Supercharged Python
This page intentionally left blank
Supercharged Python

Brian Overland
John Bennett
To my beautiful and brilliant mother, Betty P. M. Overland. . . .
All the world is mad except for me and thee. Stay a little.
—Brian

To my parents, who did so much to shape who I am.
—John
# Contents

**Preface**
- What Makes Python Special? xxiii
- Paths to Learning: Where Do I Start? xxiv
- Clarity and Examples Are Everything xxiv
- Learning Aids: Icons xxv
- What You’ll Learn xxvi
- Have Fun xxvi

**Acknowledgments** xxvii

**About the Authors** xxix

## Chapter 1
**Review of the Fundamentals** 1

1.1 Python Quick Start 1
1.2 Variables and Naming Names 4
1.3 Combined Assignment Operators 4
1.4 Summary of Python Arithmetic Operators 5
1.5 Elementary Data Types: Integer and Floating Point 6
1.6 Basic Input and Output 7
1.7 Function Definitions 9
1.8 The Python “if” Statement 11
1.9 The Python “while” Statement 12
1.10 A Couple of Cool Little Apps 14
1.11 Summary of Python Boolean Operators 15
1.12 Function Arguments and Return Values 16
1.13 The Forward Reference Problem 19
1.14 Python Strings 19
1.15 Python Lists (and a Cool Sorting App) 21
1.16 The “for” Statement and Ranges 23
1.17 Tuples 25
1.18 Dictionaries 26
1.19 Sets 28
1.20 Global and Local Variables 29
Summary 31
Review Questions 31
Suggested Problems 32

Chapter 2  

Advanced String Capabilities 33
2.1 Strings Are Immutable 33
2.2 Numeric Conversions, Including Binary 34
2.3 String Operators (+, =, *, >, etc.) 36
2.4 Indexing and Slicing 39
2.5 Single-Character Functions (Character Codes) 42
2.6 Building Strings Using “join” 44
2.7 Important String Functions 46
2.8 Binary, Hex, and Octal Conversion Functions 47
2.9 Simple Boolean (“is”) Methods 48
2.10 Case Conversion Methods 49
2.11 Search-and-Replace Methods 50
2.12 Breaking Up Input Using “split” 53
2.13 Stripping 54
2.14 Justification Methods 55
Summary 56
Review Questions 57
Suggested Problems 57
Chapter 3  

**Advanced List Capabilities**

3.1 Creating and Using Python Lists 59  
3.2 Copying Lists Versus Copying List Variables 61  
3.3 Indexing 61  
  3.3.1 Positive Indexes 62  
  3.3.2 Negative Indexes 63  
  3.3.3 Generating Index Numbers Using “enumerate” 63  
3.4 Getting Data from Slices 64  
3.5 Assigning into Slices 67  
3.6 List Operators 67  
3.7 Shallow Versus Deep Copying 69  
3.8 List Functions 71  
3.9 List Methods: Modifying a List 73  
3.10 List Methods: Getting Information on Contents 75  
3.11 List Methods: Reorganizing 75  
3.12 Lists as Stacks: RPN Application 78  
3.13 The “reduce” Function 81  
3.14 Lambda Functions 83  
3.15 List Comprehension 84  
3.16 Dictionary and Set Comprehension 87  
3.17 Passing Arguments Through a List 89  
3.18 Multidimensional Lists 90  
  3.18.1 Unbalanced Matrixes 91  
  3.18.2 Creating Arbitrarily Large Matrixes 91  
Summary 93  
Review Questions 93  
Suggested Problems 94

