedDict()
d1['a'] = 'A'
d1['b'] = 'B'
d1['c'] = 'C'
```
d2 = collections.OrderedDict()
d2['c'] = 'C'
d2['b'] = 'B'
d2['a'] = 'A'

print d1 == d2

In this case, since the two ordered dictionaries are created from values in a different order, they are considered to be different.

$ python collections_orderdict_equality.py

dict : True
OrderedDict: False

See Also:
collections (http://docs.python.org/library/collections.html) The standard library documentation for this module.

2.2 array—Sequence of Fixed-Type Data

**Purpose** Manage sequences of fixed-type numerical data efficiently.

**Python Version** 1.4 and later

The array module defines a sequence data structure that looks very much like a list, except that all members have to be of the same primitive type. Refer to the standard library documentation for array for a complete list of the types supported.

2.2.1 Initialization

An array is instantiated with an argument describing the type of data to be allowed, and possibly an initial sequence of data to store in the array.

```python
import array
import binascii

s = 'This is the array.'
a = array.array('c', s)

print 'As string:', s
print 'As array :', a
print 'As hex   :', binascii.hexlify(a)
```
In this example, the array is configured to hold a sequence of bytes and is initialized with a simple string.

$ python array_string.py

As string: This is the array.
As array : array('c', 'This is the array.')
As hex   : 54686973206973207468652061727261792e

2.2.2 Manipulating Arrays

An array can be extended and otherwise manipulated in the same ways as other Python sequences.

```python
import array
import pprint

a = array.array('i', xrange(3))
print 'Initial :', a

a.extend(xrange(3))
print 'Extended:', a

print 'Slice  :', a[2:5]

print 'Iterator:'
print list(enumerate(a))
```

The supported operations include slicing, iterating, and adding elements to the end.

$ python array_sequence.py

Initial : array('i', [0, 1, 2])
Extended: array('i', [0, 1, 2, 0, 1, 2])
Slice   : array('i', [2, 0, 1])
Iterator:
[(0, 0), (1, 1), (2, 2), (3, 0), (4, 1), (5, 2)]

2.2.3 Arrays and Files

The contents of an array can be written to and read from files using built-in methods coded efficiently for that purpose.
import array
import binascii
import tempfile

a = array.array('i', xrange(5))
print 'A1:', a

# Write the array of numbers to a temporary file
output = tempfile.NamedTemporaryFile()
a.tofile(output.file)  # must pass an *actual* file
output.flush()

# Read the raw data
with open(output.name, 'rb') as input:
    raw_data = input.read()
    print 'Raw Contents:', binascii.hexlify(raw_data)

    # Read the data into an array
    input.seek(0)
a2 = array.array('i')
a2.fromfile(input, len(a))
    print 'A2:', a2

This example illustrates reading the data raw, directly from the binary file, versus reading it into a new array and converting the bytes to the appropriate types.

$ python array_file.py
A1: array('i', [0, 1, 2, 3, 4])
Raw Contents: 00000000000000000000000000000000
A2: array('i', [0, 1, 2, 3, 4])

2.2.4 Alternate Byte Ordering

If the data in the array is not in the native byte order, or needs to be swapped before being sent to a system with a different byte order (or over the network), it is possible to convert the entire array without iterating over the elements from Python.

import array
import binascii

def to_hex(a):
    chars_per_item = a.itemsize * 2  # 2 hex digits
hex_version = binascii.hexlify(a)
num_chunks = len(hex_version) / chars_per_item

for i in xrange(num_chunks):
    start = i*chars_per_item
    end = start + chars_per_item
    yield hex_version[start:end]

a1 = array.array('i', xrange(5))
a2 = array.array('i', xrange(5))
a2.byteswap()

fmt = '%10s %10s %10s %10s'
print fmt % ('A1 hex', 'A1', 'A2 hex', 'A2')
print fmt % (('-') * 10,) * 4
for values in zip(to_hex(a1), a1, to_hex(a2), a2):
    print fmt % values

The byteswap() method switches the byte order of the items in the array from within C, so it is much more efficient than looping over the data in Python.

$ python array_byteswap.py

<table>
<thead>
<tr>
<th>A1 hex</th>
<th>A1</th>
<th>A2 hex</th>
<th>A2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01000000</td>
<td>1</td>
<td>00000001</td>
<td>16777216</td>
</tr>
<tr>
<td>02000000</td>
<td>2</td>
<td>00000002</td>
<td>33554432</td>
</tr>
<tr>
<td>03000000</td>
<td>3</td>
<td>00000003</td>
<td>50331648</td>
</tr>
<tr>
<td>04000000</td>
<td>4</td>
<td>00000004</td>
<td>67108864</td>
</tr>
</tbody>
</table>

See Also:
array (http://docs.python.org/library/array.html) The standard library documentation for this module.

struct (page 102) The struct module.

Numerical Python (www.scipy.org) NumPy is a Python library for working with large data sets efficiently.

2.3  heapq—Heap Sort Algorithm

Purpose The heapq module implements a min-heap sort algorithm suitable for use with Python’s lists.

Python Version New in 2.3 with additions in 2.5
A heap is a tree-like data structure where the child nodes have a sort-order relationship with the parents. Binary heaps can be represented using a list or an array organized so that the children of element N are at positions 2*N+1 and 2*N+2 (for zero-based indexes). This layout makes it possible to rearrange heaps in place, so it is not necessary to reallocate as much memory when adding or removing items.

A max-heap ensures that the parent is larger than or equal to both of its children. A min-heap requires that the parent be less than or equal to its children. Python’s heapq module implements a min-heap.

### 2.3.1 Example Data

The examples in this section use the data in `heapq_heapdata.py`.

```python
# This data was generated with the random module.

data = [19, 9, 4, 10, 11]

The heap output is printed using `heapq_showtree.py`.

```
2.3.2 Creating a Heap

There are two basic ways to create a heap: `heappush()` and `heapify()`.

```python
import heapq
from heapq_showtree import show_tree
from heapq_heapdata import data

heap = []
print('random :', data)
print
for n in data:
    print('add %3d:' % n)
    heapq.heappush(heap, n)
    show_tree(heap)

Using `heappush()`, the heap sort order of the elements is maintained as new items are added from a data source.
```

```bash
$ python heapq_heappush.py
random : [19, 9, 4, 10, 11]
add 19:

19
------------------------------------
add 9:

9
  19
------------------------------------
add 4:

4
  19   9
------------------------------------
add 10:

4
```

If the data is already in memory, it is more efficient to use `heapify()` to rearrange the items of the list in place.

```python
import heapq
from heapq_showtree import show_tree
from heapq_heapdata import data

print('random:', data)
heapq.heapify(data)
print('heapified: ')
show_tree(data)
```

The result of building a list in heap order one item at a time is the same as building it unordered and then calling `heapify()`.

```bash
$ python heapq_heapify.py
random : [19, 9, 4, 10, 11]
heapified :

        4
       / \
      9   19
     /   / \
    10  11
```

### 2.3.3 Accessing Contents of a Heap

Once the heap is organized correctly, use `heappop()` to remove the element with the lowest value.

```python
import heapq
from heapq_showtree import show_tree
from heapq_heapdata import data
```
print 'random :', data
heapq.heapify(data)
print 'heapified :'
show_tree(data)

for i in xrange(2):
    smallest = heapq.heappop(data)
    print 'pop %3d:' % smallest
    show_tree(data)

In this example, adapted from the stdlib documentation, \texttt{heapify()} and \texttt{heappop()} are used to sort a list of numbers.

```
$ python heapq_heappop.py
random : [19, 9, 4, 10, 11]
heapified :

    4
   /|
  9 19
 /|
10 11

------------------------------------

pop 4:

    9
   /|
  10 19
 /|
11

------------------------------------

pop 9:

    10
   /|
  11 19

------------------------------------
```

To remove existing elements and replace them with new values in a single operation, use \texttt{heapreplace()}.

```
import heapq
from heapq_showtree import show_tree
from heapq_heapdata import data
```
heapq.heapify(data)
print 'start:'
show_tree(data)

for n in [0, 13]:
    smallest = heapq.heapreplace(data, n)
    print 'replace %2d with %2d:' % (smallest, n)
show_tree(data)

Replacing elements in place makes it possible to maintain a fixed-size heap, such as a queue of jobs ordered by priority.

$ python heapq_heapreplace.py

start:

\[
\begin{array}{cccc}
4 & & & \\
9 & 19 & & \\
10 & 11 & & \\
\end{array}
\]

replace 4 with 0:

\[
\begin{array}{cccc}
0 & & & \\
9 & 19 & & \\
10 & 11 & & \\
\end{array}
\]

replace 0 with 13:

\[
\begin{array}{cccc}
9 & & & \\
10 & 19 & & \\
13 & 11 & & \\
\end{array}
\]

2.3.4 Data Extremes from a Heap

heapq also includes two functions to examine an iterable to find a range of the largest or smallest values it contains.

import heapq
from heapq_heapdata import data
2.4. bisect—Maintain Lists in Sorted Order

Purpose

Maintains a list in sorted order without having to call sort each time an item is added to the list.

Python Version

1.4 and later

The bisect module implements an algorithm for inserting elements into a list while maintaining the list in sorted order. For some cases, this is more efficient than repeatedly sorting a list or explicitly sorting a large list after it is constructed.

2.4.1 Inserting in Sorted Order

Here is a simple example using insort() to insert items into a list in sorted order.
import bisect
import random

# Use a constant seed to ensure that
# the same pseudo-random numbers
# are used each time the loop is run.
random.seed(1)

print 'New  Pos  Contents'
print '---  ---  --------'

# Generate random numbers and
# insert them into a list in sorted
# order.
l = []
for i in range(1, 15):
    r = random.randint(1, 100)
    position = bisect.bisect(l, r)
    bisect.insort(l, r)
    print '%3d  %3d  ' % (r, position), l

The first column of the output shows the new random number. The second column shows the position where the number will be inserted into the list. The remainder of each line is the current sorted list.

$ python bisect_example.py

New  Pos  Contents
---  ---  --------
 14   0  [14]
 85   1  [14, 85]
 77   1  [14, 77, 85]
 26   1  [14, 26, 77, 85]
 50   2  [14, 26, 50, 77, 85]
 45   2  [14, 26, 45, 50, 77, 85]
 66   4  [14, 26, 45, 50, 66, 77, 85]
 79   6  [14, 26, 45, 50, 66, 77, 79, 85]
 10   0  [10, 14, 26, 45, 50, 66, 77, 79, 85]
  3   0  [3, 10, 14, 26, 45, 50, 66, 77, 79, 85]
 84   9  [3, 10, 14, 26, 45, 50, 66, 77, 79, 84, 85]
 44   4  [3, 10, 14, 26, 44, 45, 50, 66, 77, 79, 84, 85]
 77   9  [3, 10, 14, 26, 44, 45, 50, 66, 77, 77, 79, 84, 85]
  1   0  [1, 3, 10, 14, 26, 44, 45, 50, 66, 77, 77, 79, 84, 85]
This is a simple example, and for the amount of data being manipulated, it might be faster to simply build the list and then sort it once. But for long lists, significant time and memory savings can be achieved using an insertion sort algorithm such as this one.

### 2.4.2 Handling Duplicates

The result set shown previously includes a repeated value, 77. The `bisect` module provides two ways to handle repeats. New values can be inserted to the left of existing values or to the right. The `insort()` function is actually an alias for `insert_right()`, which inserts after the existing value. The corresponding function `insert_left()` inserts before the existing value.

```python
import bisect
import random

# Reset the seed
random.seed(1)

print 'New  Pos  Contents'
print '---  ---  --------'

# Use bisect_left and insert_left.
for i in range(1, 15):
    r = random.randint(1, 100)
    position = bisect.bisect_left(l, r)
    bisect.insort_left(l, r)
    print '%3d  %3d' % (r, position), l
```

When the same data is manipulated using `bisect_left()` and `insert_left()`, the results are the same sorted list, but the insert positions are different for the duplicate values.

$ python bisect_example2.py

<table>
<thead>
<tr>
<th>New</th>
<th>Pos</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>0</td>
<td>[14]</td>
</tr>
<tr>
<td>85</td>
<td>1</td>
<td>[14, 85]</td>
</tr>
<tr>
<td>77</td>
<td>1</td>
<td>[14, 77, 85]</td>
</tr>
<tr>
<td>26</td>
<td>1</td>
<td>[14, 26, 77, 85]</td>
</tr>
<tr>
<td>50</td>
<td>2</td>
<td>[14, 26, 50, 77, 85]</td>
</tr>
<tr>
<td>45</td>
<td>2</td>
<td>[14, 26, 45, 50, 77, 85]</td>
</tr>
</tbody>
</table>
In addition to the Python implementation, a faster C implementation is available. If the C version is present, that implementation automatically overrides the pure Python implementation when `bisect` is imported.

**See Also:**
- `bisect` ([http://docs.python.org/library/bisect.html](http://docs.python.org/library/bisect.html)) The standard library documentation for this module.

## 2.5 Queue—Thread-Safe FIFO Implementation

**Purpose** Provides a thread-safe FIFO implementation.

**Python Version** At least 1.4

The `Queue` module provides a first-in, first-out (FIFO) data structure suitable for multithreaded programming. It can be used to pass messages or other data safely between producer and consumer threads. Locking is handled for the caller, so many threads can work with the same `Queue` instance safely. The size of a `Queue` (the number of elements it contains) may be restricted to throttle memory usage or processing.

**Note:** This discussion assumes you already understand the general nature of a queue. If you do not, you may want to read some of the references before continuing.

### 2.5.1 Basic FIFO Queue

The `Queue` class implements a basic first-in, first-out container. Elements are added to one end of the sequence using `put()`, and removed from the other end using `get()`.
import Queue

q = Queue.Queue()

for i in range(5):
    q.put(i)

while not q.empty():
    print q.get(),

This example uses a single thread to illustrate that elements are removed from the
queue in the same order they are inserted.

$ python Queue_fifo.py
0 1 2 3 4

2.5.2 LIFO Queue

In contrast to the standard FIFO implementation of Queue, the LifoQueue uses last-in,
first-out (LIFO) ordering (normally associated with a stack data structure).

import Queue

q = Queue.LifoQueue()

for i in range(5):
    q.put(i)

while not q.empty():
    print q.get(),

The item most recently put into the queue is removed by get.

$ python Queue_lifo.py
4 3 2 1 0
2.5.3 Priority Queue

Sometimes, the processing order of the items in a queue needs to be based on characteristics of those items, rather than just on the order in which they are created or added to the queue. For example, print jobs from the payroll department may take precedence over a code listing printed by a developer. `PriorityQueue` uses the sort order of the contents of the queue to decide which to retrieve.

```python
import Queue
import threading

class Job(object):
    def __init__(self, priority, description):
        self.priority = priority
        self.description = description
        print 'New job:', description
        return
    def __cmp__(self, other):
        return cmp(self.priority, other.priority)

q = Queue.PriorityQueue()
q.put(Job(3, 'Mid-level job'))
q.put(Job(10, 'Low-level job'))
q.put(Job(1, 'Important job'))

def process_job(q):
    while True:
        next_job = q.get()
        print 'Processing job:', next_job.description
        q.task_done()

workers = [ threading.Thread(target=process_job, args=(q,)),
            threading.Thread(target=process_job, args=(q,)),
        ]
for w in workers:
    w.setDaemon(True)
    w.start()

q.join()
```

This example has multiple threads consuming the jobs, which are to be processed based on the priority of items in the queue at the time `get()` was called. The order
of processing for items added to the queue while the consumer threads are running depends on thread context switching.

```bash
$ python Queue_priority.py
New job: Mid-level job
New job: Low-level job
New job: Important job
Processing job: Important job
Processing job: Mid-level job
Processing job: Low-level job
```

### 2.5.4 Building a Threaded Podcast Client

The source code for the podcasting client in this section demonstrates how to use the `Queue` class with multiple threads. The program reads one or more RSS feeds, queues up the enclosures for the five most recent episodes to be downloaded, and processes several downloads in parallel using threads. It does not have enough error handling for production use, but the skeleton implementation provides an example of how to use the `Queue` module.

First, some operating parameters are established. Normally, these would come from user inputs (preferences, a database, etc.). The example uses hard-coded values for the number of threads and a list of URLs to fetch.

```python
from Queue import Queue
from threading import Thread
import time
import urllib
import urlparse
import feedparser

# Set up some global variables
num_fetch_threads = 2
enclosure_queue = Queue()

# A real app wouldn’t use hard-coded data...
feed_urls = [ 'http://advocacy.python.org/podcasts/littlebit.rss', ]

The function `downloadEnclosures()` will run in the worker thread and process the downloads using `urllib`.```
def downloadEnclosures(i, q):
    """This is the worker thread function.
    It processes items in the queue one after another. These daemon threads go into an
    infinite loop, and only exit when the main thread ends.
    """
    while True:
        print 'Looking for the next enclosure' % i
        url = q.get()
        parsed_url = urlparse.urlparse(url)
        print 'Downloading:' % i, parsed_url.path
        response = urllib.urlopen(url)
        data = response.read()
        # Save the downloaded file to the current directory
        outfile_name = url.rpartition('/')[2]
        with open(outfile_name, 'wb') as outfile:
            outfile.write(data)
        q.task_done()

Once the threads’ target function is defined, the worker threads can be started. When `downloadEnclosures()` processes the statement `url = q.get()`, it blocks and waits until the queue has something to return. That means it is safe to start the threads before there is anything in the queue.

# Set up some threads to fetch the enclosures
for i in range(num_fetch_threads):
    worker = Thread(target=downloadEnclosures,
                    args=(i, enclosure_queue,))
    worker.setDaemon(True)
    worker.start()

The next step is to retrieve the feed contents using Mark Pilgrim’s `feedparser` module (www.feedparser.org) and enqueue the URLs of the enclosures. As soon as the first URL is added to the queue, one of the worker threads picks it up and starts downloading it. The loop will continue to add items until the feed is exhausted, and the worker threads will take turns dequing URLs to download them.

# Download the feed(s) and put the enclosure URLs into
# the queue.
for url in feed_urls:
    response = feedparser.parse(url, agent='fetch_podcasts.py')
for entry in response['entries'][-5:]:
    for enclosure in entry.get('enclosures', []):
        parsed_url = urlparse.urlparse(enclosure['url'])
        print 'Queuing:', parsed_url.path
        enclosure_queue.put(enclosure['url'])

    The only thing left to do is wait for the queue to empty out again, using join().

    # Now wait for the queue to be empty, indicating that we have
    # processed all the downloads.
    print '*** Main thread waiting'
    enclosure_queue.join()
    print '*** Done'

Running the sample script produces the following.

$ python fetch_podcasts.py

0: Looking for the next enclosure
1: Looking for the next enclosure
Queuing: /podcasts/littlebit/2010-04-18.mp3
Queuing: /podcasts/littlebit/2010-05-22.mp3
Queuing: /podcasts/littlebit/2010-06-06.mp3
Queuing: /podcasts/littlebit/2010-07-26.mp3
Queuing: /podcasts/littlebit/2010-11-25.mp3
*** Main thread waiting
0: Downloading: /podcasts/littlebit/2010-04-18.mp3
0: Looking for the next enclosure
0: Downloading: /podcasts/littlebit/2010-05-22.mp3
0: Looking for the next enclosure
0: Downloading: /podcasts/littlebit/2010-06-06.mp3
0: Looking for the next enclosure
0: Downloading: /podcasts/littlebit/2010-07-26.mp3
0: Looking for the next enclosure
0: Downloading: /podcasts/littlebit/2010-11-25.mp3
0: Looking for the next enclosure
*** Done

The actual output will depend on the contents of the RSS feed used.

See Also:
Queue (http://docs.python.org/lib/module-Queue.html) Standard library documentation for this module.
Deque (page 75) from collections (page 70) The collections module includes a deque (double-ended queue) class.


2.6 struct—Binary Data Structures

Purpose Convert between strings and binary data.

Python Version 1.4 and later

The struct module includes functions for converting between strings of bytes and native Python data types, such as numbers and strings.

2.6.1 Functions vs. Struct Class

There is a set of module-level functions for working with structured values, and there is also the Struct class. Format specifiers are converted from their string format to a compiled representation, similar to the way regular expressions are handled. The conversion takes some resources, so it is typically more efficient to do it once when creating a Struct instance and call methods on the instance, instead of using the module-level functions. The following examples all use the Struct class.

2.6.2 Packing and Unpacking

.Structs support packing data into strings and unpacking data from strings using format specifiers made up of characters representing the data type and optional count and endianness indicators. Refer to the standard library documentation for a complete list of the supported format specifiers.

In this example, the specifier calls for an integer or long value, a two-character string, and a floating-point number. The spaces in the format specifier are included to separate the type indicators and are ignored when the format is compiled.

```python
import struct
import binascii

values = (1, 'ab', 2.7)
s = struct.Struct('I 2s f')
packed_data = s.pack(*values)
```
2.6. struct—Binary Data Structures

print 'Original values:', values
print 'Format string :', s.format
print 'Uses :', s.size, 'bytes'
print 'Packed Value :', binascii.hexlify(packed_data)

The example converts the packed value to a sequence of hex bytes for printing with binascii.hexlify(), since some characters are nulls.

$ python struct_pack.py

Original values: (1, 'ab', 2.7)
Format string : I 2s f
Uses : 12 bytes
Packed Value : 0100000061620000cdcc2c40

Use unpack() to extract data from its packed representation.

import struct
import binascii

packed_data = binascii.unhexlify('0100000061620000cdcc2c40')

s = struct.Struct('I 2s f')
unpacked_data = s.unpack(packed_data)
print 'Unpacked Values:', unpacked_data

Passing the packed value to unpack() gives basically the same values back (note the discrepancy in the floating-point value).

$ python struct_unpack.py

Unpacked Values: (1, 'ab', 2.700000047683716)

2.6.3 Endianness

By default, values are encoded using the native C library notion of *endianness*. It is easy to override that choice by providing an explicit endianness directive in the format string.

import struct
import binascii
values = (1, 'ab', 2.7)
print 'Original values:', values
endianness = [
('@', 'native, native'),
('=', 'native, standard'),
('<', 'little-endian'),
('>', 'big-endian'),
('!', 'network'),
]

for code, name in endianness:
s = struct.Struct(code + 'I2sf')
packed_data = s.pack(*values)
print
print 'Format string :', s.format, 'for', name
print 'Uses :', s.size, 'bytes'
print 'Packed Value :', binascii.hexlify(packed_data)
print 'Unpacked Value :', s.unpack(packed_data)

Table 2.1 lists the byte order specifiers used by Struct.

Table 2.1. Byte Order Specifiers for struct

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>@</td>
<td>Native order</td>
</tr>
<tr>
<td>=</td>
<td>Native standard</td>
</tr>
<tr>
<td>&lt;</td>
<td>Little-endian</td>
</tr>
<tr>
<td>&gt;</td>
<td>Big-endian</td>
</tr>
<tr>
<td>!</td>
<td>Network order</td>
</tr>
</tbody>
</table>

$ python struct_endianness.py

Original values: (1, 'ab', 2.7)

Format string : @ I 2s f for native, native
Uses : 12 bytes
Packed Value : 0100000061620000cdcc2c40
Unpacked Value : (1, 'ab', 2.700000047683716)

Format string : = I 2s f for native, standard
Uses : 10 bytes
Packed Value : 010000006162cdcc2c40
2.6.4 Buffers

Working with binary packed data is typically reserved for performance-sensitive situations or when passing data into and out of extension modules. These cases can be optimized by avoiding the overhead of allocating a new buffer for each packed structure. The `pack_into()` and `unpack_from()` methods support writing to preallocated buffers directly.

```python
import struct
import binascii

s = struct.Struct('I 2s f')
values = (1, 'ab', 2.7)
print('Original:', values)

print('ctypes string buffer')

import ctypes
b = ctypes.create_string_buffer(s.size)
print('Before :', binascii.hexlify(b.raw)
s.pack_into(b, 0, *values)
print('After :', binascii.hexlify(b.raw)
print('Unpacked:', s.unpack_from(b, 0))
```
print
print 'array'

import array
a = array.array('c', '\0' * s.size)
print 'Before :', binascii.hexlify(a)
s.pack_into(a, 0, *values)
print 'After :', binascii.hexlify(a)
print 'Unpacked:', s.unpack_from(a, 0)

The size attribute of the Struct tells us how big the buffer needs to be.

$ python struct_buffers.py

Original: (1, 'ab', 2.7)

catypes string buffer
Before : 000000000000000000000000
After : 0100000061620000cdcc2c40
Unpacked: (1, 'ab', 2.700000047683716)

array
Before : 000000000000000000000000
After : 0100000061620000cdcc2c40
Unpacked: (1, 'ab', 2.700000047683716)

See Also:
struct (http://docs.python.org/library/struct.html) The standard library documentation for this module.
array (page 84) The array module, for working with sequences of fixed-type values.
binascii (http://docs.python.org/library/binascii.html) The binascii module, for producing ASCII representations of binary data.

2.7 weakref—Impermanent References to Objects

Purpose Refer to an “expensive” object, but allow its memory to be reclaimed by the garbage collector if there are no other nonweak references.

Python Version 2.1 and later
The `weakref` module supports weak references to objects. A normal reference increments the reference count on the object and prevents it from being garbage collected. This is not always desirable, either when a circular reference might be present or when building a cache of objects that should be deleted when memory is needed. A weak reference is a handle to an object that does not keep it from being cleaned up automatically.

### 2.7.1 References

Weak references to objects are managed through the `ref` class. To retrieve the original object, call the reference object.

```python
import weakref
class ExpensiveObject(object):
    def __del__(self):
        print '(Deleting %s)' % self

obj = ExpensiveObject()
r = weakref.ref(obj)

print 'obj:', obj
print 'ref:', r
print 'r():', r()

print 'deleting obj'
del obj
print 'r():', r()
```

In this case, since `obj` is deleted before the second call to the reference, the `ref` returns `None`.

```
$ python weakref_ref.py
obj: <__main__.ExpensiveObject object at 0x100da5750>
ref: <weakref at 0x100d99b50; to 'ExpensiveObject' at 0x100da5750>
r(): <__main__.ExpensiveObject object at 0x100da5750>
deleting obj
(Deleting <__main__.ExpensiveObject object at 0x100da5750>)
r(): None
```
2.7.2 Reference Callbacks

The `ref` constructor accepts an optional callback function to invoke when the referenced object is deleted.

```python
import weakref

class ExpensiveObject(object):
    def __del__(self):
        print '(Deleting %s)' % self

def callback(reference):
    """Invoked when referenced object is deleted"""
    print 'callback(', reference, ')

obj = ExpensiveObject()
ref = weakref.ref(obj, callback)

print 'obj:', obj
print 'ref:', ref
print 'r():', ref()

deleting obj
del obj
print 'r():', ref()
```

The callback receives the reference object as an argument after the reference is “dead” and no longer refers to the original object. One use for this feature is to remove the weak reference object from a cache.

$ python weakref_ref_callback.py

```
obj: <__main__.ExpensiveObject object at 0x100da1950>
ref: <weakref at 0x100d99ba8; to 'ExpensiveObject' at 0x100da1950>
r(): <__main__.ExpensiveObject object at 0x100da1950>
deleting obj
callback( <weakref at 0x100d99ba8; dead> )
(Deleting <__main__.ExpensiveObject object at 0x100da1950>)
r(): None
```

2.7.3 Proxies

It is sometimes more convenient to use a proxy, rather than a weak reference. Proxies can be used as though they were the original object and do not need to be called before
the object is accessible. That means they can be passed to a library that does not know it is receiving a reference instead of the real object.

```python
import weakref

class ExpensiveObject(object):
    def __init__(self, name):
        self.name = name
    def __del__(self):
        print '(Deleting %s)' % self

obj = ExpensiveObject('My Object')
r = weakref.ref(obj)
p = weakref.proxy(obj)

print 'via obj:', obj.name
print 'via ref:', r().name
print 'via proxy:', p.name
del obj
print 'via proxy:', p.name

If the proxy is accessed after the referent object is removed, a ReferenceError exception is raised.

$ python weakref_proxy.py

via obj: My Object
via ref: My Object
via proxy: My Object
(Deleting <__main__.ExpensiveObject object at 0x100da27d0>)
via proxy:
Traceback (most recent call last):
  File "weakref_proxy.py", line 26, in <module>
    print 'via proxy:', p.name
ReferenceError: weakly-referenced object no longer exists

2.7.4 Cyclic References

One use for weak references is to allow cyclic references without preventing garbage collection. This example illustrates the difference between using regular objects and proxies when a graph includes a cycle.

The `Graph` class in `weakref_graph.py` accepts any object given to it as the “next” node in the sequence. For the sake of brevity, this implementation supports
a single outgoing reference from each node, which is of limited use generally, but makes it easy to create cycles for these examples. The function `demo()` is a utility function to exercise the `Graph` class by creating a cycle and then removing various references.

```python
import gc
from pprint import pprint
import weakref

class Graph(object):
    def __init__(self, name):
        self.name = name
        self.other = None
    def set_next(self, other):
        print '%s.set_next(%r)' % (self.name, other)
        self.other = other
    def all_nodes(self):
        "Generate the nodes in the graph sequence."
        yield self
        n = self.other
        while n and n.name != self.name:
            yield n
            n = n.other
        if n is self:
            yield n
        return
    def __str__(self):
        return '->'.join(n.name for n in self.all_nodes())
    def __repr__(self):
        return '<%s at 0x%x name=%s>' % (self.__class__.__name__,
                                           id(self), self.name)
    def __del__(self):
        print '(Deleting %s)' % self.name
        self.set_next(None)

def collect_and_show_garbage():
    "Show what garbage is present."
    print 'Collecting...'
    n = gc.collect()
    print 'Unreachable objects:', n
    print 'Garbage:',
    pprint(gc.garbage)
```
def demo(graph_factory):
    print 'Set up graph:'
    one = graph_factory('one')
    two = graph_factory('two')
    three = graph_factory('three')
    one.set_next(two)
    two.set_next(three)
    three.set_next(one)

    print
    print 'Graph:'
    print str(one)
    collect_and_show_garbage()

    print
    three = None
    two = None
    print 'After 2 references removed:'
    print str(one)
    collect_and_show_garbage()

    print
    print 'Removing last reference:'
    one = None
    collect_and_show_garbage()

This example uses the gc module to help debug the leak. The DEBUG_LEAK flag causes gc to print information about objects that cannot be seen, other than through the reference the garbage collector has to them.

import gc
from pprint import pprint
import weakref

from weakref_graph import Graph, demo, collect_and_show_garbage

gc.set_debug(gc.DEBUG_LEAK)

print 'Setting up the cycle'
print
demo(Graph)
print
print 'Breaking the cycle and cleaning up garbage'
print
gc.garbage[0].set_next(None)
while gc.garbage:
    del gc.garbage[0]
print
collect_and_show_garbage()

Even after deleting the local references to the Graph instances in `demo()`, the graphs all show up in the garbage list and cannot be collected. Several dictionaries are also found in the garbage list. They are the `__dict__` values from the Graph instances and contain the attributes for those objects. The graphs can be forcibly deleted, since the program knows what they are. Enabling unbuffered I/O by passing the `-u` option to the interpreter ensures that the output from the `print` statements in this example program (written to standard output) and the debug output from `gc` (written to standard error) are interleaved correctly.

$ python -u weakref_cycle.py

Setting up the cycle

Set up graph:
one.set_next(<Graph at 0x100db7590 name=two>)
two.set_next(<Graph at 0x100db75d0 name=three>)
three.set_next(<Graph at 0x100db7550 name=one>)

Graph:
one->two->three->one
Collecting...
Unreachable objects: 0
Garbage:[]

After 2 references removed:
one->two->three->one
Collecting...
Unreachable objects: 0
Garbage:[]

Removing last reference:
Collecting...
gc: uncollectable <Graph 0x100db7550>
gc: uncollectable <Graph 0x100db7590>
The next step is to create a more intelligent `WeakGraph` class that knows how to avoid creating cycles with regular references by using weak references when a cycle is detected.

```python
import gc
from pprint import pprint
import weakref

from weakref_graph import Graph, demo

class WeakGraph(Graph):
    def set_next(self, other):
        if other is not None:
            # See if we should replace the reference to other with a weakref.
            if self in other.all_nodes():
                other = weakref.proxy(other)
```

Unreachable objects: 0
Garbage:[]
super(WeakGraph, self).set_next(other)
return
demo(WeakGraph)

Since the WeakGraph instances use proxies to refer to objects that have already been seen, as demo() removes all local references to the objects, the cycle is broken and the garbage collector can delete the objects.

$ python weakref_weakgraph.py

Set up graph:
one.set_next(<WeakGraph at 0x100db4790 name=two>)
two.set_next(<WeakGraph at 0x100db47d0 name=three>)
three.set_next(<weakproxy at 0x100dac6d8 to WeakGraph at 0x100db4750>)

Graph:
one->two->three
Collecting...
Unreachable objects: 0
Garbage:[]

After 2 references removed:
one->two->three
Collecting...
Unreachable objects: 0
Garbage:[]

Removing last reference:
(Deleting one)
one.set_next(None)
(Deleting two)
two.set_next(None)
(Deleting three)
three.set_next(None)
Collecting...
Unreachable objects: 0
Garbage:[]

2.7.5 Caching Objects

The ref and proxy classes are considered “low level.” While they are useful for maintaining weak references to individual objects and allowing cycles to be garbage
collected, the WeakKeyDictionary and WeakValueDictionary provide a more appropriate API for creating a cache of several objects.

The WeakValueDictionary uses weak references to the values it holds, allowing them to be garbage collected when other code is not actually using them. Using explicit calls to the garbage collector illustrates the difference between memory handling with a regular dictionary and WeakValueDictionary.

```python
import gc
from pprint import pprint
import weakref

gc.set_debug(gc.DEBUG_LEAK)

class ExpensiveObject(object):
    def __init__(self, name):
        self.name = name
    def __repr__(self):
        return 'ExpensiveObject(%s)' % self.name
    def __del__(self):
        print ' (Deleting %s)' % self

def demo(cache_factory):
    # hold objects so any weak references
    # are not removed immediately
    all_refs = {}
    # create the cache using the factory
    print 'CACHE TYPE: ', cache_factory
    cache = cache_factory()
    for name in [ 'one', 'two', 'three' ]:
        o = ExpensiveObject(name)
        cache[name] = o
        all_refs[name] = o
        del o # decref

    print ' all_refs =',
    pprint(all_refs)
    print '\n Before, cache contains: ', cache.keys()
    for name, value in cache.items():
        print ' %s = %s' % (name, value)
        del value # decref

    # Remove all references to the objects except the cache
    print '\n Cleanup:'
```


def all_refs
    gc.collect()

    print '\n  After, cache contains:', cache.keys()
    for name, value in cache.items():
        print '    %s = %s' % (name, value)
    print '  demo returning'
    return
demo(dict)
print
demo(weakref.WeakValueDictionary)

    Any loop variables that refer to the values being cached must be cleared explicitly
    so the reference count of the object is decremented. Otherwise, the garbage collec-
    tor would not remove the objects, and they would remain in the cache. Similarly, the
    all_refs variable is used to hold references to prevent them from being garbage collected
    prematurely.

$ python weakref_valuedict.py

CACHE TYPE: <type 'dict'>
    all_refs ={'one': ExpensiveObject(one),
          'three': ExpensiveObject(three),
          'two': ExpensiveObject(two)}

Before, cache contains: ['three', 'two', 'one']
    three = ExpensiveObject(three)
    two = ExpensiveObject(two)
    one = ExpensiveObject(one)

Cleanup:

After, cache contains: ['three', 'two', 'one']
    three = ExpensiveObject(three)
    two = ExpensiveObject(two)
    one = ExpensiveObject(one)
demo returning
    (Deleting ExpensiveObject(three))
    (Deleting ExpensiveObject(two))
    (Deleting ExpensiveObject(one))
CACHE TYPE: weakref.WeakValueDictionary
all_refs ={‘one’: ExpensiveObject(one),
‘three’: ExpensiveObject(three),
‘two’: ExpensiveObject(two)}

Before, cache contains: [‘three’, ’two’, ’one’]
three = ExpensiveObject(three)
two = ExpensiveObject(two)
one = ExpensiveObject(one)

Cleanup:
(Deleting ExpensiveObject(three))
(Deleting ExpensiveObject(two))
(Deleting ExpensiveObject(one))

After, cache contains: []
demo returning

The WeakKeyDictionary works similarly, but it uses weak references for the keys instead of the values in the dictionary.

WARNING: The library documentation for weakref contains this warning:
Caution: Because a WeakValueDictionary is built on top of a Python dictionary, it must not change size when iterating over it. This can be difficult to ensure for a WeakValueDictionary because actions performed by the program during iteration may cause items in the dictionary to vanish “by magic” (as a side effect of garbage collection).

See Also:
weakref (http://docs.python.org/lib/module-weakref.html) Standard library documentation for this module.
gc (page 1138) The gc module is the interface to the interpreter’s garbage collector.

2.8 copy—Duplicate Objects

Purpose Provides functions for duplicating objects using shallow or deep copy semantics.

Python Version 1.4 and later
The \texttt{copy} module includes two functions, \texttt{copy()} and \texttt{deepcopy()}, for duplicating existing objects.

### 2.8.1 Shallow Copies

The shallow copy created by \texttt{copy()} is a new container populated with references to the contents of the original object. When making a shallow copy of a list object, a new list is constructed and the elements of the original object are appended to it.

```python
import copy
class MyClass:
    def \_init\_(self, name):
        self.name = name
    def \_cmp\_(self, other):
        return cmp(self.name, other.name)
a = MyClass('a')
my_list = [a]
dup = copy.copy(my_list)

print '  my_list:', my_list
print '  dup:', dup
print '  dup is my_list:', (dup is my_list)
print '  dup == my_list:', (dup == my_list)
print '  dup[0] is my_list[0]:', (dup[0] is my_list[0])
print '  dup[0] == my_list[0]:', (dup[0] == my_list[0])
```

For a shallow copy, the \texttt{MyClass} instance is not duplicated, so the reference in the \texttt{dup} list is to the same object that is in \texttt{my_list}.

```
$ python copy_shallow.py
```

```
my_list: [<__main__.MyClass instance at 0x100dadc68>]
dup: [<__main__.MyClass instance at 0x100dadc68>]
dup is my_list: False
dup == my_list: True
dup[0] is my_list[0]: True
dup[0] == my_list[0]: True
```

### 2.8.2 Deep Copies

The deep copy created by \texttt{deepcopy()} is a new container populated with copies of the contents of the original object. To make a deep copy of a list, a new list

```python
...```
is constructed, the elements of the original list are copied, and then those copies are appended to the new list.

Replacing the call to copy() with deepcopy() makes the difference in the output apparent.

dup = copy.deepcopy(my_list)

The first element of the list is no longer the same object reference, but when the two objects are compared, they still evaluate as being equal.

$ python copy_deep.py

  my_list: [<__main__.MyClass instance at 0x100dadc68>]
  dup: [<__main__.MyClass instance at 0x100dadc20>]
  dup is my_list: False
  dup == my_list: True
dup[0] is my_list[0]: False
dup[0] == my_list[0]: True

2.8.3 Customizing Copy Behavior

It is possible to control how copies are made using the __copy__() and __deepcopy__() special methods.

• __copy__() is called without any arguments and should return a shallow copy of the object.
• __deepcopy__() is called with a memo dictionary and should return a deep copy of the object. Any member attributes that need to be deep-copied should be passed to copy.deepcopy(), along with the memo dictionary, to control for recursion. (The memo dictionary is explained in more detail later.)

This example illustrates how the methods are called.

import copy

class MyClass:
    def __init__(self, name):
        self.name = name
    def __cmp__(self, other):
        return cmp(self.name, other.name)
def __copy__(self):
    print '__copy__()'
    return MyClass(self.name)

def __deepcopy__(self, memo):
    print '__deepcopy__(%s)' % str(memo)
    return MyClass(copy.deepcopy(self.name, memo))

a = MyClass('a')
sc = copy.copy(a)
dc = copy.deepcopy(a)

The memo dictionary is used to keep track of the values that have been copied already, to avoid infinite recursion.

$ python copy_hooks.py
__copy__()
__deepcopy__({})

2.8.4 Recursion in Deep Copy

To avoid problems with duplicating recursive data structures, deepcopy() uses a dictionary to track objects that have already been copied. This dictionary is passed to the __deepcopy__() method so it can be examined there as well.

This example shows how an interconnected data structure, such as a directed graph, can assist with protecting against recursion by implementing a __deepcopy__() method.

import copy
import pprint

class Graph:
    def __init__(self, name, connections):
        self.name = name
        self.connections = connections

    def add_connection(self, other):
        self.connections.append(other)

    def __repr__(self):
        return 'Graph(name=%s, id=%s)' % (self.name, id(self))
def __deepcopy__(self, memo):
    print '\nCalling __deepcopy__ for %r' % self
    if self in memo:
        existing = memo.get(self)
        print ' Already copied to %r' % existing
        return existing
    print ' Memo dictionary:'
    pprint.pprint(memo, indent=4, width=40)
    dup = Graph(copy.deepcopy(self.name, memo), [])
    print ' Copying to new object %s' % dup
    memo[self] = dup
    for c in self.connections:
        dup.add_connection(copy.deepcopy(c, memo))
    return dup

root = Graph('root', [])
a = Graph('a', [root])
b = Graph('b', [a, root])
root.add_connection(a)
root.add_connection(b)
dup = copy.deepcopy(root)

The Graph class includes a few basic directed-graph methods. An instance can be initialized with a name and a list of existing nodes to which it is connected. The add_connection() method is used to set up bidirectional connections. It is also used by the deepcopy operator.

The __deepcopy__() method prints messages to show how it is called and manages the memo dictionary contents, as needed. Instead of copying the connection list wholesale, it creates a new list and appends copies of the individual connections to it. That ensures that the memo dictionary is updated as each new node is duplicated and avoids recursion issues or extra copies of nodes. As before, it returns the copied object when it is done.

There are several cycles in the graph shown in Figure 2.1, but handling the recursion with the memo dictionary prevents the traversal from causing a stack overflow error. When the root node is copied, the output is as follows.

$ python copy_recursion.py

Calling __deepcopy__ for Graph(name=root, id=4309347072)
   Memo dictionary:
   {}
Figure 2.1. Deepcopy for an object graph with cycles

Copying to new object Graph(name=root, id=4309347360)

Calling __deepcopy__ for Graph(name=a, id=4309347144)

Memo dictionary:
{
    Graph(name=root, id=4309347072): Graph(name=root, id=4309347360),
    4307936896: ['root'],
    4309253504: 'root'
}

Copying to new object Graph(name=a, id=4309347504)

Copying to new object Graph(name=root, id=4309347072)

Already copied to Graph(name=root, id=4309347360)

Copying to new object Graph(name=b, id=4309347144)

Calling __deepcopy__ for Graph(name=b, id=4309347216)

Memo dictionary:
{
    Graph(name=root, id=4309347072): Graph(name=root, id=4309347360),
    Graph(name=a, id=4309347144): Graph(name=a, id=4309347504),
    4307936896: ['root',
                'a',
                Graph(name=root, id=4309347072),
                Graph(name=a, id=4309347144)],
    4308678136: 'a',
    4309253504: 'root',
    4309347072: Graph(name=root, id=4309347360),
    4309347144: Graph(name=a, id=4309347504)
}

The second time the root node is encountered, while the a node is being copied, __deepcopy__() detects the recursion and reuses the existing value from the memo dictionary instead of creating a new object.
See Also:

* copy ([http://docs.python.org/library/copy.html](http://docs.python.org/library/copy.html)) The standard library documentation for this module.