Chapter 4  

**Shortcuts, Command Line, and Packages**

4.1 Overview 95  
4.2 Twenty-Two Programming Shortcuts 95  
  4.2.1 Use Python Line Continuation as Needed 96  
  4.2.2 Use “for” Loops Intelligently 97  
  4.2.3 Understand Combined Operator Assignment (+= etc.) 98
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2.4</td>
<td>Use Multiple Assignment</td>
<td>100</td>
</tr>
<tr>
<td>4.2.5</td>
<td>Use Tuple Assignment</td>
<td>101</td>
</tr>
<tr>
<td>4.2.6</td>
<td>Use Advanced Tuple Assignment</td>
<td>102</td>
</tr>
<tr>
<td>4.2.7</td>
<td>Use List and String “Multiplication”</td>
<td>104</td>
</tr>
<tr>
<td>4.2.8</td>
<td>Return Multiple Values</td>
<td>105</td>
</tr>
<tr>
<td>4.2.9</td>
<td>Use Loops and the “else” Keyword</td>
<td>106</td>
</tr>
<tr>
<td>4.2.10</td>
<td>Take Advantage of Boolean Values and “not”</td>
<td>107</td>
</tr>
<tr>
<td>4.2.11</td>
<td>Treat Strings as Lists of Characters</td>
<td>107</td>
</tr>
<tr>
<td>4.2.12</td>
<td>Eliminate Characters by Using “replace”</td>
<td>108</td>
</tr>
<tr>
<td>4.2.13</td>
<td>Don’t Write Unnecessary Loops</td>
<td>108</td>
</tr>
<tr>
<td>4.2.14</td>
<td>Use Chained Comparisons ((n &lt; x &lt; m))</td>
<td>108</td>
</tr>
<tr>
<td>4.2.15</td>
<td>Simulate “switch” with a Table of Functions</td>
<td>109</td>
</tr>
<tr>
<td>4.2.16</td>
<td>Use the “is” Operator Correctly</td>
<td>110</td>
</tr>
<tr>
<td>4.2.17</td>
<td>Use One-Line “for” Loops</td>
<td>111</td>
</tr>
<tr>
<td>4.2.18</td>
<td>Squeeze Multiple Statements onto a Line</td>
<td>112</td>
</tr>
<tr>
<td>4.2.19</td>
<td>Write One-Line if/then/else Statements</td>
<td>112</td>
</tr>
<tr>
<td>4.2.20</td>
<td>Create Enum Values with “range”</td>
<td>113</td>
</tr>
<tr>
<td>4.2.21</td>
<td>Reduce the Inefficiency of the “print” Function Within IDLE</td>
<td>114</td>
</tr>
<tr>
<td>4.2.22</td>
<td>Place Underscores Inside Large Numbers</td>
<td>115</td>
</tr>
<tr>
<td>4.3</td>
<td>Running Python from the Command Line</td>
<td>115</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Running on a Windows-Based System</td>
<td>115</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Running on a Macintosh System</td>
<td>116</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Using pip or pip3 to Download Packages</td>
<td>117</td>
</tr>
<tr>
<td>4.4</td>
<td>Writing and Using Doc Strings</td>
<td>117</td>
</tr>
<tr>
<td>4.5</td>
<td>Importing Packages</td>
<td>119</td>
</tr>
<tr>
<td>4.6</td>
<td>A Guided Tour of Python Packages</td>
<td>121</td>
</tr>
<tr>
<td>4.7</td>
<td>Functions as First-Class Objects</td>
<td>123</td>
</tr>
<tr>
<td>4.8</td>
<td>Variable-Length Argument Lists</td>
<td>125</td>
</tr>
<tr>
<td>4.8.1</td>
<td>The *args List</td>
<td>125</td>
</tr>
<tr>
<td>4.8.2</td>
<td>The **kwargs List</td>
<td>127</td>
</tr>
<tr>
<td>4.9</td>
<td>Decorators and Function Profilers</td>
<td>128</td>
</tr>
<tr>
<td>4.10</td>
<td>Generators</td>
<td>132</td>
</tr>
<tr>
<td>4.10.1</td>
<td>What’s an Iterator?</td>
<td>132</td>
</tr>
<tr>
<td>4.10.2</td>
<td>Introducing Generators</td>
<td>133</td>
</tr>
<tr>
<td>4.11</td>
<td>Accessing Command-Line Arguments</td>
<td>138</td>
</tr>
<tr>
<td>Summary</td>
<td></td>
<td>141</td>
</tr>
<tr>
<td>Questions for Review</td>
<td></td>
<td>142</td>
</tr>
<tr>
<td>Suggested Problems</td>
<td></td>
<td>142</td>
</tr>
</tbody>
</table>
### Chapter 5