## 2.9 pprint—Pretty-Print Data Structures

**Purpose** Pretty-print data structures.

**Python Version** 1.4 and later

`pprint` contains a “pretty printer” for producing aesthetically pleasing views of data structures. The formatter produces representations of data structures that can be parsed correctly by the interpreter and are also easy for a human to read. The output is kept on a single line, if possible, and indented when split across multiple lines.

The examples in this section all depend on `pprint_data.py`, which contains the following.

```python
data = [ (1, { 'a':'A', 'b':'B', 'c':'C', 'd':'D' }),
        (2, { 'e':'E', 'f':'F', 'g':'G', 'h':'H',
              'i':'I', 'j':'J', 'k':'K', 'l':'L',
              })],
```

### 2.9.1 Printing

The simplest way to use the module is through the `pprint()` function.

```python
from pprint import pprint
from pprint_data import data

print 'PRINT:'
print data
print
print 'PPRINT:'
pprint(data)
```

`pprint()` formats an object and writes it to the data stream passed as argument (or `sys.stdout` by default).

```
$ python pprint_pprint.py
```
PRINT:
[(1, {'a': 'A', 'c': 'C', 'b': 'B', 'd': 'D'}), (2, {'e': 'E', 'g': 'G', 'f': 'F', 'i': 'I', 'h': 'H', 'k': 'K', 'j': 'J', 'l': 'L'})]

PPRINT:
[(1, {'a': 'A', 'b': 'B', 'c': 'C', 'd': 'D'}),
 (2, {'e': 'E',
      'f': 'F',
      'g': 'G',
      'h': 'H',
      'i': 'I',
      'j': 'J',
      'k': 'K',
      'l': 'L'})]

2.9.2 Formatting

To format a data structure without writing it directly to a stream (i.e., for logging), use `pformat()` to build a string representation.

```python
import logging
from pprint import pformat
from pprint_data import data

logging.basicConfig(level=logging.DEBUG,
                    format='%(levelname)-8s %(message)s',
                    )

logging.debug('Logging pformatted data')
formatted = pformat(data)
for line in formatted.splitlines():
    logging.debug(line.rstrip())
```

The formatted string can then be printed or logged independently.

$ python pprint_pformat.py

DEBUG Logging pformatted data
DEBUG [(1, {'a': 'A', 'b': 'B', 'c': 'C', 'd': 'D'}),
       (2, {'e': 'E', 'f': 'F',
           'g': 'G', 'h': 'H', 'i': 'I', 'j': 'J', 'k': 'K', 'l': 'L'})]
2.9. pprint—Pretty-Print Data Structures

DEBUG 'g': 'G',
DEBUG 'h': 'H',
DEBUG 'i': 'I',
DEBUG 'j': 'J',
DEBUG 'k': 'K',
DEBUG 'l': 'L})"

2.9.3 Arbitrary Classes

The PrettyPrinter class used by pprint() can also work with custom classes, if they define a __repr__() method.

```python
from pprint import pprint

class node(object):
    def __init__(self, name, contents=[]):
        self.name = name
        self.contents = contents[:]
    def __repr__(self):
        return ('node(' + repr(self.name) + ' ', ' +
                repr(self.contents) + ')"

trees = [ node('node-1'),
          node('node-2', [ node('node-2-1')]),
          node('node-3', [ node('node-3-1')])]
pprint(trees)
```

The representations of the nested objects are combined by the PrettyPrinter to return the full string representation.

```
$ python pprint_arbitrary_object.py

[node('node-1', []),
 node('node-2', [ node('node-2-1', [])]),
 node('node-3', [ node('node-3-1', [])])]
```

2.9.4 Recursion

Recursive data structures are represented with a reference to the original source of the data, with the form <Recursion on typename with id=number>.
from pprint import pprint

local_data = ['a', 'b', 1, 2]
local_data.append(local_data)

print('id(local_data) =>', id(local_data))
pprint(local_data)

In this example, the list local_data is added to itself, creating a recursive reference.

$ python pprint_recursion.py
id(local_data) => 4309215280
['a', 'b', 1, 2, <Recursion on list with id=4309215280>]

2.9.5 Limiting Nested Output
For very deep data structures, it may not be desirable for the output to include all details. The data may not format properly, the formatted text might be too large to manage, or some of the data may be extraneous.

from pprint import pprint
from pprint_data import data

pprint(data, depth=1)

Use the depth argument to control how far down into the nested data structure the pretty printer recurses. Levels not included in the output are represented by an ellipsis.

$ python pprint_depth.py
[(...), (...)]

2.9.6 Controlling Output Width
The default output width for the formatted text is 80 columns. To adjust that width, use the width argument to pprint().

from pprint import pprint
from pprint_data import data

for width in [80, 5]:
    print('WIDTH =', width)
    pprint(data, width=width)
    print

When the width is too low to accommodate the formatted data structure, the lines are not truncated or wrapped if that would introduce invalid syntax.

$ python pprint_width.py

WIDTH = 80
[(1, {'a': 'A', 'b': 'B', 'c': 'C', 'd': 'D'}),
 (2,
  {'e': 'E',
   'f': 'F',
   'g': 'G',
   'h': 'H',
   'i': 'I',
   'j': 'J',
   'k': 'K',
   'l': 'L'})]

WIDTH = 5
[(1,
  {'a': 'A',
   'b': 'B',
   'c': 'C',
   'd': 'D'}),
 (2,
  {'e': 'E',
   'f': 'F',
   'g': 'G',
   'h': 'H',
   'i': 'I',
   'j': 'J',
   'k': 'K',
   'l': 'L'})]

See Also:
pprint (http://docs.python.org/lib/module-pprint.html) Standard library documentation for this module.
This page intentionally left blank
INDEX

SYMBOLS
?!-pattern, regular expressions, 47–48
(dot), character sets in pattern syntax, 23–24
: (colon), 360–362, 862
\ (backslash), escape codes for predefined character sets, 22
| (pipe symbol), 35, 413–418
= (equals sign), config files, 862
?: (question mark/colon), noncapturing groups, 36–37
! (exclamation point), shell commands, 848–849
$ (dollar sign),
    string.Template, 5–7
() (parentheses), dissecting matches with groups, 30–36
* (asterisk)
    bullet points, 13
    filename pattern matching in glob, 258–259
    repetition in pattern syntax, 17
?-pattern, regular expressions, 46–50
? (question mark)
    positional parameters with queries in sqlite3, 360
    repetition in pattern syntax, 17–20
    searching text with multiline input, 39
    shell commands in cmd, 848–849
    single character wildcard in glob, 259–260
[ ] (square brackets), config file sections, 862
^ (carat), 21, 39
{ } (curly braces),
    string.Template, 5–7
{m}, repetition in pattern syntax, 17–18
{n}, repetition in pattern syntax, 18

A
Abbreviations, option flags, 45
abc module
    abstract properties, 1182–1186
    concrete methods, 1181–1182
    defined, 1169
    how abstract base classes work, 1178
    implementation through subclassing, 1179–1181
    purpose of, 1178
    reasons to use abstract base classes, 1178
    reference guide, 1186
    registering concrete class, 1179
    ABCMeta class, 1178
    abc_register() function, 1179
    abspath() function, os.path, 254
Abstract base classes. See abc module
Abstract properties, abc, 1182–1186
abstractmethod(), abstract base classes, 1178
@abstractproperty, abc module, 1182–1186
accept(), socket, 572–573
Access
    network communications. See socket module
    network resources. See urllib, urllib2 module
Access control
    for concurrent use of resources in threading, 524–526
    Internet spiders, 674–677
    restricting for data in sqlite3, 384–386
    shared resources in multiprocessing, 546–550
    shared resources in threading, 517–523
access() function, os, 1127–1128
ACCESS_COPY argument, mmap, 280, 282–283
ACCESS_READ argument, mmap, 280
ACCESS_WRITE argument, mmap, 280–281
acquire() method, multiprocessing, 548
acquire() method, threading, 518–519, 522–524
Action class, 819–820
Actions
    argparse, 799–802, 819–820

1261
Aggregation functions, `sqlite3`, 380–381
Alarms, `signal`, 501–504
Alerts, nonfatal. See warnings module
Algorithms
  context manager utilities. See `contextlib` module
  functional interface to built-in operators. See `operator` module
  iterator functions. See `itertools` module
  manipulating functions. See `functools` module
overview of, 129
Aliased argument, `platform`, 1130–1131
Aliases, customizing `pdb` debugger, 1009–1011
all_done(), `atexit`, 890
Alternate API names,
  `SimpleXMLRPCServer`, 716–717
Alternate byte ordering, `array`, 86–87
Alternate representations, `math`, 227–229
Anchoring
  in pattern syntax, `re`, 24–26
  searching text using multiline input, 39
Angles, `math`, 238–240
Angular distribution, `random`, 223
annotate() function, `dircache`, 321–322
`anydbm` module
  creating new database, 348–349
  creating new shelf for data storage, 344
  database types, 347–348
  defined, 334, 346
  error cases, 349–350
  opening existing database, 348–349
  purpose of, 347
  reference guide, 350
APIs
  context manager, 164–167
  establishing with alternate names, 716–717
  establishing with arbitrary names, 719
  establishing with dotted names, 718–719
  Introspection, 724–725
  testing compliance with, 162–163
append action
  `argparse`, 799–802
  `optparse`, 786
append() method, `IMAP4` messages, 753–755
append_const action, `argparse`, 799–802
Appending to archives
  `tarfile`, 455
  `zipfile`, 464–465
Application building blocks command-line filters. See `fileinput` module
command-line option and argument parsing. See `argparse` module
command-line option parsers. See `getopt` module; `optparse` module
classification files. See `ConfigParser` module
GNU readline library. See `readline` module
line-oriented command processors. See `cmd` module
overview of, 769–770
parsing shell-style syntaxes. See `shlex` module
program shutdown callbacks with `atexit`, 890–894
reporting with logging module, 878–883
secure password prompt with `getpass`, 836–839
timed event scheduler with `sched`, 890–894
Applications
  localization with `gettext`, 907–908
  `optparse` help settings, 793–795
Approximation distribution, `random`, 222
Arbitrary API names,
  `SimpleXMLRPCServer`, 719
architecture() function, platform, 1133–1134
Archives, email
  listing mailbox subfolders, IMAP4, 743
  manipulating, See mailbox module
Archiving, data
  overview of, 421
  tarfile. See tarfile module
  zipfile. See zipfile module
argparse module
  argument actions, 799–802
  argument groups, 810–812
  automatically generated options, 805–807
  comparing with optparse, 796, 798
  conflicting options, 808–810
  defined, 769
  defining arguments, 796
  mutually exclusive options, 812–813
  nesting parsers, 813–814
  option prefixes, 802–803
  parsing command line, 796–797
  purpose of, 795
  reference guide, 822–823
  setting up parser, 796
  sharing parser rules, 807–808
  simple examples, 797–799
  sources of arguments, 804–805
  variable argument lists, 815–817
argparse module, advanced argument processing
  argument types, 817–819
  defining custom actions, 820–822
  file arguments, 819–820
  variable argument lists, 815–817
ArgumentParser class, argparse
  argument types, 817–819
  defined, 796
  option prefixes, 803
  simple examples, 797
Arguments
  command, 840–842
  command-line option parsing, See argparse module
  configuring callbacks for multiple. See optparse module
  fetching messages in IMAP4, 749–752
  getopt() function, 771
  method and function, 1209–1210
  network resource access with urllib, 653–655
  network resource access with urllib2, 660–661
  passing object to threads as, 506
  passing to custom thread type, 514
  passing to multiprocessing Process, 530
  platform() function, 1130–1131
  select() function, 595–596
  server address lookups with getaddrinfo(), 569–570
Arithmetic
  Counter support for, 73–74
  Decimal class, 199–200
  operators, 155–157, 183–184
  using fractions in, 210
ArithmeticError class, 1217
array module
  alternate byte ordering, 86–87
  defined, 69
  and files, 85–86
  initialization, 84–85
  manipulating arrays, 85
  purpose of, 84
  reference guide, 87
Arrays, plural values with gettext, 905–907
ASCII characters
  enabling Unicode matching, 39–40
  encoding and decoding data in strings, 335–336
  encoding binary data. See base64 module
assert*() methods, unittest, 952
assertFalse() method, unittest, 953
asserting truth, unittest, 952–953
AssertionError exception, 1217–1218
assertTrue() method, unittest, 953
assertion. See * (asterisk)
async_chat class, 629–630
asyncchat module
  client, 632–634
  message terminators, 629–630
  purpose of, 629
  putting it all together, 634–636
  reference guide, 636
  server and handler, 630–632
Asynchronous I/O, networking.
  See asyncore module
Asynchronous protocol handler.
  See asyncchat module
Asynchronous system events.
  See signal module
asyncore module
  asyncchat vs., 630–632
  clients, 621–623
  event loop, 623–625
  purpose of, 619
  reference guide, 629
  servers, 619–621
  SMTPServer using, 735
  working with files, 628–629
  working with other event loops, 625–627
atexit module
  defined, 770
  examples, 890–891
  handling exceptions, 893–894
  purpose of, 890
  reference guide, 894
  when callbacks are not invoked, 891–893
atof() function, locale, 917
atoi() function, locale, 917
attrib property, nodes, 392
Attribute getters, operator, 159–160
AttributeError exception, 1218–1219
Attributes
  configuring cmd through, 847–848
  parsed node, ElementTree, 391–393
Authentication
  argparse group for, 811
  failure, IMAP server, 740–741
  SMTP, 730–732
Authorizer function, sqlite3, 384
Auto-completion, cmd, 843–844
Auto-commit mode, sqlite3, 375–376
Automated framework testing. See unittest module
Automatically generated options, argparse, 805–807
avg() function, sqlite3, 380–381
B
B64decode(), 671–672
Babyl format, mailbox, 768
Back-references, re, 50–56
backslash (\), predefined character sets, 22
backslashreplace mode, codec error handling, 292, 294
Backup files, fileinput, 889
Base classes, exceptions, 1216
Base16 encoding, 673–674
Base32 encoding, 673
Base64 decoding, 671–672
Base64 encoding, 670–671
base64 module
Base64 decoding, 671–672
Base64 encoding, 670–671
defined, 637
other encodings, 673–674
purpose of, 670
reference guide, 674
URL-safe variations, 672–673
BaseException class, 1216
BaseHTTPServer module defined, 637
handling errors, 649–650
HTTP GET, 644–646
HTTP POST, 646–647
purpose of, 644
reference guide, 651
setting headers, 650–651
threading and forking, 648–649
basename() function, path parsing, 249–250
BaseServer class, SocketServer, 609–610
basicConfig() function, logging, 879
betavariate() function, random, 223
Bidirectional communication with process, 487–489
Binary data
preparing for transmission, 591–593
structures, 102–106
XML-RPC server, 710–712
Binary digests, hmac, 475–476
Binary heaps, heapq, 88
Binary read mode, gzip, 433–434
bisect() method, heapq, 89–90
bisect module
defined, 69
handling duplicates, 95–96
inserting in sorted order, 93–95
purpose of, 93
reference guide, 96
Blank lines
with doctest, 930–932
with linecache, 263
Bodies of text, comparing, 62–65
BOM (byte-order marker), codecs, 289–291
Boolean
argparse options, 797
logical operations with operator, 154
optparse options, 785–786
break command, breakpoints in pdb, 990, 992–993, 998
break lineno, pdb, 990–991
Breakpoints, pdb
conditional, 998–999
ignoring, 999–1001
managing, 993–996
restarting program without losing current, 1008–1009
setting, 990–993
temporary, 997–998
triggering actions on, 1001–1002
Browser, class, 1039–1043
BufferAwareCompleter class, readline, 828–831
BufferedIncrementalDecoder, codecs, 313
BufferedIncrementalEncoder, codecs, 312
Buffers, struct, 105–106
Build-time version information, settings in sys, 1055–1057
Building paths, os.path, 252–253
Building threaded podcast client, Queue, 99–101
Building trees, ElementTree, 405–408
Built-in exception classes. See exceptions module
Built-in modules, sys, 1080–1091
Built-in operators. See operator module
__builtins__ namespace, application localization with gettext, 908–909
__builtins__ namespace, gettext, 908–909
Bulk loading, sqlite3, 362–363
Byte-compiling source files, compileall, 1037–1039
byte-order marker (BOM), codecs, 289–291
Byte ordering
alternate arrays, 86–87
encoding strings in codecs, 289–291
memory management with sys, 1070–1071
specifiers for struct, 103
Bytecodes
counting with dis, 1078
finding for your version of interpreter, 1186
modifying check intervals with sys, 1074–1078
Python disassembler for. See dis module
byteswap() method, array, 87
bz2 module
compressing networked data, 443–448
incremental compression and decompression, 438–439
mixed content streams, 439–440
one-shot operations in memory, 436–438
purpose of, 436
reading compressed files, 442–443
reference guide, 448
writing compressed files, 440–442
BZ2Compressor, 438–439, 444–445
### Compressing Network Data

- bz2: 445–447
- Incremental Decompression: 438–439
- Mixed Content Streams: 424–425

### BZ2File

- BZ2RequestHandler: 443–447

### Bzip2 Compression

See bz2 module

### Calculations

- Math: 230–233

### Calendar Class

- Calculating Dates: 194–195
- Defined: 173
- Formatting Examples: 191–194
- Purpose of: 191
- Reference Guide: 196

### Call Events

- Sys: 1102–1103

### Callbacks

- For Options with optparse: 788–790
- Program Shutdown with atexit: 890–894
- Reference: 108

### CalledProcessError

- Exception: subprocess, 483–484

### Callee Graphs

- Pstats: 1029–1031

### Caller Graphs

- Pstats: 1029–1031

### Canceling Events

- Sched: 897–898

### Can Fetch

- Internet Spider: access control, 675–676

### Canonical Name Value

- Server Addresses: 570

### Capwords

- String: 4–5

### Carat (^)

- 21, 39

### Case-Insensitive Matching

- Embedding Flags in Patterns: 44–45
- Searching Text: 37–38

### Case-Sensitive Matching

- Glob Pattern Matching: 315–317

### CGI

- Module: HTTP POST Requests: 646–647

### CGIHTTPBunker

- Defined: 919
- Enabling Detailed Tracebacks: 966–968
- Exception Properties: 971–972
- HTML Output: 972
- Local Variables in Tracebacks: 968–971
- Logging Tracebacks: 972–975
- Purpose of: 965–966
- Reference Guide: 975
- Standard Traceback Dumps: 966

### Chain

- Function: itertools, 142–143

### Character Maps

- Codecs: 307–309

### Character Sets

- Pattern Syntax: 20–24
- Using Escape Codes for Predefined: 22–24

### Characters

- Glob Module: 258–260
- Charmap Decode: codecs, 308
- Charmap Encode: codecs, 308
- Chdir Function: os, 1112

### Check Intervals

- Sys: 1074–1078

### Check Call

- Function: subprocess, 483–484

### Check Output

- Function: subprocess, 484–486

### Checksums

- Computing in zlib: 425

### Child Processes

- Managing I/O of: 1112–1116
- Waiting For: 1125–1127

### Chmod

- Function: File Permissions in UNIX: 1117–1118

### Choice

- Function: Random: 215–216

### Choice Type

- Optparse: 784

### Circular Normal Distribution

- Random: 223

### Circular References

- Pickle: 340–343

### Class Browser

- Pyclbr: 1039–1043

### Class Hierarchies

- Method Resolution Order: 1212–1213
- Working With: 1210–1212

### Classes

- Abstract Base: abc Module
- Built-in Exception: exceptions module
- Disassembling Methods: 1189–1190
- Inspecting With: inspect, 1104–1106
- Scanning With: pycparser, 1041–1042

### CleanUpGraph Class

- 1153–1159

### Clear Command

- Breakpoints in: pdb, 996

### Clear Method

- Signaling between Threads: 516

### Client

- Implementing with: asynchat, 632–634
- Implementing with: asyncore, 621–623
- Library for: xmlrpclib module

### Clock Time

- See time module

### Close

- Function: Creating Custom Tree Builder: 398
- Deleting Email Messages: 758
- Echo Server in: TCP/IP Sockets: 573
- Process Pools in: multiprocessing, 554
- Removing Temporary Files: 266

### Closing

- Function: Open Handles in: contextlib, 169–170

### Cmd Class

- 839–840

### Cmd Module

- Alternative Inputs: 849–851
- Auto-Completion: 843–844
- Command Arguments: 840–842
- Commands from: sys.argv, 851–852
cmd module (continued)
configuring through attributes, 847–848
defined, 769
live help, 842–843
overriding base class methods, 845–846
processing commands, 839–840
purpose of, 839
reference guide, 852
running shell commands, 848–849
cmdloop(), overriding base class methods, 846

cmp() function, filecmp, 325–326
cmpfiles() function, 326–327
cmp_to_key() function, collation order, 140–141
Code coverage report, trace, 1013–1017
CodecInfo object, 309–310

codecs module
byte order, 289–291
defined, 248
defining custom encoding, 307–313
encoding translation, 298–300
encodings, 285–287
error handling, 291–295
incremental encoding, 301–303
non-Unicode encodings, 300–301
opening Unicode configuration files, 863–864
purpose of, 284
reference guide, 313–314
standard input and output streams, 295–298
Unicode data and network communication, 303–307
Unicode primer, 284–285
working with files, 287–289
Collations
customizing in sqlite3, 381–383
functools comparison functions, 140–141
collect() function, forcing garbage collection, 1141–1146

collections module
Counter, 70–74
defaultdict, 74–75
defined, 69–70
deque, 75–79
namedtuple, 79–82
OrderedDict, 82–84
reference guide, 84
colon (:), 360–362, 862
Columns, sqlite3
defining new, 363–366
determining types for, 366–368
restricting access to data, 384–386
combine() function, datetime, 188–189

Comma-separated value files. See csv module
Command handler, cmd, 839–840
Command-line filter framework. See fileinput module
interface, with timeit, 1035–1036
interpreter options, with sys, 1057–1058
invoking compileall from, 1039
processors. See cmd module
runtime arguments with sys, 1062–1063
starting pdb debugger from, 976
using trace directly from, 1012–1013
Command-line option parsing and arguments. See argparse module
Command-line option parsing getopt. See getopt module
optparse. See optparse module
Commands
interacting with another, 490–492
running external, with os, 1121–1122
running external, with subprocess, 482–486
triggering actions on breakpoints, 1001–1002
comment() function, hierarchy of Element nodes, 400–401
commenters property, shlex, 854

Comments
embedded, with shlex, 854
inserting into regular expressions, 43–44
commit(), database changes, 368–370
commonprefix() function, path parsing, 251
communicate() method
interacting with another command, 490–492
working with pipes, 486–489
Communication
accessing network. See socket module
configuring nonblocking socket, 593–594
using pickle for inter-process, 334, 338
Compact output, JSON, 692–694
cmp() function, text, 62–64
Comparison
creating UUID objects to handle, 689–690
files and directories. See filecmp module
UNIX-style filenames, 315–317
values in datetime, 187–188
Comparison, functools
collation order, 140–141
overview of, 138
reference guide, 141
rich comparison, 138–140
Comparison operators
date and time values, 185
with operator, 154–155
compile() function, expressions, 14–15
compileall module, 920,
1037–1039
compile_dir(), compileall, 1037–1038
compile_path(), compileall, 1038–1039
Compiler optimizations, dis, 1198–1199
complete() accessing completion buffer, 830
text with readline, 826–827
complete_prefix, command auto-completion, 843–844
Complex numbers, 235
compress() method, bz2
compressing network data, 443
incremental compression, 439
one-shot operations in memory, 436–438

compress() method, zlib
compressing network data, 426–427
incremental compression and decompression, 424
Compress object, zlib, 423–424
Compression, data
archives in tarfile, 456
bzip2 format. See bz2 module
GNU zip library. See zlib module
gzip module, 430–436
overview of, 421
ZIP archives. See zipfile module
Compresslevel argument
writing compressed files in BZ2File, 440–442
writing compressed files in gzip, 431
compress_type argument, zipfile, 463
Concrete classes, abc
abstract properties, 1182–1186
how abstract base classes work, 1178
methods in abstract base classes, 1181–1182
registering, 1179
Concurrent operations. See threading module
condition command, pdb, 998–999
Condition object
synchronizing processes, 547–548
synchronizing threads, 523–524
Conditional breakpoints, 998–999
ConfigParser module
accessing configuration settings, 864–869
combining values with interpolation, 875–878
configuration file format, 862
defined, 770
modifying settings, 869–871
option search path, 872–875
purpose of, 861–862
reading configuration files, 862–864
reference guide, 878
saving configuration files, 871–872
Configuration files
configuring readline library, 823–824
saving in pdb debugger, 1011–1012
working with. See ConfigParser module
Configuration variables,
sysconfig, 1160–1161
conflict_handler, argpase, 807–808
connect() function
creating embedded relational database, 352
sending email message with smtplib, 728
socket setup for TCP/IP echo client, 573–574
Connections
easy TCP/IP client, 575–577
to IMAP server, 739–740
monitoring multiple, with select() function, 596–597
segments of pipe with subprocess, 489–490
to server with xmlrpclib, 704–706
sharing with sqlite3, 383–384
constant property, abc, 1183
Constants
option actions in optparse, 785
text, 4–9
Consuming, deque, 77–78
Container data types
Counter, 70–74
defaultdict, 74–75
deque, 75–79
namedtuple, 79–82
OrderedDict, 82–84
Context manager
locks, 522–523
utilities. See contextlib module
Context, running profiler in, 1026
diff() function, difflib output, 65
contextlib module
closing open handles, 169–170
text, 871–872
context manager API, 164–167
defined, 129
from generator to context manager, 167–168
nesting contexts, 168–169
purpose of, 163
reference guide, 170–171
c Kontextmanager() decorator, 167–168
Contexts
decimal module, 201–205
nesting, 168–169
reference guide, 207
continue command, pdb breakpoints, 991
Controlling parser, shlex, 856–858
Conversion
argument types in argparse, 817–819
optparse option values, 783
Converter, 364
Cookie module
alternative output formats, 682–683
creating and setting cookies, 678
defined, 637
deprecated classes, 683
encoded values, 680–681
morsels, 678–680
purpose of, 677–678
receiving and parsing cookie headers, 681–682
reference guide, 683

copy() function
creating shallow copies with copy, 118
files, with shutil, 273
IMAP4 messages, 755–756

-copy__() method, 118–119, 819–820

copy module
customizing copy behavior, 119–120
deep copies, 118–119
defined, 70
purpose of, 117–118
recursion in deep copy, 120–123
reference guide, 123
shallow copies, 118

copy2() function, shutil, 273–274

copyfile() function, shutil, 271–272
copyfileobj() function, shutil, 272
Copying directories, 276–277 duplicating objects using copy. See copy module files, 271–275
copymode() function, shutil, 274–276
copysign() function, math, 229–230
copystat() function, shutil, 275–276
copytree() function, shutil, 276–277
Cosine, math hyperbolic functions, 243–244 trigonometric functions, 240–243
count action, optparse, 787–788
count() function customizing aggregation in sqlite3, 380–381 new iterator values with itertools, 146–147
Counter container accessing counts, 71–73 container data type, 70 initializing, 70–71 supporting arithmetic, 73–74 Counts, accessing with Counter, 71–73
count_words(), MapReduce, 558
Coverage report information, trace, 1013–1017 CoverageResults, Trace object, 1020–1021
cPickle, importing, 335 cProfile module, 1022
CPUs, setting process limits, 1137
crc32() function, checksums in zlib, 425
create(), messages in IMAP4, 756
create_aggregate(), sqlite3, 381 create_connection(), TCP/IP clients, 575–577
createfunction() method, sqlite3, 379–380
CRITICAL level, logging, 881
Cryptography creating UUID name-based values, 686–688 generating hashes and message digests. See hashlib module message signing and verification. See hmac module cStringIO buffers, 314–315 CSV (comma-separated value) files. See csv module
Current date, 182 Current process, multiprocessing, 531–532 Current thread, threading, 507–508 Current usage, resource, 1134–1135 Current working directory, os, 1112
package importing, with sys, 1091–1093 site configuration, with site, 1051–1052 sorting, with sqlite3, 381–383 user configuration, with site, 1053–1054
cycle() function, itertools, 147 Cyclic references, weakref, 109–114
D Daemon processes, multiprocessing, 532–534 Daemon threads, threading, 509–511, 512–513
Data archiving overview of, 421 tar archive access. See tarfile module
ZIP archive access. See zipfile module
Data argument, SMTPServer class, 734
Data communication, Unicode, 303–307
Data compression bzip2 compression. See bz2 module GNU zlib compression. See zlib module overview of, 421 read and write GNU zip files. See gzip module
ZIP archives. See zipfile module
Data(), creating custom XML tree builder, 398
Data decompression archives in tarfile, 456 bzip2 format. See bz2 module GNU zip library. See zlib module gzip module, 430–436 overview of, 421
ZIP archives, See zipfile module
data definition language (DDL) statements, 353–355
Data extremes, from heap, 92–93
Data files
retrieving for packages with `pkgutil`, 1255–1258
retrieving with `zipimport`, 1244–1246

Data persistence and exchange
- `anydbm` module, 347–350
- comma-separated value files. See `csv` module
- embedded relational database. See `sqlite3` module
- object serialization. See `pickle` module
overview of, 333–334
- `shelve` module, 343–346
- `whichdb` module, 350–351
XML manipulation API. See `ElementTree`

Data structures
- `array` module, 84–87
- `bisect` module, 93–96
- `collections` module. See `collections` module
- `copy` module, 117–123
- `heapq` module, 87–93
overview of, 69–70
- `print` module, 123–127
- `Queue` module, 96–102
- `struct` module, 102–106
- `weakref` module. See `weakref` module

Data types
- encoding and decoding in `JSON`, 690
- XML-RPC server, 706–709

Database types, `anydbm`, 347–348

Databases
- identifying DBM-style formats, 350–351
- implementing embedded relational. See `sqlite3` module
- providing interface for DBM-style. See `anydbm` module

Data_encoding value, translation, 299

Date arithmetic, `datetime`, 186–187
Date class, `calendar`, 182–185
Date columns, `sqlite3` converters for, 364
Date values
- comparing time and, 184–185
- `datetime` module, 182–185

Dates and times
- `calendar` module dates, 191–196
clock time. See `time` module
- `locale` module, 917–918
- manipulating values. See `datetime` module
overview of, 173
- `datetime` class, 188–189
datetime module
combining dates and times, 188–189
- comparing values, 187–188
- converters for date/timestamp columns in `sqlite3`, 364
date arithmetic, 186–187
dates, 182–185
defined, 173
- formatting and parsing, 189–190
- purpose of, 180
- reference guide, 190–191
time zones, 190
timedelta, 185–186
times, 181–182
day attribute, date class, 182–183
- `DBfilenameShelf` class, 343–344
dbhash module, 347, 348–349
dbm module
- accessing DBM-style databases, 347–348
- creating new database, 348–349
- creating new shelf, 344
- DBM-style databases. See also `anydbm` module, 350–351
- `DDL` (data definition language) statements, 353–355
DEBUG level, `logging`, 881–882
- `DEBUG_COLLECTABLE` flag, gc, 1152, 1154
DEBUG_MEMORY flag, `gc`, 1159
- `DEBUG_SAVEALL` flag, gc, 1156, 1159
DEBUG_STATS flag, gc, 1152
- `DEBUG_UNCOLLECTABLE` flag, gc, 1152, 1154
decimal module
- arithmetic, 199–200
- contexts, 201–207
- `Decimal` class, 198–199
defined, 197
- fractions, 207–211
- `math` module, 223–245
- purpose of, 197
- random module, 211–223
- special values, 200–201
decode() method, custom encoding, 312–313
decoded() method, encodings, 286
Decoding
Base64, 671–672
data in strings with `pickle`, 335–336
error handling with `codecs`, 294–295
files with `codecs`, 287–289
- `JSON`, 690, 697–700
Decoding maps, 307–309
decompress() method
- compressing network data in `bz2`, 443
- compressing network data in `zlib`, 426–427
Decompress object, `zlib`, 423–425
Decompression, data
- archives in `tarfile`, 456
- bzip2 format. See `bz2` module
Decompression, data (continued)

GNU zip library. See zlib module
gzip module, 430–436
overview of, 421
ZIP archives. See zipfile module

Decompression, zlib
compressing network data, 426–430
incremental, 423–424
in mixed content streams, 424–425
working with data in memory, 422–423
decompressobj(), zlib, 424–425

Decorators, functools
acquiring function properties, 132–133, 136–138
other callables, 133–136
partial objects, 130–132
reference guide, 141
dedented_text, textwrap, 11–13

Deep copies, copy
creating, 118–119
customizing copy behavior, 119
recursion, 120–123
__deepcopy__() method, copy, 118–123
deepcopy() method, 118–119
default() method, cmd, 840, 846
DEFAULT section,
ConfigParser, 872, 876
Defaultdict, container data type, 74–75

DEPRECATED isolation level,
sqlite3, 373–374

Degrees
converting from radians to, 239–240
converting to radians from, 238–239

Delay function, Scheduler, 894–896

Deleting
e-mail messages, 756–758
messages from Maildir mailbox, 764–765
messages from mbox mailbox, 761–762
delitem() function, sequence operators, 158
denominator values, creating fraction instances, 207–208
DeprecationWarning, 182, 1233
dep
consuming, 77–78
container data type, 75–76
populating, 76–77
rotation, 78–79
detect_types flag, sqlite3, 363–366

Developer tools
byte-compiling source files, 1037–1039
creating class browser, 1039–1043
detailed traceback reports. See pdb module
exceptions and stack traces. See traceback module
interactive debugger. See pdb module
online help for modules, 920–921
overview of, 919–920
performance analysis with profile, 1022–1026
performance analysis with pstats, 1027–1031
testing with automated framework. See unittest module
testing with documentation. See doctest module
timing execution of bits of code. See timeit module
tracing program flow. See trace module

Dialect parameters, csv, 415–417
Dialects, csv
automatically detecting, 417–418
dialect parameters, 415–417
overview of, 413–414

Dictionaries
JSON format for encoding, 694
storing values using timeit, 1033–1035

DirReader class, csv, 418–420
DirWriter class, csv, 418–420

Diff-based reporting options,
doctest, 933–935

Differ class, 62, 65
difflib module
comparing arbitrary types, 66–68
comparing bodies of text, 62–65
comparing sequences, 61–62
junk data, 65–66
reference guide, 68
digest() method
binary digests in hmac, 475–476
calculating MD5 hash in hashlib, 470
dircache module
annotated listings, 321–322
defined, 247
listing directory contents, 319–321
purpose of, 319
reference guide, 322
dircmp class, filecmp, 326, 328–332

Directories

cache listings, 319–322
comparing, 327–332
compiling one only, 1037–1038
creating temporary, 268–269
functions in os, 1118–1119
installing message catalogs in, 902
site module user, 1047–1048

Directory trees
copying directories, 276–277
moving directory, 278
removing directory and its contents, 277–278
traversing in os, 1120–1121
traversing in os.path, 256–257
dirname() function, path parsing, 250
dis() function, 1187
dis module
basic disassembly, 1187
classes, 1189–1190
compiler optimizations, 1198–1199

counting bytecodes with, 1078
defined, 1169
disassembling functions, 1187–1189
performance analysis of loops, 1192–1198
Index

purpose of, 1186
reference guide, 1199–1200
using disassembly to debug, 1190–1192
disable command, breakpoints in pdb, 993–994
Disabling, site, 1054
__dispatch() method, MyService, 723
Dispatcher class, asyncore, 619–621
Dispatching, overriding in SimpleXMLRPCServer, 722–723
displayhook, sys, 1060–1062
Dissecting matches with groups, re, 30–36
distb() function, 1191
disutils, sysconfig extracted from, 1160
Division operators, 156–157
DNS name, creating UUID from, 687
DocFileSuite class, 945
doc_header attribute, cmd, 847–848
doctest module
defined, 919
external documentation, 939–942
getting started, 922–924
handling unpredictable output, 924–928
purpose of, 921–922
reference guide, 948–949
running tests, 942–945
test context, 945–948
test locations, 936–939
tracebacks, 928–930
using unittest vs., 922
working around whitespace, 930–935
DocTestSuite class, 945
Documentation
retrieving strings with inspect, 1206–1207
testing through. See doctest module
Documents, XML
building with Element nodes, 400–401
finding nodes in, 390–391
parsing, 387
watching events while parsing, 393–396
do_EOF(), cmd, 839–840
do_GET() method, HTTP GET, 644–646
dollar sign ($), string.Template, 5–7
Domain, installing message catalogs in directories, 902
Domain sockets, UNIX, 583–587
do_POST() method, HTTP POST, 646–647
do_shell(), cmd, 848–849
dot (), character sets in pattern syntax, 23–24
DOTALL regular expression flag, 39, 45
Dotted API names, SimpleXMLRPCServer, 718–719, 721
Double-ended queue (deque), collections, 75–79
double_space() function, doctest, 930
down (d) command, pdb, 980
downloadEnclosures() function, Queue class, 99–102
dropwhile() function, itertools, 148–149, 150
dump() function, json, 700–701
dumpdbm module, 348–349
dumps () function
encoding data structure with pickle, 335–336
JSON format, 692–694
Duplicating objects. See copy module

E
Echo client
implementing with asynchat, 632–636
implementing with asyncore, 621–625
TCP/IP, 573–574
UDP, 581–583
Echo server
implementing with asynchat, 630–632, 634–636
implementing with asyncore, 619–625
SocketServer example, 610–615
TCP/IP socket, 572–573
UDP, 581–583
EchoHandler class, 620–621, 630–632
EchoRequestHandler, SocketServer, 611–612
ehlo(), SMTP encryption, 730–732
element () function, ElementTree, 400–401
elements() method, Counter, 72
ElementTree
building documents with element nodes, 400–401
building trees from lists of nodes, 405–408
creating custom tree builder, 396–398
defined, 334
finding nodes in document, 390–391
parsed note attributes, 391–393
parsing strings, 398–400
parsing XML document, 387–388
pretty-printing XML, 401–403
purpose of, 387
reference guide, 410–411
serializing XML to stream, 408–410
setting element properties, 403–405
traversing element tree, 388–390
watching events while parsing, 393–396
ELLIPSIS option, unpredictable output in doctest, 925
Email
IMAP4 client library. See imaplib module
manipulating archives. See mailbox module
sample mail servers, smtpd module, 734–738
SMTP client. See smtplib module
Embedded comments, shlex, 854
Embedded flags in patterns, searching text, 44–45
Embedded relational database. See sqlite3 module
empdir() function, tempfile, 270–271
emptyline(), cmd, 846
enable command, breakpoints in pdb, 994–996
enable() function, cgitb, 969, 972–973
encode() method
custom encoding, 312–313
JSONEncoder class, 698
encodedFile() function, translations, 298–299
Encoding
binary data with ASCII. See base64 module
Cookie headers, 680–681
data in strings with pickle, 335–336
files for upload with urllib2, 664–667
JSON, classes for, 697–700
JSON, custom types, 695–697
JSON, dictionaries, 694
JSON, simple data types, 690
JSON, working with streams and files, 700–701
network resource access with urllib, 653–655
network resource access with urllib2, 660–661
Encoding, codecs
byte ordering, 289–291
defining custom, 307
derror handling, 291–294
incremental, 301–303
non-Unicode, 300–301
standard I/O streams, 295–298
translation, 298–300
understanding, 285–287
Unicode data and network communication, 303–307
working with files, 287–289
Encoding maps, 307–309
Encryption, SMTP class, 732–733
end events, watching while parsing, 393–396
end() method
creating custom tree builder, 398
finding patterns in text, 14
end-ns events, watching while parsing, 394–396
Endianness
byte ordering in codecs, 289–291
reference guide, 314
struct module, 103–105
__enter__() method,
contextlib, 164–165
enter() method, sched, 895, 897–898
enterabs() method, sched, 897–898
enumerate(), threads, 512–513
Enumerations, optparse, 784
Environment variables, os, 1111–1112
EnvironmentError class,
exceptions, 1217
EOFError exception, 1220
epoll() function, select, 608
Equality
OrderedDict, 83–84
testing with unittest, 953–955
equals sign (=), config files, 862
erf() function, math, 244–245
erfc() function, math, 245
Error cases, anydbm, 349–350
error conflict_handler,
argparse, 808–810
Error handling. See also Exception handling.
BaseHTTPRequestServer, 649–650
codecs, 291–295
imports, 1094–1095
linecache, 263–264
logging, 878–883
shlex, 858–859
subprocess, 483–486
tracebacks. See traceback module
ERROR level, logging, 881–882
Escape codes, 22–24, 39–40
Event loop, asyncio, 623–627
Events
asynchronous system. See signal module
flags for poll(), 604
hooks for settrace(), 1101
POLLERR, 607
signaling between processes, 545–546
signaling between threads, 516–517
watching while parsing, 393–396, 894–898
excel dialect, CSV, 414
excel-tabs dialect, CSV, 414
excepthook, sys, 1072
Exception class, 1216
Exception classes, built-in. See exceptions module
Exception handling. See also Error handling.
argparse, 808–810
atexit, 893–894
cgitb. See cgitb module
readline ignoring, 827
sys, 1071–1074
traceback, 959–962
tracing program as it runs, 1106–1107
type conversion in argparse, 818
XML-RPC server, 712
Exceptional sockets, select() function, 598
Exceptional values, math, 224–226
Exceptions
debugging using dis, 1190–1192
testing for, unittest, 955–956
exceptions module
base classes, 1216–1217
defined, 1169
purpose of, 1216
raised exceptions. See Raised exceptions
reference guide, 1233
warning categories, 1233
Exchange, data. See data persistence and exchange
exc_info(), sys, 1072–1073
exclamation point (!), shell commands, 848–849
EXCLUSIVE isolation level, sqlite3, 374–375
exec() function, os, 1124–1125, 1127
Executable architecture, platform, 1133–1134
execute() method, sqlite3, 355, 359–360
execute_many() method,
sqlite3, 362–363
Index 1273

executescript() method, sqlite3, 354

Execution
  changing flow in pdb, 1002–1009
  timing for small bits of code. See timeit module
  using trace directly from command line, 1012–1013

Execution stack, pdb, 979–984

Exit code, sys, 1064–1065
__exit__() method, contextlib, 164–167
exp() function, math, 237
expandcars() function, os.path, 253
expanduser() function, os.path, 252
expml() function, math, 237–238

Exponential distribution, random, 222

Exponents, math, 234–238

Exporting database contents, sqlite3, 376–378

Exposed methods,
  SimpleXMLRPCServer, 720–723

expovariate() function, random, 222

EXPUNGE command, emptying email trash, 757–758

Failure, debugging after, 978–979

Fault objects, XML-RPC exception handling, 711–714

feedcache module, 346
feedparser module, 100–101
fetch() method, IMAP4, 749–752
fetchall() method, sqlite3, 355–356
fetchmany() method, sqlite3, 356–357
fetchone() method, sqlite3, 356