**Formatting Text Precisely**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Formatting with the Percent Sign Operator (%)</td>
<td>145</td>
</tr>
<tr>
<td>5.2 Percent Sign (%) Format Specifiers</td>
<td>147</td>
</tr>
<tr>
<td>5.3 Percent Sign (%) Variable-Length Print Fields</td>
<td>150</td>
</tr>
<tr>
<td>5.4 The Global “format” Function</td>
<td>152</td>
</tr>
<tr>
<td>5.5 Introduction to the “format” Method</td>
<td>156</td>
</tr>
<tr>
<td>5.6 Ordering by Position (Name or Number)</td>
<td>158</td>
</tr>
<tr>
<td>5.7 “Repr” Versus String Conversion</td>
<td>161</td>
</tr>
<tr>
<td>5.8 The “spec” Field of the “format” Function and Method</td>
<td>162</td>
</tr>
<tr>
<td>5.8.1 Print-Field Width</td>
<td>163</td>
</tr>
<tr>
<td>5.8.2 Text Justification: “fill” and “align” Characters</td>
<td>164</td>
</tr>
<tr>
<td>5.8.3 The “sign” Character</td>
<td>166</td>
</tr>
<tr>
<td>5.8.4 The Leading-Zero Character (0)</td>
<td>167</td>
</tr>
<tr>
<td>5.8.5 Thousands Place Separator</td>
<td>168</td>
</tr>
<tr>
<td>5.8.6 Controlling Precision</td>
<td>170</td>
</tr>
<tr>
<td>5.8.7 “Precision” Used with Strings (Truncation)</td>
<td>172</td>
</tr>
<tr>
<td>5.8.8 “Type” Specifiers</td>
<td>173</td>
</tr>
<tr>
<td>5.8.9 Displaying in Binary Radix</td>
<td>174</td>
</tr>
<tr>
<td>5.8.10 Displaying in Octal and Hex Radix</td>
<td>174</td>
</tr>
<tr>
<td>5.8.11 Displaying Percentages</td>
<td>175</td>
</tr>
<tr>
<td>5.8.12 Binary Radix Example</td>
<td>176</td>
</tr>
<tr>
<td>5.9 Variable-Size Fields</td>
<td>176</td>
</tr>
<tr>
<td>Summary</td>
<td>178</td>
</tr>
<tr>
<td>Review Questions</td>
<td>179</td>
</tr>
<tr>
<td>Suggested Problems</td>
<td>179</td>
</tr>
</tbody>
</table>

### Chapter 6

**Regular Expressions, Part I**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1 Introduction to Regular Expressions</td>
<td>181</td>
</tr>
<tr>
<td>6.2 A Practical Example: Phone Numbers</td>
<td>183</td>
</tr>
<tr>
<td>6.3 Refining Matches</td>
<td>185</td>
</tr>
<tr>
<td>6.4 How Regular Expressions Work: Compiling Versus Running</td>
<td>188</td>
</tr>
<tr>
<td>6.5 Ignoring Case, and Other Function Flags</td>
<td>192</td>
</tr>
<tr>
<td>6.6 Regular Expressions: Basic Syntax Summary</td>
<td>193</td>
</tr>
<tr>
<td>6.6.1 Meta Characters</td>
<td>194</td>
</tr>
<tr>
<td>6.6.2 Character Sets</td>
<td>195</td>
</tr>
</tbody>
</table>
Chapter 7

**Regular Expressions, Part II**

7.1 Summary of Advanced RegEx Grammar
7.2 Noncapture Groups
    7.2.1 The Canonical Number Example
    7.2.2 Fixing the Tagging Problem
7.3 Greedy Versus Non-Greedy Matching
7.4 The Look-Ahead Feature
7.5 Checking Multiple Patterns (Look-Ahead)
7.6 Negative Look-Ahead
7.7 Named Groups
7.8 The “re.split” Function
7.9 The Scanner Class and the RPN Project
7.10 RPN: Doing Even More with Scanner

Summary
Review Questions
Suggested Problems

Chapter 8

**Text and Binary Files**

8.1 Two Kinds of Files: Text and Binary
    8.1.1 Text Files
    8.1.2 Binary Files
## 8. Approaches to Binary Files: A Summary

8.2 Approaches to Binary Files: A Summary 247
8.3 The File/Directory System 248
8.4 Handling File-Opening Exceptions 249
8.5 Using the “with” Keyword 252
8.6 Summary of Read/Write Operations 252
8.7 Text File Operations in Depth 254
8.8 Using the File Pointer (“seek”) 257
8.9 Reading Text into the RPN Project 258
  8.9.1 The RPN Interpreter to Date 258
  8.9.2 Reading RPN from a Text File 260
  8.9.3 Adding an Assignment Operator to RPN 262
8.10 Direct Binary Read/Write 268
8.11 Converting Data to Fixed-Length Fields (“struct”) 269
  8.11.1 Writing and Reading One Number at a Time 272
  8.11.2 Writing and Reading Several Numbers at a Time 272
  8.11.3 Writing and Reading a Fixed-Length String 273
  8.11.4 Writing and Reading a Variable-Length String 274
  8.11.5 Writing and Reading Strings and Numerics Together 275
  8.11.6 Low-Level Details: Big Endian Versus Little Endian 276
8.12 Using the Pickling Package 278
8.13 Using the “shelve” Package 280

## Summary

282

## Review Questions

283

## Suggested Problems

283

### Chapter 9

**Classes and Magic Methods**

9.1 Classes and Objects: Basic Syntax 285
9.2 More About Instance Variables 287
9.3 The “__init__” and “__new__” Methods 288
9.4 Classes and the Forward Reference Problem 289
9.5 Methods Generally 290
9.6 Public and Private Variables and Methods 292
9.7 Inheritance 293
9.8 Multiple Inheritance 294
9.9 Magic Methods, Summarized 295
Chapter 10  

Decimal, Money, and Other Classes  

10.1 Overview of Numeric Classes  327  
10.2 Limitations of Floating-Point Format  328  
10.3 Introducing the Decimal Class  329  
10.4 Special Operations on Decimal Objects  332  
10.5 A Decimal Class Application  335  
10.6 Designing a Money Class  336  
10.7 Writing the Basic Money Class (Containment)  337  
10.8 Displaying Money Objects ("__str__", "__repr__")  338  
10.9 Other Monetary Operations  339  
10.10 Demo: A Money Calculator  342  
10.11 Setting the Default Currency  345  
10.12 Money and Inheritance  347  
10.13 The Fraction Class  349  
10.14 The Complex Class  353  
Summary  357  
Review Questions  357  
Suggested Problems  358
Chapter 11  The Random and Math Packages

11.1 Overview of the Random Package 359
11.2 A Tour of Random Functions 360
11.3 Testing Random Behavior 361
11.4 A Random-Integer Game 363
11.5 Creating a Deck Object 365
11.6 Adding Pictograms to the Deck 368
11.7 Charting a Normal Distribution 370
11.8 Writing Your Own Random-Number Generator 374
  11.8.1 Principles of Generating Random Numbers 374
  11.8.2 A Sample Generator 374
11.9 Overview of the Math Package 376
11.10 A Tour of Math Package Functions 376
11.11 Using Special Values (pi) 377
11.12 Trig Functions: Height of a Tree 378
11.13 Logarithms: Number Guessing Revisited 381
  11.13.1 How Logarithms Work 381
  11.13.2 Applying a Logarithm to a Practical Problem 382
Summary 385
Review Questions 385
Suggested Problems 386

Chapter 12  The “numpy” (Numeric Python) Package

12.1 Overview of the “array,” “numpy,” and “matplotlib” Packages 387
  12.1.1 The “array” Package 387
  12.1.2 The “numpy” Package 387
  12.1.3 The “numpy.random” Package 388
  12.1.4 The “matplotlib” Package 388
12.2 Using the “array” Package 388
12.3 Downloading and Importing “numpy” 390
12.4 Introduction to “numpy”: Sum 1 to 1 Million 391
12.5 Creating “numpy” Arrays 392
  12.5.1 The “array” Function (Conversion to an Array) 394
  12.5.2 The “arange” Function 396
Chapter 13

Advanced Uses of “numpy”