Fibonacci sequence calculator, 1023–1026

File arguments, argparse, 819–820
__file__ attribute, data files, 1244–1246

File descriptors
  mmap, 279–280
  os, 1116

file-dispatcher class, asyncore, 628–629

File format, ConfigParser, 862

File system
  comparing files. See filecmp module
  dircache module, 319–322

filename manipulation. See os.path module
fnmatch module, 315–318
glob module, 257–260

high-level file operations. See shutil module
linecache module, 261–265
mmap module, 279–284
overview of, 247–248
permissions with os, 1116–1118, 1127–1128

string encoding and decoding. See codecs module
StringIO module, 314–315
temporary file system objects. See tempfile module
  working with directories, 1118–1119

file_wrapper class, 628–629

filecmp module
  comparing directories, 327–328
  comparing files, 325–327
  defined, 247–248
  example data, 323–325
  purpose of, 322–323
  reference guide, 332
  using differences in program, 328–332

fileinput module
  converting M3U files to RSS, 883–886
  defined, 770
  in-place filtering, 887–889
  progress metadata, 886–887
  purpose of, 883
  reference guide, 889
  filelineno() function, fileinput, 886–887

filemode argument, rotating log files, 879

filenamemodule
  fileinput, 886–887

Filenames
  alternate archive member names in tarfile, 453–454
  alternate archive member names in zipfile, 462–463
  pattern matching with glob, 257–260
  platform-independent manipulation of. See os.path module
Filenames (continued)
predicting in temporary files, 269–270
specifying breakpoints in another file, 991–992
UNIX-style comparisons, 315–317
fileno() method, mmap, 279–280
FileReader, asyncore, 628–629
Files. See also file system
arrays and, 85–86
comparing, 325–327
logging to, 879
reading asynchronously in asyncore, 628–629
running tests in doctest by, 944–945
working with codecs, 287–289
working with json, 700–701
file_to_words() function, MapReduce, 558
FileType, argparse, 819–820
fill() function, textwrap, 10–12
filter() function, UNIX-style filename comparisons, 317–318
Filters
directory, 1037
with itertools, 148–151
processing text files as. See fileinput module
warning, 1170–1174
filterwarnings() function, 1172–1174
finalize() method, sqlite3, 380
find() function, gettext, 903–904
findall() function
finding nodes in document, ElementTree, 390–391
multiple pattern matches in text, 15–16
splitting strings with patterns, 58–60
Finder phase, custom importer, 1083–1085
finditer() function, re, 15–17
find_module() method
with imp, 1237–1238
inside ZIP archive, 1241–1242
finish() method,
SocketServer, 610
finish_request() method,
SocketServer, 610
First-in, first-out (FIFO). See also Queue module, 96–97
Fixed numbers. See decimal module
Fixed-type numerical data, sequence, 84–87
Fixtures, unittest test, 956–957
Flags
options with ConfigParser, 868–869
variable argument definitions in argparse, 815–817
Flags, regular expression abbreviations for, 45
case-insensitive matching, 37–38
embedding in patterns, 44–45
multiline input, 38–39
Unicode, 39–40
verbose expression syntax, 40–44
Float class, fractions, 209
float_info, memory management in sys, 1069–1070
Floating point columns, SQL support for, 363–366
Floating-point numbers. See also decimal module
absolute value of, 229–230
alternate representations, 227–229
common calculations, 230–233
converting to rational value with fractions, 210–211
generating random integers, 214–215
Floating-point values commonly used math calculations, 230–233
converting to integers in math, 226–227
Floating-point values creating fraction instances from, 208–209
generating random numbers, 211–212
memory management with sys, 1069–1070
testing for exceptional, 224–226
time class, 182
FloatingPointError exception, 1220
floor() function, math, 226–227
floordiv() operator, 156
flush() method
incremental compression/decompression in zlib, 424
incremental decompression in bz2, 439
fmod() function, math, 232–233
fnmatch module
defined, 247
filtering, 317–318
purpose of, 315
reference guide, 318
simple matching, 315–317
translating patterns, 318
fnmatchcase() function, 316–317
Folders, Maildir mailbox, 766–768
forcing garbage collection, gc, 1141–1146
fork() function, os, 1122–1125, 1127
Forking, adding to HTTPServer, 648–649
ForkingMixIn, 617–618, 649
format() function, locale, 916–917
format_exception() function, traceback, 958, 961–962
formatmonth() method, calendar, 192
format_stack() function, traceback, 958, 964
Formatting calendars, 191–194
dates and times with datetime, 189–190
dates and times with locale, 917–918
DBM-style database with whichdb, 350–351
e-mail messages. See mailbox module
JSON, 692–694
numbers with locale, 916–917
printing with pprint, 123–127
stack trace in traceback, 958
time zones with time, 178
warnings, 1176
formatwarning() function, warning, 1176
formatyear() method, calendar, 192–194
fractions module
approximating values, 210–211
arithmetic, 210
creating fraction instances, 207–210
defined, 197
purpose of, 207
reference guide, 211
Frames, inspecting runtime environment, 1213–1216
frexp() function, math, 228–229
From headers, smtplib, 728
from_float() method, Decimal class, 198
fromordinal() function, datetime, 184, 189
fromtimestamp() function, datetime, 183–184, 189
fsum() function, math, 231
Functions
arguments for, 1209–1210
disassembling, 1187–1189
mathematical. See math module
scanning using pycalbr, 1042–1043
setting breakpoints, 991
string, 4–5
Struct class vs., 102
tools for manipulating. See functools module
traceback module, 958–959
using Python in SQL, 378–380
functools module
acquiring function properties, 132–133
acquiring function properties for decorators, 136–138
comparison, 138–141
decorators. See decorators, functools
defined, 129
other callables, 133–136
partial objects, 130–132
purpose of, 129
reference guide, 141
FutureWarning, 1233
G
gamma() function, math, 232
gammavariate() function, random, 223
Garbage collector. See also gc module, 1065–1066
Gauss Error function, statistics, 244–245
gauss() function, random, 222
gc module, 1138–1160
collection thresholds and generations, 1148–1151
debugging memory leaks, 1151–1159
defined, 1138–1160
forcing garbage collection, 1141–1146
purpose of, 1138
reference guide, 1159–1160
references to objects that cannot be collected, 1146–1148
tracing references, 1138–1141
gdbm module, 347–349
Generations, gc collection, 1148–1151
Generator function, contextlib, 167–168
GeneratorExit exception, 1221
get() method
basic FIFO queue, 97
ConfigParser, 865–867, 875–878
LifoQueue, 97
PriorityQueue, 98–99
GET requests
BaseHTTPServer, 644–646
client, 657–660
getaddrinfo() function, socket, 568–570, 576
getargspec() function, inspect, 1209–1210
getargvalues() function, inspect, 1213
getatime() function, os.path, 254
getboolean() method, ConfigParser, 867–868
getfqdn() function, socket, 565
get_history_item(), readline, 832–834
gethostbyaddr() function, socket, 565
gethostname() function, socket, 563–564
gethostbyname() function, socket, 563, 577–580
getinfo() method, zipfile, 458–459
getint() method, ConfigParser, 867
getline() function, linecache, 263–264
get_logger(), multiprocessing, 539–540
getmember(), tarfile, 449–450
getmembers() function, inspect, 1201–1203, 1204–1206
getmembers(), tarfile, 449–450
get_code() method, zipimport, 1242–1243
getcomments() function, inspect, 1206–1207
get_config_vars() function, sysconfig, 1160–1163
getcontext(), decimal module, 201–202
gmtime() function, os.path, 254
get_current_history_length(), readline, 832–834
getcwd() function, os, 1112
get_data() function, pkgutil, 1255–1258
get_data() method pkgutil, 1097
sys, 1095–1097
zipimport, 1246
getdefaultencoding() function, sys, 1058–1059
getdefaultlocale() function, codecs, 298
getdoc() function, inspect, 1206
getfloat() method, ConfigParser, 867–868
getfqdn() function, socket, 565
get_history_item(), readline, 832–834
gethostbyaddr() function, socket, 565
gethostname() function, socket, 563–564
gethostbyname() function, socket, 563, 577–580
getinfo() method, zipfile, 458–459
getint() method, ConfigParser, 867
getline() function, linecache, 263–264
get_logger(), multiprocessing, 539–540
getmember(), tarfile, 449–450
getmembers() function, inspect, 1201–1203, 1204–1206
getmembers(), tarfile, 449–450
G
getmoduleinfo() function, inspect, 1201–1203
getmro() function, inspect, 1212–1213
getmtime() function, os.path, 254
getnames(), tarfile, 449
getnode() function, uuid, 684–686
get_opcodes(), difflib, 67
getopt() function, getopt, 771
getopt module, 770–777
abbreviating long-form options, 775
complete example of, 772–775
defined, 769
ending argument processing, 777
function arguments, 771
GNU-style option parsing, 775–777
long-form options, 772
optparse replacing, 777, 779–781
purpose of, 770–771
reference guide, 777
short-form options, 771–772
getpath module
defined, 769
element of, 836–837
purpose of, 836
reference guide, 838
using without terminal, 837–838
get_path(), sysconfig, 1166
get_path_names() function, sysconfig, 1163–1164
get_paths() function, sysconfig, 1164–1166
get_platform() function, sysconfig, 1167
getprotobynname(), socket, 567
get_python_version() function, sysconfig, 1167–1168
getreader() function, codecs, 298
getrecursionlimit() function, sys, 1067–1068
getrefcount() function, sys, 1065
get_referents() function, gc, 1138–1139
get_referrers() function, gc, 1147–1148
getreusage() function, resource, 1134–1135
get_scheme_names() function, sysconfig, 1163–1166
getservbyname(), socket, 566
getsignal(), signal, 499–501
getsize() function, os.path, 254
getsockname() method, socket, 580
getsource() function, inspect, 1207–1208
get_source() method, zipimport, 1243–1244
getsourcelines() function, inspect, 1207–1208
getstate() function, random, 213–214
get_suffixes() function, imp, 1236–1237
gettimeofday() function, time, 177
Groups
argparse argument, 810–812
character, formatting numbers with locale, 916
data, in itertools, 151–153
dissecting matches with, 30–36
optparse, 791–793
groups() method, Match object, 31–36
gzip module
purpose of, 430
reading compressed data, 433–434
reference guide, 436
writing compressed files, 431–433
GzipFile, 431–433, 434–436
GIL (Global Interpreter Lock)
controlling threads with sys, 1074–1078
debugging threads with sys, 1078–1080
Index

H

handle() method,
SocketServer, 610
handle_close() method,
asyncore, 621, 623–625
handle_connect() hook,
asyncore, 621
Handler, implementing with
asynchat, 632–634
handle_read() method,
asyncore, 623, 628–629
handle_request(),
SocketServer, 609
Handles, closing open, 169–170
handle_write() method,
asyncore, 623
Hanging indents, textwrap, 12–13
Hard limits, resource, 1136
has_extn(), SMTP
encryption, 730
hashlib module
creating hash by name, 471–472
incremental updates, 472–473
MD5 example, 470
purpose of, 469
reference guide, 473
sample data, 470
SHA1 example, 470–471
has_key() function,
timeit, 1034–1035
has_option(),
ConfigParser, 866–867
has_section(),
ConfigParser, 865–866
Headers
adding to outgoing request in
urllib2, 661–662
creating and setting Cookie, 678
encoding Cookie, 680–681
receiving and parsing Cookie,
681–682
setting in BaseHTTPServer,
650–651
“Heads,” picking random items, 216
Heap sort algorithm. See heapq
module
heapify() method, heapq,
90–92
heappop() method, heapq,
90–91
heapq module
accessing contents of heap, 90–92
creating heap, 89–90
data extremes from heap, 92–93
defined, 69
element data, 88
purpose of, 87–88
reference guide, 92–93
heapreplace() method, heapq,
91–92
Heaps, defined, 88
Help command, cmd,
840, 842–843
Help for modules,
pydoc, 920–921
Help messages, argparser,
805–807
Help messages, optparse
application settings, 793–795
organizing options, 791–793
overview of, 790–791
hexdigest() method
calculating MD5 hash,
hashlib, 470–471
digest() method vs., 475–476
HMAC message signatures, 474
SHA vs. MD5, 474–475
HistoryCompleter class,
readline, 832–834
hmac module
binary digests, 475–476
message signature applications,
476–479
purpose of, 473
reference guide, 479
SHA vs. MD5, 474–475
signing messages, 474
Hooks, triggering actions in
readline, 834–835
Hostname
parsing URLs, 639
socket functions to look up,
563–565
Hosts
multicast receiver running on
different, 590–591
using dynamic values with
queries, 359–362
hour attribute, time class, 181
HTML help for modules, pydoc,
920–921
HTML output, cgitb, 972
HTMLCalendar, formatting, 192
HTTP
BaseHTTPServer. See
BaseHTTPServer module
cookies. See Cookie module
HTTP GET, 644, 657–660
HTTP POST, 646–647, 661
Human-consumable results, JSON,
692–694
Hyperbolic functions, math,
243–244
hypot() function, math, 242–243
I
I/O operations
asynchronous network. See
asyncore module
codecs, 287–289, 295–298
waiting for I/O efficiently. See
select module
id() values, pickle, 342–343
idpattern class attribute,
string.Template, 7–9
ifilter() function,
iter tools, 150
ifilterfalse() function,
iter tools, 150–151
ignore command, breakpoints in
pdb, 999–1001
ignore mode, codec error
handling, 292–293, 295
IGNORECASE regular expression
flag
abbreviation, 45
creating back-references in re, 53
searching text, 37–38
Ignoring breakpoints, 999–1001
Ignoring signals, 502
Illegal jumps, execution flow in pdb,
1005–1008
imap() function, itertools,
145–146, 148
IMAP (Internet Message Access
Protocol). See also imaplib
module, 738–739
IMAP4_SSL. See imaplib
module, 738–739
IMAP4_stream, 739
imaplib module
connecting to server, 739–741
defined, 727
deleting messages, 756–758
example configuration, 741
fetching messages, 749–752
imaplib module (continued)
listing mailboxes, 741–744
mailbox status, 744–745
moving and copying messages, 755–756
purpose of, 738–739
reference guide, 758
search criteria, 747–749
searching for messages, 746–747
selecting mailbox, 745–746
uploading messages, 753–755
variations, 739
whole messages, 752–753
IMMEDIATE isolation level, sqlite3, 374
imp module
defined, 1235
example package, 1236
finding modules, 1237–1238
loading modules, 1238–1240
module types, 1236–1237
purpose of, 1235–1236
reference guide, 1240
Impermanent references to objects.
See weakref module
Import errors, 1094–1095
Import hooks, 1083
Import mechanism, Python. See imp module
Import path, site
adding user-specific locations to, 1047–1048
configuring, 1046–1047
path configuration files, 1049–1051
Import path, sys, 1081–1083
Imported modules, sys, 1080–1081
Importer cache, sys, 1097–1098
ImportError exception
overview of, 1221–1222
raised by find_module(), 1238
sys, 1094–1095
Imports. See also Modules and imports
from shelf, 1085–1091
target functions in multiprocessing, 530–531
 ImportError, 1233
In-memory approach to compression and decompression, 422–423, 436–438
In-memory databases, sqlite3, 376–378
in-place filtering, fileinput, 887–889
In-place operators, 158–159
INADDR_ANY, socket
choosing address for listening, TCP/IP, 579
receiving multicast messages, 590
IncompleteImplementation, abc, 1180–1181
Incremental compression and decompression
bz2 module, 438–439
zlib module, 423–424
Incremental encoding, codecs, 301–303
Incremental updates, hashlib, 472–473
IncrementalDecoder, codecs, 301–303, 312
IncrementalEncoder, codecs, 301–303, 312
Indent, JSON format, 692–693
Indentation, paragraph
combining dedent and fill, 11–12
hanging, 12–13
removing from paragraph, 10–11
IndexError exception,
1222–1223
inet_aton(), IP address
in socket, 570–571
inet_ntoa(), IP address
in socket, 570–571
inet_ntop(), IP address
in socket, 571
inet_pton(), IP address
in socket, 571
INF (infinity) value, testing in
math, 224–225
infile arguments, saving result data in
trace, 1021
INFO level, logging, 881–882
info() method, urllib2, 658
infolist() method, zipfile, 458
__init__() method
asyncore, 621
inspect, 1205–1206
threading, 527–528
Initialization
array, 84–85
Counter, 70–71
Input
alternative cmd, 849–851
converting iterators, 145–146
searching text using multiline, 38–39
standard streams with codecs, 295–298
streams with sys, 1063–1064
input() function, fileinput, 884
Input history, readline, 832–834
input_loop() function, readline, 826
insert statements, sqlite3, 355
Inserting, bisect, 93–95
insert_text(), readline, 835
insort() method, bisect, 93–95
insort_left() method, bisect, 95–96
insort_right() method, bisect, 95–96
inspect module
class hierarchies, 1210–1212
defined, 1169
documentation strings, 1206–1207
element module, 1200–1201
inspecting classes, 1204–1206
inspecting modules, 1203–1204
method and function arguments, 1209–1210
method resolution order, 1212–1213
module information, 1201–1203
purpose of, 1200
reference guide, 1217
retrieving source, 1207–1208
stack and frames, 1213–1216
Inspecting live objects. See inspect module
Installation paths, sysconfig, 1163–1166
install() function, application
localization with gettext, 908
Integers
converting floating-point values to, 226–227
generating random, 214–215
identifying signals by, 498
SQL support for columns, 363–366
Interacting with another command, subprocess, 490–492
Interactive debugger. See pdb module
Interactive help for modules, pydoc, 921
Interactive interpreter, starting pdb debugger, 977
Interactive prompts, interpreter settings in sys, 1059–1060
Interface
checking with abstract base classes. See abc module
programming with trace, 1018–1020
Internationalization and localization
 cultural localization API. See locale module
message catalogs. See gettext module
overview of, 899
reference guide, 920
Internet
controlling spiders, 674–677
encoding binary data, 670–674
HTTP cookies. See Cookie module
implementing Web servers. See BaseHTTPServer module
JavaScript Object Notation. See json module
network resource access. See urllib module; urllib2 module
overview of, 637–638
splitting URLs into components. See urlparse module
universally unique identifiers. See uuid module
XML-RPC client library. See xmlrpclib module
XML-RPC server. See SimpleXMLRPCServer module
Internet Message Access Protocol (IMAP). See also imaplib module, 738–739
Interpolation
ConfigParser, 875–878
templates vs. standard string, 5–6
InterpolationDepthError, ConfigParser, 877
Interpreter
compile-time configuration. See sysconfig module
getting information about current, 1129–1130
starting pdb debugger within, 977
Interpreter settings, sys
build-time version information, 1055–1057
command-line option, 1057–1058
displayhook, 1060–1062
install location, 1062
interactive prompts, 1059–1060
Unicode defaults, 1058–1059
intro attribute, configuring cmd, 847–848
Introspection API,
 SimpleXMLRPCServer module, 724–726
Inverse hyperbolic functions, math, 244
Inverse trigonometric functions, math, 243
Invertcaps, codec, 307–312
IOError exception
argparse, 818
overview of, 1221
retrieving package data with sys, 1096
IP addresses, socket
AF_INET sockets for IPv4, 562
AF_INET6 sockets for IPv6, 562
choosing for listening, 577–580
finding service information, 566–568
looking up hosts on network, 563–565
for multicast, 588, 590–591
representations, 570–571
IP_MULTICAST_TTL, TTL, 588–589
IPPROTO_prefix, socket, 568
IS-8601 format, datetime objects, 189–190
is_() function, operator, 154
isinstance(), abc, 1178, 1179
islice() function, itertools, 144
ismethod() predicate, inspect, 1205
isnan() function, checking for NaN, 226
is_not() function, operator, 154
Isolation levels, sqlite3, 372–376
is_package() method, zipimport, 1244
isset() method, threading, 517
is_set(), multiprocessing, 545–546
issubclass(), abc, 1178, 1179
is_tarfile() function, testing tar files, 448–449
is_zipfile() function, testing ZIP files, 457
Item getters, operator, 159–161
items(), ConfigParser, 865
items(), mailbox, 765
iter() function, ElementTree, 388–390
Iterator functions. See itertools module
iterdump() method, Connection, 376–378
iteritems(), mailbox, 765
iterparse() function, ElementTree, 394–396
itertools module
converting inputs, 145–146
defined, 129
filtering, 148–151
grouping data, 151–153
merging and splitting iterators, 142–145
performance analysis of loops, 1197–1198
producing new values, 146–148
purpose of, 141–142
reference guide, 153
izip() function, itertools, 143–144, 148
JavaScript Object Notation. See json module

join() method
  in multiprocessing, 534–537, 542–543, 554
  in os.path, 252–253
  in threading, 510–511

json module
  defined, 638
  encoder and decoder classes, 697–700
  encoding and decoding simple data types, 690–691
  encoding dictionaries, 694
  human-consumable vs. compact output, 692–694
  mixed data streams, 701–702
  purpose of, 690
  reference guide, 702
  working with custom types, 695–697
  working with streams and files, 700–701

JSONDecoder class, JSON, 699–700, 701–702

JSONEncoder class, 698–699

js_output() method, Cookie, 682–683

jump command, pdb
  changing execution flow, 1002
  illegal jumps, 1005–1008
  jump ahead, 1002–1003
  jump back, 1004

jumpahead() function, random, 220–221

Junk data, difflib, 65–66

K

kevent() function, select, 608

KeyboardInterrupt exception, 502, 1223

KeyDownError exception, 1034–1035, 1223

kill() function, os. fork(), 1123

kqueue() function, select, 608

Language, installing message catalogs in directories by, 902

Language tools
  abstract base classes. See abc module
  built-in exception classes. See exceptions module
  cultural localization API.
    See locale module
  inspecting live objects. See inspect module
  message translation and catalogs.
    See gettext module
  nonfatal alerts with warnings module, 1170–1177
  overview of, 1169–1170
  Python bytecode disassembler.
    See dis module
  last-in, first-out (LIFO) queue, 97
  ldexp() function, math, 228–229
  lgamma() function, math, 232–233

Libraries, logging, 878

LIFO (last-in, first-out) queue, 97

LifoQueue, 97

Limits, resource, 1135–1138

Line number, warning filters, 1170, 1174

Line-oriented command processors.
  See cmd module

linenocache module
  defined, 247
  error handling, 263–264
  handling blank lines, 263
  purpose of, 261
  reading Python source files, 264–265
  reading specific lines, 262
  reference guide, 265
  test data, 261–262

load() function, pickle, 336

Local context, decimal, 204–205

local() function, threading, 526–528

Local variables in tracebacksw, cgitb, 968–971

Locale directory, 902–904

locale module, 909–918
  currency, 915–916
  date and time formatting, 917–918
  defined, 899
  formatting numbers, 916–917
Index 1281

parsing numbers, 917
probing current locale, 909–915
purpose of, 909
reference guide, 918
localeconv() function, locale, 911–915

Localization
cultural localization API. See locale module
message translation and catalogs. See gettext module

localtime() function, time, 177
local_variable value, inspect, 1214

Location
for interpreter installation in sys, 1062
standard I/O streams, 297–298
temporary file, 270–271
test, with doctest, 936–939

Lock object
access control with multiprocessing, 546–547
access control with threading, 517–520
as context managers, 522–523
re-entrant locks, 521–522
synchronizing processes with multiprocessing, 547–548
synchronizing threads with threading, 523–524
lock_holder(), threading, 519–521

Locking modes, sqlite3. See isolation levels, sqlite3
log() function, logarithms in math, 235–236

Log levels, logging, 880–882
Logarithms, math, 234–238
logging module, 878–883
directing output, 880–882

Math
alternate representations, 227–229
angles, 238–240
common calculations, 230–233
converting to integers, 226–227
defined, 197
exponents and logarithms, 234–238
hyperbolic functions, 243–244
positive and negative signs, 229–230
purpose of, 223
reference guide, 244–245
special constants, 223–224
special functions, 244–245
testing for exceptional values, 224–226
trigonometry, 240–243

Mailboxes
IMAP4
listing archive subfolders, 743–744
retrieving account, 741–743
search criteria, 747–748
searching for messages, 746–747
selecting, 745–746
status conditions, 744–745

Maildir format, mailbox, 762–764

Mailfrom argument, SMTPServer, 734

Makedirs() function, os, 1119

Makefile() function, codecs, 307–313
maketrans() function, string, 4–5

Manager, multiprocessing, 550–553

Manipulation, array, 85
map() function, vs. imap(), itertools, 145
MapReduce, multiprocessing, 555–559

match() function, re, 26–30

Match object
compiling expressions, 14–15
dissecting matches with groups, 31
finding multiple matches, 15–16
finding patterns in text, 14
pattern syntax, 17

match.groups(), re, 32

Math module
alternate representations, 227–229
angles, 238–240
common calculations, 230–233
converting to integers, 226–227
defined, 197
exponents and logarithms, 234–238
hyperbolic functions, 243–244
positive and negative signs, 229–230
purpose of, 223
reference guide, 244–245
special constants, 223–224
special functions, 244–245
testing for exceptional values, 224–226
trigonometry, 240–243
Index

Mathematics
fixed and floating-point numbers.
See decimal module
mathematical functions.
See math module
overview of, 197
pseudorandom number
generators. See random
module
rational numbers in fractions
module, 207–211

max attribute
date class, 184
time class, 181
max() function, sqlite3, 380–381
Max-heaps, heapq, 88
maxBytes, rotating log files, 880
Maximum values, sys, 1069
maxint, sys, 1069
MAX_INTERPOLATION_DEPTH,
substitution errors, 877
maxtasksperchild parameter,
process pools, 554
maxunicode, sys, 1069
mbox format, mailbox 762
mbox format, mailbox module, 759–762
MD5 hashes
calculating in hashlib, 470
UUID 3 and 5 name-based values
using, 686–688
vs. SHA for hmac, 474–475
Memory management. See gc
module
Memory management and limits,
sys
byte ordering, 1070–1071
floating-point values, 1069–1070
maximum values, 1069
object size, 1066–1068
recursion, 1068–1069
reference counts, 1065–1066
Memory-map files. See mmap
module
MemoryError exception,
1224–1225
Merging iterators, itertools, 142–144
Mersenne Twister algorithm,
random based on, 211
Message catalogs,
internationalization. See
ggettext module
Message signatures, hmac, 474, 476–479
Message terminators, asynchat, 629–630
message_ids argument, IMAP4, 749–752
message_parts argument,
IMAP4, 749–752
Messages
combining calls in XML-RPC
into single, 712–714
passing to processes with
multiprocessing,
541–545
reporting informational, with
logging, 878–883
sending SMTP, 728–730
setting log levels, 880–882
warning filter, 1170
Messages, IMAP4 email
deleting, 756–758
fetching, 749–752
moving and copying, 755–756
retrieving whole, 752–753
search criteria, 747–748
searching mailbox for, 746–747
uploading, 753–755
Meta path, sys, 1098–1101
Metacharacters, pattern syntax
anchoring instructions, 24–26
caracter sets, 20–24
escape codes for predefined
character sets, 22–24
expressing repetition, 17–20
overview of, 16–17
__metaclass__, abstract base
classes, 1178
Metadata
accessing current line in
fileinput, 886–887
copying file, 274–275
reading from archive in
tarfile, 449–450
reading from archive in
zipfile, 457–459
metavar argument, help in
optparse, 791
Method Resolution Order (MRO),
for class hierarchies, 1212–1213
__methodHelp(), Introspection
API, 724–725
Methods
arguments for, 1209–1210
concrete, in abstract base classes,
1181–1182
configuration settings, 864–869
disassembling class, 1189–1190
overriding base class in cmd,
845–846
microsecond attribute
date class, 182–183
time class, 181–182
MIME content, uploading files in
urllib2, 664–667
min attribute
date class, 184
time class, 181
min() function, customizing in
sqlite3, 380–381
Min-heaps, heapq, 88
minute attribute, time, 181
misc_header attribute, cmd,
847–848
Mixed content streams
bz2, 439–440
zlib, 424–425
mkdir() function, creating
directories in os, 1118–1119
mkdtemp() function, tempfile,
267–270
mtime() function, time, 177
mmap module
defined, 248
purpose of, 279
reading, 279–280
reference guide, 284
regular expressions, 283–284
writing, 280–283
MMDF format, mailbox, 768
modf() function, math, 227–229
Modules
gathering information with
inspect, 1201–1203
import mechanism for loading
code in. See imp module
inspecting with inspect,
1203–1204
localization, with gettext,
908
online help for, 920–921
Index

running tests in doctest by, 942–943
warning filters, 1170, 1173–1174

Modules and imports
built-in modules, 1081
custom importers, 1083–1085
custom package importing, 1091–1093
handling import errors, 1094–1095
import path, 1081–1083
imported modules, 1080–1081
importer cache, 1097–1098
importing from shelve, 1085–1091
meta path, 1098–1101
package data, 1095–1097
reloading modules in custom importer, 1093–1094

Modules and packages
loading Python code from ZIP archives. See zipimport module
overview of, 1235
package utilities. See pkgutil module
Python’s import mechanism. See imp module
reference guide, 1258
month attribute, date class, 182–183
monthcalendar() method, Calendar, 192, 194–195
Morsel object, Cookie, 678–680, 681–683
most_common() method, Counter, 72–73
move() function
moving directory with shutil, 278
moving messages in imaplib, 755–756
MP3 files, converting to RSS feed, 883–886
MRO (Method Resolution Order), for class hierarchies, 1212–1213
MultiCall class, xmlrpclib module, 712–714
Multicast groups, defined, 588
Multicast messages
example output, 590–591
overview of, 587–588
receiving, 589–590
sending, 588–589
UDP used for, 562
Multiline input, text search, 38–39
MULTILINE regular expression flag, 38–39, 45
MultiPartForm class, urllib2, 666
Multiple simultaneous generators, random, 219–221
multiprocessing module basics, 529–530
controlling access to resources, 546–547
controlling concurrent access to resources, 548–550
demon processes, 532–534
determining current process, 531–532
importable target functions, 530–531
logging, 539–540
managing shared state, 550–551
MapReduce implementation, 555–559
passing messages to processes, 541–544
process exit status, 537–538
process pools, 553–555
purpose of, 529
reference guide, 559
shared namespaces, 551–553
signaling between processes, 545–546
subclasseing Process, 540–541
synchronizing operations, 547–548
terminating processes, 536–537
waiting for processes, 534–536
Mutually exclusive options, argpase, 812–813
my_function(), doctest, 922
MyThreadWithArgs, subclassing Thread, 514

N
{n}, repetition in pattern syntax, 18
Name-based values, UUID 3 and 5, 686–688
Named groups
creating back-references in re, 52–53
modifying strings with patterns, 56
syntax for, 33–34
verbose mode expressions vs., 41
NamedError exception, 1225
namedlist() method, reading metadata in zipfile, 458
Namespace
creating shared, multiprocessing, 551–555
creating UUID name-based values, 686–688
incorporating into APIs, 716–719, 720–721
as return value from parse_args(), 797
Naming
current process in multiprocessing, 530–531
current thread in threading, 507–508
hashes, 471–472
logger instances, 882–883
SimpleXMLRPCServer alternate API, 716–717
SimpleXMLRPCServer arbitrary API, 719
SimpleXMLRPCServer dotted API, 718–719
NaN (Not a Number), testing in math, 225–226
Nargs option, optparse, 789–790
ndiff() function, difflib, 64–66
Negative look-ahead assertion, regular expressions, 47–48
Negative look-behind assertion, regular expressions, 48–49
Negative signs, math, 229–230
Nested data structure, pprint, 126
Index 1285

configuration. See sys module
getting information with
platform, 1131–1133
portable access to features. See
os module
resource management with
resource, 1134–1138
used to build interpreter in sys,
1056–1057
version implementation with
platform, 1129–1134
operator module
arithmetic operators, 155–157
attribute and item “getters,”
159–161
combining operators and custom
classes, 161–162
comparison operators, 154–155
defined, 129
logical operations, 154
in-place operators, 158–159
purpose of, 153
reference guide, 163
sequence operators, 157–158
type checking, 162–163
Option actions, optparse,
784–790
Option flags, regular expression
case-insensitive matching, 37–38
embedding flags in patterns,
42–43
input with multiple lines, 38–39
Unicode, 39–40
verbose expression syntax, 40–42
Option groups, optparse, 791–793
Option values, optparse, 781–784
Optional arguments,
argparse, 810
Optional parameters, trace, 1022
OptionParser, optparse
creating, 777–778
help messages, 790–791, 793–795
setting option values, 781–784
Options, ConfigParser
accessing configuration
settings, 865
defined, 862
as flags, 868–869
testing if values are present,
865–867
Options, ConfigParser file
removing, 870
search process, 872–875
option_string value, argparse, 820
Optparse, 793–795
optparse module
argparse vs., 795–796, 798
creating OptionParser,
777–778
defined, 769
help messages, 790–795
option actions, 784–790
option values, 781–784
purpose of, 777
reference guide, 795
replacing getopt with, 779–781
short- and long-form options,
778–779
OR operation, re, 37
OrderedDict, collections,
82–84
os module
creating processes with
os.fork(), 1122–1125
defined, 1045
directories, 1118–1119
file descriptors, 1116
file system permissions,
1116–1118, 1127–1128
pipes, 1112–1116
process environment, 1111–1112
process owner, 1108–1110
process working directory, 1112
purpose of, 1108
reference guide, 1128–1129
running external commands,
1121–1122
spawn() family of functions,
1127
symbolic links, 1119
waiting for child process,
1125–1127
walking directory tree, 1120–1121
os.environ object, 1111–1112
OSError exception, 1110,
1226–1227
os.exit(), atexit, 892
os.fork(), creating processes
with, 1122–1125
os.kill() function, signal
receiving signals, 499
sending signals, 501
os.open() method, mmap,
279–280
os.path module
building paths, 252–253
defined, 247
file times, 254–255
normalizing paths, 253–254
parsing paths, 248–251
purpose of, 248
reference guide, 257
testing files, 255–256
traversing directory tree, 256–257
os.stat() function, os.path,
254–255
Outcomes, unittest test, 950–952
Outfile arguments, trace, 1021
Outline nodes, finding in document
with ElementTree, 390–391
Output
capturing errors, 488
capturing when running external
command, 484–485
combining regular and error,
488–489
HTML format in cgitb, 972
JSON compact, 692–694
limiting report contents in
pstats, 1028–1029
standard streams with codecs,
295–298
streams with sys, 1063–1064
unpredictable, in doctest,
924–928
OverflowError exception, 225,
1227–1228
overlapping events, sched,
896–897
P
Packages
import mechanism for loading
code. See imp module
retrieving data with sys,
1095–1097
utilities for. See pkgutil
module
Packaging data into strings, struct,
102–103
pack_into() method, struct,
105–106
Paragraphs, formatting with
textwrap. See textwrap
module
Parameters, query, 360–362
Pareto (power law), 222
paretovariate() function, random, 222
parse() function, ElementTree, 387
parse_and_bind() function, readline, 823–824
parse_args() parsing command line with argparse, 796–797
parsing command line with optparse, 778
setting optparse values as default, 781–782
PARSE_DECLTYPES, sqlite3, 363–366
ParseFlags(), imaplib, 752
parseline(), cmd, 846
Parsing command-line options. See Command-line option parsing
Cookie headers, 681–682
dates and times, 189–190
numbers with locale, 917
paths with os.path, 247–251
shell-style syntaxes. See shlex module
times, 178
unparsing URLs with urlparse, 641–642
URLs with urlparse, 638–640
Parsing, ElementTree
creating custom tree builder, 396–398
parsed note attributes, 391–393
strings, 398–400
traversing parsed tree, 388–390
watching events while, 393–396
XML documents, 387–388
partial objects, functools
acquiring function properties, 132–133
defined, 130
other callables, 133–136
overview of, 130–132
partition(), MapReduce, 558
Passwords
opening Unicode configuration files, 863–864
parsing URLs, 639
secure prompt with getpass, 836–839
__path__ attribute, data files, 1244–1246
pathname2url() function, urllib, 655–657
Paths
building from other strings in os.path, 252–253
configuration files in site, 1049–1051
installation using sysconfig, 1163–1166
joining URLs with urlparse, 642–643
managing with PKG files, 1251–1253
normalizing in os.path, 253–254
parsing in os.path, 247–251
retrieving network resources with URLs vs., 655–657
pattern attribute, string.Template, 8
Pattern matching filenames, with glob, 257–260, 315–317
listing mailbox folders in imaplib, 743–744
searching and changing text. See re module
warning filters with, 1172–1174
Pattern syntax, re
anchoring, 24–26
caracter sets, 20–24
escape codes, 22–24
overview of, 16–17
repetition, 17–20
pdb module
breakpoints, 990–1002
changing execution flow, 1002–1009
customizing debugger with aliases, 1009–1011
defined, 920
examining variables on stack, 981–984
handing previous interactive exception, 1073
navigating execution stack, 979–981
purpose of, 975
saving configuration settings, 1011–1012
starting debugger, 976–979
stepping through program, 984–990
Peer argument, SMTPServer, 734
PendingDeprecationWarning, 1233
Performance analysis
of loops with dis, 1192–1198
with profile, 1022–1026
with pstats, 1027–1031
Permissions
copying file, 273
copying file metadata, 274–276
file system functions, 1116–1117
UNIX Domain Sockets, 586
Permutations, random, 216–218
Persistence. See Data persistence and exchange
Persistent storage of objects. See shelve module
pformat() function, pprint, 124–125
Picking random items, random, 215–216
pickle module
binary objects sending objects using, 711
circular references, 340–343
defined, 333
encoding and decoding data in strings, 335–336
importing, 335
insecurity of, 334
json module vs., 690, 692
problems reconstructing objects, 338–340
purpose of, 334
reference guide, 343
unpicklable objects, 340
working with streams, 336–338
pipe symbol (|), 35, 413–418
Pipes
connecting segments of, 489–490
managing child processes in os, 1112–1116
working directly with, 486–489
PKG files, managing paths with, 1251–1253
pkgutil module
defined, 1235
development versions of packages, 1249–1251
managing paths with PKG files, 1251–1253
nested packages, 1253–1255
package data, 1255–1258
package import paths, 1247–1249
purpose of, 1247
reference guide, 1258
Placeholders, queries in sqlite3, 359–362
Plain-text help for modules, pydoc, 920
platform() function, 1130–1131
platform module
defined, 1045
executable architecture, 1133–1134
interpreter, 1129–1130
operating system and hardware info, 1131–1133
platform() function, 1130–1131
purpose of, 1129
reference guide, 1134
Platform-specific options, select, 608
Platform specifier, sysconfig, 1167
Plural values, gettext, 905–907
pm() function, cgitb, 978–979
Podcasting client, threaded, 99–102
PodcastListToCSV, TreeBuilder, 398
poll() function, select, 595, 603–608
POLLERR flag, select, 607
POLLHUP flag, select, 606
Pool class, multiprocessing
MapReduce implementation, 555–559
process pools, 553–555
Popen class, subprocess module
connecting segments of pipe, 489–490
defined, 482
interacting with another command, 490–492
signaling between processes, 492–498
working directly with pipes, 486–489
popen() function, pipes, 1112–1116
Populating, deque, 76–77
Ports
getting service information with socket, 566–568
parsing URLs in urlparse, 639
SocketServer echo example, 615
Positional arguments, argparsing, 810
Positional parameters, queries in sqlite3, 360
Positive look-ahead assertion, regular expressions, 46–47
Positive look-behind assertion, regular expressions, 49–50
Positive signs, math, 229–230
POSIX systems
access() function warnings, 1128
detail available through sysconfig, 1161–1162
installation paths with sysconfig, 1163–1166
vs. non-POSIX parsing with shlex, 869–871
Post-mortem debugging, 978–979
POST requests
BaseHTTPRequest, 646–647
client, 661
SimpleXMLRPCServer, 715–716
postcmd(), cmd, 846
postloop(), cmd, 846
post_mortem() function, cgitb, 978–979
pow() function, math, 234
pprint() function, 123–125
pprint module
arbitrary classes, 125
controlling output width, 126–127
formatting, 124–125
limiting nested input, 126
printing, 123–124
purpose of, 123
recursion, 125–126
reference guide, 127
Pre-instance context, decimal, 205–206
prec attribute, decimal contexts, 202–203
Precision, decimal module contexts
local context, 204–205
overview of, 202–203
pre-instance context, 205–206
rounding to, 203–204
threads, 206–207
precmd(), cmd, 846
Predicate functions, inspect, 1203–1204
Predicting names, tempfile, 269–270
Prefix_chars parameter, argparsing, 803
Prefixes, argparse option, 802–803
Preinput hook, readline, 834–835
preloop(), cmd, 846
Pretty-print data structures. See also pprint module, 123–127
pretty-print (pp) command, pdb, 983
Pretty-printing XML,
ElementTree, 401–403
print (p) command, pdb, 983–984
print_callees(), pstats, 1030–1031
print_callers(), pstats, 1030–1031
print_event(), sched, 895
print_exc() function, traceback, 959–960
print_exception() function, traceback, 960–961
print_stack() function, traceback, 963–964
Priorities, event, 897
PriorityQueue, 98–99
prmonth() method, calendar, 191
Probing current locale, locale, 909–915
Process environment, os, 1111–1112
Process exit status, multiprocessing, 537–538
Process groups, subprocess, 494–496
Process owners, changing with os, 1108–1110
Process pools, multiprocessing, 553–555
Process working directory, retrieving with os, 1112
Processes
creating with os.fork(), 1122–1125
platform independent. See subprocess module
running external commands with os, 1121–1122
waiting for child, 1125–1127
Processes and threads
asynchronous system events. See signal module
managing concurrent operations. See threading module
managing processes like threads. See multiprocessing module
overview of, 481
spawning additional processes. See subprocess module
process_message() method, SMTPServer class, 734–735
Processor clock time, 174–176
process_request() method, SocketServer, 610
profile module
defined, 920
running in context, 1026
running profiler, 1023–1026
Program shutdown callbacks, atexit, 890–894
Programming
following flow of. See trace module
restarting with pdb, 1008–1009
starting pdb debugger within, 977–978
stepping through execution in pdb, 984–990
tracing as they run, 1101–1107
Prompts
cmd command, 840
configuring prompt attribute in cmd, 847–848
interactive interpreter in sys, 1059–1060
Properties
abstract, in abc, 1182–1186
acquiring function, in functools, 136–138
functools, 132–133
retrieving file, in os.path, 254–255
setting Element, 403–405
showing exceptions, in cgitb, 971–972
socket, 562
Protocol handlers
asynchronous. See asynchat module
creating custom, with urllib2, 667–670
Proxies, weakref, 108–109
Proxy server, smtpd, 737–738
pstats module
caller and callee graphs, 1029–1031
limiting report contents, 1028–1029
reference guide, 1031
saving and working with statistics, 1027–1028
Pseudorandom number generators. See random module
.pth extension, path configuration files, 1049–1051
.public() method, MyService, 723
PureProxy class, 737–738
.put() method
basic FIFO queue, 97
LifoQueue, 97
.pyc file, Python ZIP archives, 466–467
pyc module
defined, 920
purpose of, 1039–1041
reference guide, 1043
scanning for classes, 1041–1042
scanning for functions, 1042–1043
pygettext module, 919–921
pydoc module, 900–901
Python
bytecode disassembler. See dis module
import mechanism. See imp module
loading code from ZIP archives. See zipimport module
reading source files, 264–265
version and platform, sysconfig, 1167–1168
ZIP archives, 466–467
python_build() function, 1133–1134
python_compiler() function, 1133–1134
PYTHONUSERBASE environment variable, 1048
python_version() function, 1133–1134
python_version_tuple() function, 1133–1134
PyUnit. See unittest module
PyZipFile class, Python ZIP archives, 466–467

Q
Queries, sqlite3
metadata, 357–358
retrieving data, 355–357
using variables with, 359–362
question mark. See ? (question mark)
question mark, colon (?,:), noncapturing groups, 36–37
Queue module
basic FIFO queue, 96–97
building threaded podcast client, 99–101
communicating between processes with multiprocessing, 541–545
defined, 70
LifoQueue, 97
PriorityQueue, 98–99
purpose of, 96
reference guide, 101–102
thread-safe FIFO implementation, 96–102
tracing references with gc, 1139–1141
QUOTE_ALL option, csv, 413
Quote strings, shlex, 852–854
quote() function, urllib, 655
QUOTE_MINIMAL option, csv, 413
QUOTE_NONE option, csv, 413
QUOTE_NONNUMERIC option, csv, 413
quote_plus() function, urllib, 655
Quoting behavior, csv, 413
Radians, math, 238–243

Raised exceptions
AssertionError, 1217–1218
AttributeError, 1218–1219
EOFError, 1220
FloatingPointError, 1220
GeneratorExit, 1220–1221
ImportError, 1221–1222
IndexError, 1222–1223
IOError, 1223
KeyboardInterrupt, 1223
KeyError, 1223
MemoryError, 1224–1225
NameError, 1225
NotImplementedError, 1225–1226
OSError, 1226–1227
OverflowError, 1227–1228
ReferenceError, 1228–1229
RuntimeError, 1229–1230
SyntaxError, 1230
SystemError, 1230
SystemExit, 1230
TypeError, 1230–1231
UnboundLocalError, 1231–1232
UnicodeError, 1232
ValueError, 1232
ZeroDivisionError, 1232

raises_exception(), XML-RPC, 713–714

RAM (random access memory), in-memory databases, 376

randint() function, random integers, 214–215
random access memory (RAM), in-memory databases, 376

Random class, 219–221
random() function, generating random numbers, 211–212
random integers, 214–215
saving state, 213–214
seeding, 212–213

Random module
defined, 197
generating random numbers, 211–212
generating random values in UUID 4, 688–689

multiple simultaneous generators, 219–221
nonuniform distributions, 222–223
permutations, 216–218
picking random items, 215–216
purpose of, 211
random integers, 214–215
reference guide, 223
sampling, 218–219
saving state, 213–214
seeding, 212–213
SystemRandom class, 221–222
Random numbers
generating with random, 211–212
UUID 4 values, 688–689

randrange() function, random, 215

Rational numbers
approximating values, 210–211
arithmetic, 210
creating fraction instances, 207–210
Fraction class, 207
raw_decode() method, JSON, 701–702
raw_input() function, readline, 827
rcpttos argument, SMTPServer class, 734
Re-entrant locks, threading, 521–522

re module
compiling expressions, 14–15
constraining search, 26–30
dissecting matches with groups, 30–36
finding patterns in text with, 14
looking ahead or behind, 45–50
modifying strings with patterns, 56–58
multiple matches, 15–16
overview of, 13
reference guide, 60
retrieving account mailboxes in imaplib, 742
Self-referencing expressions, 50–56
splitting with patterns, 58–60

Readable results, JSON vs. pickle, 692
Readable sockets, poll() function, 605
Readable sockets, select() function, 596–597

reader() function
isolation levels in sqlite3, 373
reading data from CSV file, 411–412
read_history_file(), readline, 832–834

Reading
compressed data in gzip, 433–434
compressed files in bz2, 442–443
configuration files in ConfigParser, 862–864
data from CSV file, 411–412
Maildir mailbox, 764
mbox mailbox, 760–761
metadata from archive in tarfile, 449–450
metadata from archive in zipfile, 457–459
text files efficiently. See linecache module
using mmap to create memory-mapped file, 279–280
read_init_file() function, readline, 824
readline module
accessing completion buffer, 828–831
completing text, 824–827
configuring, 823–824
as default mode for Cmd() to interact with user, 849–851
defined, 769
hooks for triggering actions, 834–835
purpose of, 823
reference guide, 835–836
tracking input history, 832–834
readlines() method, 315, 658
readlink() function, symbolic links with os, 1119
readmodule() function,
pyclbr, 1041–1042
readmodule_ex() function,
pyclbr, 1042–1043
Receiver, multicast, 589–590
receive_signal(), signal, 499
Reconstructing objects, problems in pickle, 338–340
recursion() function
inspect, 1214–1215
programming trace interface, 1018–1020
recursion module, trace
calling relationships, 1017–1018
code coverage report information, 1013–1017
example program, 1012
programming interface, 1018–1020
tracing execution, 1012–1013
Recursion
in alias definitions in pdb, 1010–1011
controlling memory in sys, 1068–1069
in deep copy, 120–123
pprint, 125–126
recv()
 echo client, TCP/IP socket, 573–574
 echo server, TCP/IP socket, 573
nonblocking communication and timeouts vs., 594
using poll(), 605–606
redisplay(), readline, 835
ref class, weakref, 107–108
Reference counting, memory management in sys, 1065–1066
ReferenceError exception, 109, 1228–1229
References
finding for objects that cannot be collected, 1146–1148
impermanent, to objects. See weakref module
tracing with gc, 1138–1141
RegexObject, compiling expressions, 14–15
register()
 alternate API names in SimpleXMLRPCServer, 716–717
atexit, 890–891
encoding, 309
registering concrete class in abc, 1179
register_adapter() function, sqlite3, 364–365
register_converter() function, sqlite3, 364–365
Registered handlers, signal, 499–501
register_introspection_functions(), SimpleXMLRPCServer, 724–726
Regular expressions
syntax for. See re module
translating glob patterns to, 318
understanding, 13
using memory-mapped files with, 283–284
Relational database, embedded. See sqlite3 module
Relationships, trace
collecting/reporting on, 1017–1018
release() method
multiprocessing, 548
threading, 523–524
reload() function, imported modules in sys, 1083, 1239–1240
Reloading
imported modules, 1083
modules in custom importer, 1093–1094
remove(), messages from Maildir mailbox, 764–765
removedirs() function, os, 1119
remove_option,
 ConfigParser, 871–872
remove_section,
 ConfigParser, 870–871
repeat() function, itertools, 1032
repeated warnings, 1174–1175
repeater.py script, 491–492
Repeating options, optparse, 786–788
Repetition, pattern syntax, 17–20, 23–24
replace() method, datetime, 184
replace mode
codec error handling, 292
decoding errors, 295
encoding errors, 293
report() function,
 filecmp, 327
REPORT_CDIFF, doctest, 933–934
report_full_closure() function, filecmp, 327–328
reporthook(), urllib, 652
REPORT_NDIFF, doctest, 933
Reports
calling relationships, 1017–1018
code coverage with trace, 1013–1017
detailed traceback. See cgitb module
performance analysis with profile, 1023–1026
performance analysis with pstats, 1027–1031
traceback. See traceback module
REPORT_UDIFF, doctest, 933–934
__repr__() method,
pprint, 125
Request handler, SocketServer, 610–615
Index

Request object, urllib2, 662–664
resolve conflict_handler, argparsed, 808–810
resource limits, resource, 1135–1138
Resource management. See resource module
resource module, 1134–1138
current usage, 1134–1135
defined, 1045
purpose of, 1134
reference guide, 1138
resource limits, 1135–1138
Restricting access to data, sqlite3, 384–386
Result data, saving in trace, 1020–1021
Retrieving data, sqlite3, 355–357
return command, pdb, 989
return events, tracing program in sys, 1105–1106
reverse(), pkgutil, 1250
Rich comparison, functools, 138–140
RLock object, threading, 522
rmdir() function, removing directories in os, 1119
rmtree() function, shutil, 277–278
RobotFileParser.can_fetch(), 675–676
robotparser module
defined, 637
long-lived spiders, 676–677
purpose of, 674
reference guide, 677
robots.txt file, 674–675
testing access permissions, 675–676
robots.txt file, 662, 674–677
rollback(), changes to database in sqlite3, 370–371
RotatingFileHandler, logging, 879–880
Rotation
deque, 78–79
log file, 879–880
Rounding, decimal contexts, 202–206
Row objects, sqlite3, 358–359
row_factory property,
Connection objects, 358–359
RSS feed, converting M3U files to, 883–886
ruler attribute, configuring cmd, 847–848
Rules, breakpoint, 998–999
run()
canceling events, sched, 897–898
overlapping events, sched, 896
running profiler in profile, 1023–1026
subclassing Process by overriding, 541
subclassing Thread by overriding, 513
run command, program in pdb, 1009
runctx(), profile, 1026
runfunc() method, trace, 1019
Running external commands, os, 1121–1122
Runtime
changing execution flow in pdb, 1002–1009
environment, sys, 1062–1065
finding message catalogs at, 903–904
garbage collector. See gc module
inspecting stacks and frames at, 1213–1216
interpreter compile-time configuration. See sysconfig module
overview of, 1045–1046
portable access to OS features. See os module
site-wide configuration. See site module
system resource management with resource, 1134–1138
system-specific configuration. See sys module
system version implementation with platform, 1129–1134
RuntimeError exception, 1229–1230
RuntimeWarning, 1233
S
-S option, disabling site, 1054
SafeConfigParser
accessing configuration settings, 864–869
combining values with interpolation, 875–878
modifying configuration settings, 869–871
option search path, 872–875
safe_substitute() method, string.Template, 6–7
sample() function, random, 218–219
Saving
configuration files, 871–872
result data in trace, 1020–1021
state in random, 213–214
sched module
canceling events, 897–898
defined, 770
event priorities, 897
overlapping events, 896–897
purpose of, 894–895
reference guide, 898
running events with delay, 895–896
timed event scheduler, 894–898
Schema
creating embedded relational database, 353
defined, 352
Schemes, sysconfig, 1163
Search criteria, IMAP4 mailbox, 747–748
Search function, adding to registry for encoding, 309–310
search() function, IMAP4, 746–747, 749–752
search() function, re
compiling expressions, 14–15
constraining, 26–30
finding patterns in text, 14
multiple matches, 15–16
Search path
custom importers in sys, 1083–1085
defines in sys, 1081–1084
for options in ConfigParser, 872–875
second attribute
date class, 182–183
time class, 181
Sections, ConfigParser
accessing configuration
settings, 865
defined, 862
option search path, 872–875
removing, 870
testing whether values are
present, 865–867
Security
HMAC authentication for,
476–479
insecurity of pickle, 334
SimpleXMLRPCServer
implications, 715
seed() function, random,
212–213
seek() method
reading compressed data in
gzip, 434
reading compressed files in
bz2, 443
StringIO buffers, 315
temporary files, 267
select() function, select,
594–601
select module
nonblocking I/O with timeouts,
601–603
platform-specific options, 608
purpose of, 594–595
reference guide, 608–609
using poll() function, 603–608
using select() function,
595–601
Self-referencing expressions, re,
50–56
Semaphore
multiprocessing, 548–550
threading, 525–526
send() function
nonblocking communication and
timeouts vs., 593–594
Unicode data and network
communication, 304–305
sendall() function, TCP/IP
socket, 573–574
send_error() method,
BaseHTTPServer, 649–650
send_header() method,
BaseHTTPServer, 650–651
Sending signals, 501
sendmail(), with smtplib,
728–730
Sequence operators, operator
module, 157–158
SequenceMatcher, 65–68
Sequences
comparing lines of text. See
difflib module
of fixed-type numerical data,
84–87
operators for, 157–158
SerialCookie class, deprecated
in Cookie, 683
Serializing
defined, 333
objects. See pickle module
XML to stream in
ElementTree, 408–410
serve_forever(),
SocketServer, 609
ServerProxy
connecting to XML-RPC server,
704–706
SimpleXMLRPCServer, 715–716
Servers
classes implementing SMTP,
734–738
classes implementing Web. See
BaseHTTPServer module
connecting to IMAP, 739–740
connecting to XML-RPC,
709–710
creating network. See
SocketServer module
implementing with asyncchat,
630–632
implementing XML-RPC. See
SimpleXMLRPCServer module
SocketServer, 609–610
TCP/IP, 572–575
UDP, 581–583
using asyncore in, 619–621
Services, socket 566–570
Set-Cookie header, Cookie
module
alternative output formats,
682–683
overview of, 678
receiving and parsing Cookie
headers, 681–682
set() method
modifying configuration settings,
869–871
setting Element properties,
403–405
signaling between threads, 516
setblocking() method,
select, 594
setDaemon() method, daemon
threads, 509
set_debug() function, gc,
1151–1159
setdefault() function,
timeit, 1034
setdefaultencoding() function,
sys, 1058
set_defaults(), optparse,
781–782
setfirstweekday() method,
calendar, 194
setitem() function, sequence
operators, 158
setlocale() function, locale,
909–911
setrecursionlimit() function,
sys, 1067–1068
setrlimit() function,
resource, 1136
setsid() function, signal, 495
setsockopt, TTL multicast
messages, 588, 590
setstate() function, random,
213–214
set_terminator(),
asyncchat, 629–630
set_threshold() function, gc,
1149–1151
set_trace() function, pdb,
977–978, 983–984
settrace() function, sys,
1101–1102
setUp() method
SocketServer, 610
setup() method
unittest, 956–957
setUpStatement() function,
timeit, 1033–1035
SHA-1
calculating in hashlib,
470–471
creating UUID name-based
values, 686–688
vs. MD5 in hmac, 474–475
Shallow argument, `cmp()`, 326
Shallow argument, `cmpfiles()`, 326
Shallow copies, 118–119
Shared-argument definitions, `argparse`, 807–808
Shell commands, running in `cmd`, 848–849
Shell-style syntaxes, parsing. See `shlex` module
shelve module
creating new shelf, 343–344
defined, 333–334
importing module from, 1085–1091
purpose of, 343
reference guide, 346
specific shelf types, 346
writeback, 344–346
ShelveFinder, 1089
ShelveLoader, 1087, 1089, 1091–1093
shlex module
controlling parser, 856–858
defined, 770
embedded comments, 854
error handling, 858–859
including other sources of tokens, 855–856
POSIX vs. non-POSIX parsing, 869–871
purpose of, 852
quoted strings, 852–854
reference guide, 861
split, 855
Short-form options
`argparse`, 797
`getopt`, 771–775
`optparse`, 778–779
shouldtake() function, `itertools`, 149
shove module, 346
show_projects(), `sqlite3`, 368–370
show_results() function, `timeit`, 1033–1035
show_type() function, `xmlrpclib`, 710
showwarning() function, 1175–1176
shuffle() function, `random`, 216–218
Shutdown callbacks, program, 890–894
shutil module
copying file metadata, 274–276
copying files, 271–274
defined, 247
purpose of, 271
reference guide, 278
working with directory trees, 276–278
SIG_DFL value, 499–501
SIG_IGN value, 499–501, 502
SIGINT, 502
Signal handlers
ignoring signals, 502
receiving signals, 498–499
retrieving registered, 499–501
signals and threads, 502
signal module
alarms, 501–502
creating processes with `os.fork()`, 1123
ignoring signals, 502
purpose of, 497–498
receiving signals, 498–499
reference guide, 502
retrieving registered handlers, 499–501
sending signals, 501
signals and threads, 502–505
when callbacks are not invoked, 891
Signaling between processes
multiprocessing, 545–546
subprocess, 492–497
Signaling between threads,
threading, 516–517
signal.pause(), 502
Signals and threads, `signal`, 502–505
Signing messages, `hmac`, 474, 476–479
SIGUSR1, 502
SIGXCPU signal, 1137
simple mail transport protocol (SMTP). See `smtplib` module;
`smtplib` module
SimpleCompleter class,
`readline`, 824–827
SimpleCookie class
alternative output formats, 682–683
creating and setting, 678–679
deprecated classes vs., 683
encoding header, 681
receiving and parsing header, 682
SimpleXMLRPCServer module
alternate API names, 716–717
arbitrary API names, 719
defined, 638
dispatching calls, 722–723
dotted API names, 718–719
exposing methods of objects, 720–721
introspection API, 724–726
purpose of, 714
reference guide, 726
simple server, 714–716
Sine, `math`
hyperbolic functions, 243–244
trigonometric functions, 240–243
Single character wildcard, `glob`, 259–260
site module
customizing site configuration, 1051–1052
customizing user configuration, 1053–1054
defined, 1045
disabling, 1054
import path configuration, 1046–1047
path configuration files, 1049–1051
reference guide, 1054–1055
user directories, 1047–1048
Site-wide configuration. See site module
sitecustomize module, 1051–1052
__sizeof__() method, `sys`, 1067–1068
Sizes distribution, `random`, 223
sleep() call
EXCLUSIVE isolation level in `sqlite3`, 375
interrupted when receiving signals, 499
signals and threads, 504–505
SmartCookie class, deprecated in `Cookie`, 683
smtplib module
debugging server, 737
mail server base class, 734–737
smtpd module (continued)
proxy server, 737–738
purpose of, 734
reference guide, 738
SMTP (simple mail transport
protocol). See smtpd module; smtplib module

smtplib module
authentication and encryption,
730–732
defined, 727
purpose of, 727
reference guide, 733–734
sending email message, 728–730
verifying email address, 732–733
SMTPServer class, 734–736
sniff() method, detecting dialects
in csv, 417–418
Sniffer class, detecting dialects in
csv, 417–418
SOCK_DGRAM socket type, 562
socket class, socket
module, 561
socket module
finding service information,
566–568
IP address representations,
570–571
looking up hosts on network,
563–565
looking up server addresses,
568–570
multicast messages, 587–591
nonblocking communication and
timeouts, 593–594
overview of, 562–563
purpose of, 561
reference guide, 572, 591, 594
sending binary data, 591–593
TCP/IP. See TCP/IP sockets
TCP/IP client and server, 572–580
UDP client and server, 580–583
UNIX domain sockets, 583–587
Socket types, 562
socket.error, 563–565
socketpair() function, UNIX
Domain Sockets, 586–587
SocketServer module
adding threading or forking in
HTTPServer using, 648–649
BaseHTTPServer using
classes from, 644
echo example, 610–615
implementing server, 610
purpose of, 609
reference guide, 618–619
request handlers, 610
server objects, 609
server types, 609
threading and forking, 616–618
SOCK_STREAM socket type, 562
Soft limits, resource, 1136–1137
Sorting
creating UUID objects to handle,
689–690
customizing functions in
sqlite3, 381–383
JSON format, 692–694
maintaining lists in sorted order,
93–96
Source code
byte-compiling with
compileall, 1037–1039
creating message catalogs from,
900–903
retrieving for module from ZIP
archive, 1243–1244
retrieving with inspect,
1207–1208
source property, shlex, 855–856
sourcehook() method,
shlex, 856
spawn() functions, os, 1127
Special constants, math, 223–224
Special functions, math, 244–245
Special values, decimal, 200–201
Specific shelf types, shelve, 346
Spiders, controlling Internet,
674–677
split() function
existing string with shlex, 855
path parsing in os.path, 249
splitting strings with patterns in
re, 58–60
splittext() function, path
parsing in os.path, 250–251
Splitting iterators, itertools, 144–145
Splitting with patterns, re, 58–60
SQL-injection attacks, 359
SQLite, 351
sqlite3 module
bulk loading, 362–363
creating database, 352–355
custom aggregation, 380–381
custom sorting, 381–383
defined, 334
defining new column types,
363–366
determining types for columns,
366–368
exporting database contents,
376–378
isolation levels, 372–376
in-memory databases, 376
purpose of, 351
query metadata, 357–358
querying, 355–357
reference guide, 387
restricting access to data, 384–386
retrieving data, 355–357
row objects, 358–359
threading and connection sharing,
383–384
transactions, 368–371
using Python functions in SQL,
378–380
using variables with queries,
359–362
SQLITE_DENY operations, 386
SQLITE_IGNORE operations,
385–386
SQLITE_READ operations, 384–385
square brackets [], config file, 862
Square roots, computing in math,
234–325
stack() function, inspect,
1214–1215
Stack, inspecting runtime
environment, 1213–1216
Stack levels in warnings, 1176–1177
Stack trace
traceback working with,
963–965
tracing program as it runs,
1105–1106
StandardError class,
exceptions, 1216
starmap() function,
iterertools, 146
start events, ElementTree
parsing, 393–396
“start” input value, readline,
826–827
start() method
custom tree builder in ElementTree, 398
finding patterns in text with re, 14
multiprocessing, 529–530
threading, 505–506
start-na events, ElementTree, 394–396
start-up hook, readline, 834–835
STARTTLS extension, SMTP encryption, 731–732
stdout attribute, runtime environment in sys, 1063–1064
step command, pdb, 984–990
step() method, sqlite3, 380–381
stepping through execution of program, pdb, 984–990
“stop” input value, readline, 826–827
Storage
insecurity of pickle for, 334 of persistent objects. See shelve module
store action
argparse, 799–802
optparse, 784
store_const action
argparse, 799–802
optparse, 785
store_false action, argparse, 799–802
store_true action, argparse, 799–802
StreamReader, custom encoding, 311, 313
Streams
managing child processes in os, 1112–1115
mixed content with bz2, 439–440
mixed content with zlib, 424–425
pickle functions for, 336–338
runtime environment with sys, 1063–1064
working with gzip, 434–436
working with json, 700–701
StreamWriter, custom encoding, 311, 313
strftime() function, time, 179–180
strict mode, codec error handling, 292–293, 295
string module
advanced templates, 7–9
functions, 4–5
overview of, 4
reference guide, 9
templates, 5–7
StringIO buffers
applications of HMAC message signatures, 476–477
defined, 248
strings in GzipFile, 434–436
streams in pickle, 336
text buffers, 314–315
writing data from other sources in tarfile, 455
Strings
argparse treating all argument values as, 817–819
converting between binary data and, 102–106
encoding and decoding. See codecs module
encoding and decoding with pickle, 335–336
modifying with patterns, 56–58
parsing in ElementTree, 398–400
string.Template, 5–9
strftime() function, datetime, 179–180, 190
struct module
buffers, 105–106
data structures, 102–106
endianness, 103–105
functions vs. Struct class, 102
packing and unpacking, 102–103
purpose of, 102
reference guide, 106
sending binary data, 591–593
struct_time() function, time, 176–177, 179–180
sub(), modifying strings with patterns, 56–58
Subclassing
from abstract base class, 1179–1181
processes with multiprocessing, 540–541
reasons to use abstract base classes, 1178
threads with threading, 513–515
subdirs attribute, filecmp, 332
SubElement() function, ElementTree, 400–401
Subfolders, Maildir mailbox, 766–768
Subpatterns, groups containing, 36
subprocess module
connecting segments of pipe, 489–490
subprocess module (continued)
interacting with another command, 490–492
purpose of, 481–482
reference guide, 397
running external command, 482–486
signaling between processes, 492–497
working with pipes directly, 486–489
Substitution errors, ConfigParser, 877
Suites, test
doctest, 943
unittest, 957
unittest integration in doctest, 945
super() function, abc, 1181–1182
Switches, argparse prefixes, 802–803
Switching translations,gettext, 908
Symbolic links, os, 1119
symlink() function, os, 1119
Symlinks
copying directories, 277
functions in os, 1119
Synchronizing
processes with multiprocessing, 547–548
threads with threading, 523–524
SyntaxError exception, 1230
SyntaxWarning, 1233
sys module
defined, 1045
exception handling, 1071–1074
hook for program shutdown, 890
interpreter settings, 1055–1062
low-level thread support, 1074–1080
memory management. See Memory management and limits, sys
purpose of, 1055
reference guide, 1107–1108
runtime environment, 1062–1065
tracing program as it runs, 1101–1107
sys module, modules and imports built-in modules, 1080–1091
custom importers, 1083–1085
custom package importing, 1091–1093
handling import errors, 1094–1095
import path, 1081–1083
imported modules, 1080–1081
importer cache, 1097–1098
importing from shelf, 1085–1091
meta path, 1098–1101
package data, 1095–1097
reference guide, 1101
reloading modules in custom importer, 1093–1094
sys.api_version, 1055–1056
sys.argv, 851–852, 1062–1063
sysconfig module
configuration variables, 1160–1161
defined, 1046
installation paths, 1163–1166
purpose of, 1160
Python version and platform, 1167–1168
reference guide, 1168
sys._current_frames(), 1078–1080
sys.excepthook, 1071–1072
sys.exc_info() function, traceback, 959–961
sys.exit(), 892–893, 1064–1065
sys.flags, interpreter command-line options, 1057–1058
sys.getcheckinterval(), 1074
sys.hexversion, 1055–1056
sys.modules, 1080
sys.path
compiling, 1038–1039
configuring import path with site, 1046–1047
defined, 1080
importer cache, 1097–1098
meta path, 1098–1099
path configuration files, 1049–1051
sys.platform, 1056–1057
sys.setcheckinterval(), 1074
sys.stderr, 837, 959, 1175
sys.stdout, 837, 959
sys.version, 1055–1056
sys.version_info, 1055–1056
Tab completion. See readline module
Tables, embedded relational database, 353–355
“Tails,” picking random items, 216
takewhile() function, filtering iterators, 149–150
Tangent, math, 240–243
Tar archive access. See tarfile module
tarfile module
appending to archives, 455
creating new archives, 453
extracting files from archive, 450–452
purpose of, 448
reading metadata from archive, 449–450
reference guide, 456–457
testing tar files, 448–449
using alternate archive member names, 453–454
working with compressed archives, 456
writing data from sources other than files, 454–455
Target functions, importing in multiprocessing, 530–531
TarInfo objects
creating new archives in tarfile, 453
reading metadata in tarfile, 449
using alternate archive member names, 453–454
writing data from sources other than files, 454–455
TCP/IP sockets
choosing address for listening, 577–580
client and server together, 574–575
easy client connections, 575–577
echo client, 573–574
echo server, 572–573
UNIX Domain Sockets vs., 583–586
using poll(), 603–608
using select(), 598–601
TCP (transmission control protocol),
SOCK_STREAM socket for, 562
TCPServer class,
SocketServer, 609–610
tearDown(), unittest, 956–957
tee() function, itertools, 144–145
tempfile module
defined, 247
named files, 268
predicting names, 269–270
purpose of, 265
reference guide, 271
temporary directories, 268–269
temporary file location, 270–271
temporary files, 265–268
Templates, string, 5–9
Temporary breakpoints, 997–998
Temporary file system objects. See
tempfile module
temporary file location function, 270–271
temporary files, 265–268
Terminal, using getpass() without, 837–838
Terminating processes,
multiprocessing, 536–537
Terminators, asynchat, 632–634
Terse argument, platform() function, 1130–1131
Test context, doctest, 945–948
Test data, linecache, 261–262
__test__, doctest, 937–938
test() method, unittest, 949
testCase. See unittest module
testFail() method, unittest, 951–952
testfile() function, doctest, 944–945, 948
Testing
with automated framework. See
unittest module
in-memory databases for automated, 376
os.path files, 255–256
tar files, 448–449
through documentation. See
doctest module
ZIP files, 457
testmod() function, doctest, 942–943, 948
test_patterns, pattern syntax
anchoring, 24–26
character sets, 20–24
dissecting matches with groups, 30, 34–37
expressing repetition, 18–20
overview of, 16–17
using escape codes, 22–24
Text
command-line completion. See
readline module
comparing sequences. See
difflib module
constants and templates with
string, 4–9
encoding and decoding. See
codecs module
encoding binary data with ASCII. See
base64 module
formatting paragraphs with
textwrap, 9–13
overview of, 3
parsing shell-style syntaxes. See
shlex module
processing files as filters. See
fileinput module
reading efficiently. See
linecache module
regular expressions. See re
module
SQL support for columns, 363–366
StringI0 buffers for, 314–315
textwrap module
combining dedent and fill, 11–12
filling paragraphs, 10
hanging indents, 12–13
overview of, 9–10
reference guide, 13
removing existing indentation, 10–11
Thread-safe FIFO implementation,
Queue, 96–102
Threading
adding to HTTPServer,
648–649
and connection sharing,
sqlite3, 383–384
threading module
controlling access to resources,
517–523
daemon vs. non-daemon threads,
509–511
determining current thread,
507–508
enumerating all threads, 512–513
importable target functions in
multiprocessing, 530–531
isolation levels in sqlite3, 373
limiting concurrent access to
resources, 524–526
multiprocessing basics,
529–530
multiprocessing features
for, 529
purpose of, 505
reference guide, 528
signaling between threads,
516–517
decimal module contexts,
206–207
defined, 505
isolation levels in sqlite3, 372–376
managing processes like. See
multiprocessing module
signals and, 502–505
Threads (continued)
threading module. See threading module using Queue class with multiple, 99–102
Thresholds, gc collection, 1148–1151
time class, datetime, 181–182
time() function, 174–176
time module defined, 173
parsing and formatting times, 179–180
processor clock time, 174–176
purpose of, 173
reference guide, 180
time components, 176–177
wall clock time, 174
working with time zones, 177–179
time-to-live (TTL) value, multicast messages, 588
Time values, 181–182, 184–185
Time zones, 177–179, 190
Timed event scheduler, sched, 894–898
timedelta, datetime, 185–186
timeit module
basic example, 1032
command-line interface, 1035–1036
contents of, 1032
defined, 920
purpose of, 1031–1032
reference guide, 1037
storing values in dictionary, 1033–1035
Timeouts configuring for sockets, 594
nonblocking I/O with, 601–603
using poll(), 604
Timer class. See timeit module
Timer threads, threading, 515–516
Times and dates
calendar module, 191–196
datetime. See datetime module overview of, 173
time. See time module
Timestamps
manipulating date values, 183–184
sqlite3 converters for columns, 364
Timing execution of small bits of code. See timeit module
TLS (transport layer security) encryption, SMTP, 730–732
To headers, smtplib, 728
today() class method, current date, 182
Tokens, shlex, 855–859
toprettyxml() method, pretty-printing XML, 401–403
tostring(), serializing XML to stream, 408
total_ordering(), functools comparison, 138–140
total_seconds() function, timedelta, 184
Trace hooks
exception propagation, 1106–1107
monitoring programs, 1101
tracing function calls, 1102–1103
tracing inside functions, 1103–1104
watching stack, 1105–1106
Tracebacks
defined, 928, 958
detailed reports on. See cgitb module
recognizing with doctest, 928–930
as test outcome in unittest, 951–952
trace_calls() function, sys, 1102–1104
trace_calls_and_returns() function, sys, 1105
trace_lines() function, sys, 1103–1104
Tracing
program flow. See trace module
references with gc, 1138–1141
Tracing program as it runs, sys
exception propagation, 1106–1107
function calls, 1102–1103
inside functions, 1103–1104
overview of, 1101
watching stack, 1105–1106
Transactions, sqlite3, 368–371
translate() function
creating translation tables, 4–5
UNIX-style filename comparisons, 318
Translations
creating tables with maketrans(), 4–5
encoding, 298–300
message. See gettext module
Transmission control protocol (TCP), SOCK_STREAM socket
for, 562
transport layer security (TLS) encryption, SMTP, 730–732
Trash folder model, email, 756–757
Traversing parsed tree,
ElementTree, 388–390
Triangles, math, 240–243
triangular() function, random, 222
Trigonometry
inverse functions, 243
math functions, 240–243
math functions for angles, 238–240
truediv() operator, 156–157
truexc() function, math, 226–227
Index 1299

Truth, unittest, 952–953

truth() function, logical operations, 154

try:except block, sqlite3 transactions, 370–371

TTL (time-to-live) value, multicast messages, 588

tty, using getpass() without terminal, 837–838

Tuple, creating Decimal s from, 198–199

Type checking, operator module, 162–163

Type conversion, optparse, 783

Type parameter, add_argument(), 815–817

TypeError exception, argparse, 818

overview of, 1230–1231

time class, 182

TZ environment variable, time zones, 178

tzinfo class, datetime, 190

tzset() function, time zones, 178

UDP (user datagram protocol)
echo client, 581–582

echo server, 581

overview of, 580–581

sending multicast messages with, 588–591

SOCK_DGRAM socket type for, 562

UDPServer class, SocketServer, 609–610

UDS (UNIX Domain Sockets)

AF_UNIX sockets for, 562

communication between parent/child processes, 586–587

overview of, 583–586

permissions, 586

ugettext program, 901

unalias command, pdb, 1011

uname() function, platform, 1131–1133

UnboundLocalError exception, 1231–1232

undoc_header attribute, cmd, 847–848

ungettext() function, gettext, 905–906, 908

Unicode
codec error handling, 291–295

configuration data in ConfigParser, 863–864
data and network communication, 303–307

encoding translation, 298–300

interpreter settings in sys, 1058–1059

non-Unicode encodings, 300–301

reference guide, 313

searching text using strings, 39–40

standard I/O streams, 295–298

turning on case-insensitive matching, 45

understanding encodings, 285–287

working with files, 287–289

UNICODE regular expression flag, 39–40, 45–50

UnicodeDecodeError, 294–295

UnicodeEncodeError, 292–293, 295–298, 309

UnicodeError exception, 1232

UnicodeWarning, 1233

unified_diff() function, difflib, 64

uniform() function, random, 212

Uniform Resource Name (URN) values. See uuid module

unittest module

almost equal, 954–955

asserting truth, 952–953

basic test structure, 949

defined, 919

integration in doctest, 945

purpose of, 949

reference guide, 958

running tests, 949–950

test fixtures, 956–957

test outcomes, 950–952

test suites, 957

testing equality, 953–954

testing for exceptions, 955–956

Universally unique identifiers (UUID). See also uuid module, 684

UNIX

changing file permissions, 1117–1118

domain sockets, 583–587

filename comparisons, 315–317

filename pattern matching, 257–260

mmap() in Windows vs., 279

programming with signal handlers, 498

UNIX Domain Sockets. See UDS (UNIX Domain Sockets)

UnixDatagramServer class, SocketServer, 609, 610

UnixStreamServer class, SocketServer, 609, 610

unpack_from() method, struct, 105–106

unpack() method, struct, 103

unparsing URLs, urlparse, 641–642

Unpicklable objects, pickle, 340

Unpredictable output, doctest, 924–928

unregister(), using poll(), 606

until command, pdb, 988–989

Unused data_attribute, mixed content streams, 424–425, 440

up (u) command, pdb, 980

update() method

populating empty Counter, 71

updates in hashlib, 472–473

update_empty_counter(), functools, 132–133, 137–138

Uploading files, urllib2, 664–667

Uploading messages, IMAP4, 753–755

url2pathname() function, imaplib, 655–657

urllib2cleanup() method, urllib, 652

urldiff() function, urlparse, 640

urlencode(), urllib, 654–655

urllib2open() method, urllib, 655

urljoin() function, constructing absolute URLs, 642–643

urllib module

defined, 637

encoding arguments, 653–655
urllib module (continued)
paths vs. URLs, 655–657
purpose of, 651
reference guide, 657
simple retrieval with cache, 651–653
using Queue class with multiple threads, 99–102
urllib2 module
adding outgoing headers, 661–662
creating custom protocol handlers, 667–670
defined, 637
encoding arguments, 660–661
HTTP GET, 657–660
HTTP POST, 661
posting form data from request, 663–664
purpose of, 657
reference guide, 670
uploading files, 664–667
urlopen() method, urllib2, 657–659, 661
urlparse() function, 638–640, 641
urlparse module
defined, 637
joining, 642–643
parsing, 638–640
purpose of, 638
reference guide, 643
unparsing, 641–642
urlretrieve() method, urllib, 651–653
URLs
encoding variations safe for, 672–673
manipulating strings. See urllib module
network resource access. See urllib module; urllib2 module
urlsplit() function,
urlparse, 639–640, 641
urlunparse() function, urlparse, 641–642
URN (Uniform Resource Name) values. See uuid module
use_alarm(), signals and threads, 504–505
User datagram protocol. See UDP (user datagram protocol)
USER_BASE directory, site, 1047–1048
usercustomize module, 1053–1054
Username, urlparse, 639
Users, site
customizing configuration, 1053–1054
directories, 1047–1048
USER_SITE path name, site, 1047–1048
UserWarning, 1171–1172, 1233
USR signal, subprocess, 493–498
UTF-8
defined, 284
reference guide, 313
working with files, 287–289
UTF-16
byte ordering, 289–291
defined, 284
working with files, 287–289
UTF-32, 287–291
uuid module
defined, 637–638
purpose of, 684
version 1 values, 684–686
version 4 values, 688–689
versions 3 and 5 values, 686–688
working with UUID objects, 689–690
UUID (universally unique identifiers). See also uuid module
uuid1() function, uuid, 684–686
uuid4() function, generating random values, 688–689
V
date and time. See datetime module
event priority, 897
with interpolation,
ConfigParser, 875–878
optparse options, 781–784
plural, with gettext, 905–907
producing new iterator, 146
special, with Decimal, 200–201
storing in dictionary with timeit, 1033–1035
variable argument lists, argparser, 815–817
Variables
dynamic values with queries through, 359–362
on execution stack with pdb, 981–984
Verbose expression syntax, searching text, 40–44
Verbose option, connecting to XML-RPC server, 704
VERBOSE regular expression flag, 42–50
Verbosity levels, logging, 880–882
Verification, email address, 731–732
verify_request() method, SocketServer, 610
Version
package, 1249–1251
specifying Python, 1167–1168
version, argparser, 799–802, 806–807
tenum, 1250
Von Mises distribution, random, 223
vonmisesvariate() function, random, 223
W
date and time. See datetime module
event priority, 897
with interpolation,
ConfigParser, 875–878
optparse options, 781–784
plural, with gettext, 905–907
producing new iterator, 146
special, with Decimal, 200–201
storing in dictionary with timeit, 1033–1035
variable argument lists, argparser, 815–817
Variables
dynamic values with queries through, 359–362
on execution stack with pdb, 981–984
Verbose expression syntax, searching text, 40–44
Verbose option, connecting to XML-RPC server, 704
VERBOSE regular expression flag, 42–50
Verbosity levels, logging, 880–882
Verification, email address, 731–732
verify_request() method, SocketServer, 610
Version
package, 1249–1251
specifying Python, 1167–1168
version, argparser, 799–802, 806–807
tenum, 1250
Von Mises distribution, random, 223
vonmisesvariate() function, random, 223
wait() function
multiprocessing, 545–546
threading, 516–517
waiting for child processes in os, 1125–1127
waiting for I/O. See select module
waitpid() function, os, 1126
walk() function
directory tree with \texttt{os} 1120–1121
traversing directory tree with \texttt{os.path}, 256–257
Walking directory Tree, \texttt{os}, 1120–1121
Wall clock time, \texttt{time}, 174
\texttt{warn()} function
alternate message delivery for warnings, 1175–1176
generating warnings, 1171–1172
stack levels in warnings, 1177
Warning class, 1233
\texttt{WARNING} level, \texttt{logging}, 881–882
\texttt{warnings} module, 1170–1177
alternate message delivery functions, 1175–1176
categories and filtering, 1170–1176
declared, 1169
exceptions defined for use with, 1233
filtering with patterns, 1172–1174
formatting, 1176
generating warnings, 1171–1172
nonfatal alerts with, 1170–1177
purpose of, 1170
reference guide, 1177
repeated warnings, 1174–1175
stack levels in warnings, 1176–1177
Weak references to objects. See \texttt{weakref} module
\texttt{WeakGraph} class, \texttt{weakref}, 113–114
\texttt{WeakKeyDictionary}, \texttt{weakref}, 115–117
\texttt{weakref} module
 caching objects, 114–117
cyclic references, 109–114
data structures, 106–117
declared, 70
proxies, 108–109
purpose of, 106–107
reference callbacks, 108
reference guide, 117
references, 107
\texttt{WeakValueDictionary}, \texttt{weakref}, 115–117
\texttt{weekheader()} method,
\texttt{Calendar} class, 192
\texttt{weibullvariate()} function,
\texttt{random}, 223
where (w) command, \texttt{pdb}, 979–981, 982
\texttt{whichdb} module, 350–351
whitespace
defined, 930
doctest working with, 930–935
Width argument, \texttt{pprint()}, 126–127
Wildcards, \texttt{glob}, 258–260
Windows
\texttt{mmap()} in UNIX vs., 279
non support for zero-length mapping, 280
with statement
applying local context to block of code with, 204–205
with statement
closing open handles in \texttt{contextlib}, 170
context managers tied to, 163
locks as context manager in \texttt{threading}, 522–523
nesting contexts, 168–169
removing temporary files, 266
\texttt{writeable()} function,
\texttt{asyncore}, 621–623
\texttt{Writable sockets}
\texttt{poll()} function, 606–607
\texttt{select()} function, 597–598
\texttt{write()} method
creating new archives, 460–462
saving configuration files, 871–872
serializing XML to stream in \texttt{ElementTree}, 408–410
\texttt{StringIO} buffers, 314–315
Writeback mode, \texttt{shelve}, 344–346
\texttt{write_history_file()}, \texttt{readline}, 832–834
\texttt{writelines()} method
compressed files in \texttt{BZ2File}, 441–442
compressed files in \texttt{gzip}, 432
\texttt{writepy()} method, Python ZIP archives, 466–467
\texttt{writer()} function
\texttt{csv}, 412–413
isolation levels in \texttt{sqlite3}, 373
\texttt{writerow()} function, \texttt{csv}, 412–413
\texttt{writestr()} method
writing data from sources other than files in \texttt{zipfile}, 463
writing with \texttt{ZipInfo} instance, 463–464
Writing
compressed files in \texttt{bz2}, 440–442
compressed files in \texttt{gzip}, 431–433
\texttt{CSV} files, 412–413
data from sources other than \texttt{tarfile}, 454–455
data from sources other than \texttt{zipfile}, 462–463
memory-mapped file updates, 280–283
with \texttt{ZipInfo} instance, 463–464
\textbf{X}
\texttt{xgettext} program, 900–901
XML manipulation API. See \texttt{ElementTree}
XML-RPC protocol
client library. See \texttt{xmlrpclib} module
declared, 702
implementing server. See \texttt{SimpleXMLRPCServer} module
XML-to-CSV converter, 395–398
\texttt{xmlcharrefreplace} mode,
codec error handling, 292–293
\texttt{xml.dom.minidom} pretty printer
\texttt{XML}, 401–403
\texttt{xml.etree.ElementTree}. See \texttt{ElementTree}
\texttt{XMLID()}, \texttt{ElementTree}, 399–400
\texttt{xmlrpclib} module
binary data, 710–712
combining calls into one message, 712–714
connecting to server, 704–706
data types, 706–709
defined, 638
exception handling, 712
passing objects, 709–710
purpose of, 702–703
reference guide, 714
\texttt{XMLTreeBuilder}, \texttt{ElementTree}, 396–398
Y
year attribute, date class, 182–183
yeardays2calendar() method, Calendar, 192–193

Z
Zero-length mapping, Windows non-support for, 280
ZeroDivisionError exception, 1232–1233
ZIP archives
accessing. See zipfile module loading Python code from. See zipimport module
retrieving package data, 1256–1258
zipfile module
appending to files, 464–465
creating new archives, 460–462
extracting archived files from archive, 459–460
limitations, 467
purpose of, 457
Python ZIP archives, 466–467
reading metadata from archive, 457–459
reference guide, 467
retrieving package data, 1256–1258
testing ZIP files, 457
using alternate archive member names, 462
writing data from sources other than files, 462–463
writing with ZipInfo instance, 463–464
zipimport module
accessing code, 1242–1243
data, 1244–1246
defined, 1235
element, 1240–1241
finding module, 1241–1242
packages, 1244
purpose of, 1240
Python ZIP archives, 466–467
reference guide, 1244–1247
retrieving source code, 1243–1244
zipimporter class, 1240
ZipInfo instance, zipfile, 463–464
zlib module
checksums, 425
compressing networked data, 426–430
compressing new archives in zipfile using, 461–462
incremental compression and decompression, 423–424
mixed content streams, 424–425
purpose of, 421
reference guide, 430
working with data in memory, 422–423
ZlibRequestHandler, 426–430