13.1 Advanced Math Operations with “numpy”
13.2 Downloading “matplotlib”
13.3 Plotting Lines with “numpy” and “matplotlib”
13.4 Plotting More Than One Line
13.5 Plotting Compound Interest
13.6 Creating Histograms with “matplotlib”
13.7 Circles and the Aspect Ratio
13.8 Creating Pie Charts
13.9 Doing Linear Algebra with “numpy”
  13.9.1 The Dot Product
  13.9.2 The Outer-Product Function
  13.9.3 Other Linear Algebra Functions
13.10 Three-Dimensional Plotting
13.11 “numpy” Financial Applications
13.12 Adjusting Axes with “xticks” and “yticks” 467
13.13 “numpy” Mixed-Data Records 469
13.14 Reading and Writing “numpy” Data from Files 471
Summary 475
Review Questions 475
Suggested Problems 476

Chapter 14  Multiple Modules and the RPN Example 477

14.1 Overview of Modules in Python 477
14.2 Simple Two-Module Example 478
14.3 Variations on the “import” Statement 482
14.4 Using the “__all__” Symbol 484
14.5 Public and Private Module Variables 487
14.6 The Main Module and “__main__” 488
14.7 Gotcha! Problems with Mutual Importing 490
14.8 RPN Example: Breaking into Two Modules 493
14.9 RPN Example: Adding I/O Directives 496
14.10 Further Changes to the RPN Example 499
14.10.1 Adding Line-Number Checking 500
14.10.2 Adding Jump-If-Not-Zero 502
14.10.3 Greater-Than (>) and Get-Random-Number (!) 504
14.11 RPN: Putting It All Together 508
Summary 513
Review Questions 514
Suggested Problems 514

Chapter 15  Getting Financial Data off the Internet 517

15.1 Plan of This Chapter 517
15.2 Introducing the Pandas Package 518
15.3 “stock_load”: A Simple Data Reader 519
15.4 Producing a Simple Stock Chart 521
15.5 Adding a Title and Legend 524
15.6 Writing a “makeplot” Function (Refactoring) 525
Contents

15.7 Graphing Two Stocks Together 527
15.8 Variations: Graphing Other Data 530
15.9 Limiting the Time Period 534
15.10 Split Charts: Subplot the Volume 536
15.11 Adding a Moving-Average Line 538
15.12 Giving Choices to the User 540
Summary 544
Review Questions 545
Suggested Problems 545

Appendix A  Python Operator Precedence Table 547

Appendix B  Built-In Python Functions 549
abs(x) 550
all(iterable) 550
any(iterable) 550
ascii(obj) 551
bin(n) 551
bool(obj) 551
bytes(source, encoding) 552
callable(obj) 552
chr(n) 552
compile(cmd_str, filename, mode_str, flags=0,
       dont_inherit=False, optimize=-1) 553
complex(real=0, imag=0) 553
complex(complex_str) 554
delattr(obj, name_str) 555
dir([obj]) 555
divmod(a, b) 556
enumerate(iterable, start=0) 556
eval(expr_str [, globals [, locals]] ) 557
exec(object [, global [, locals]]) 558
filter(function, iterable) 558
float(xl)) 559
format(obj, [format_spec]) 559
frozenset([iterable]) 560
getattr(obj, name_str [,default]) 560
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>globals()</td>
<td>560</td>
</tr>
<tr>
<td>hasattr(obj, name_str)</td>
<td>561</td>
</tr>
<tr>
<td>hash(obj)</td>
<td>561</td>
</tr>
<tr>
<td>help(obj)</td>
<td>561</td>
</tr>
<tr>
<td>hex(n)</td>
<td>561</td>
</tr>
<tr>
<td>id(obj)</td>
<td>561</td>
</tr>
<tr>
<td>input([prompt_str])</td>
<td>562</td>
</tr>
<tr>
<td>int(x, base=10)</td>
<td>562</td>
</tr>
<tr>
<td>int()</td>
<td>562</td>
</tr>
<tr>
<td>isinstance(obj, class)</td>
<td>562</td>
</tr>
<tr>
<td>issubclass(class1, class2)</td>
<td>563</td>
</tr>
<tr>
<td>iter(obj)</td>
<td>563</td>
</tr>
<tr>
<td>len(sequence)</td>
<td>564</td>
</tr>
<tr>
<td>list([iterable])</td>
<td>564</td>
</tr>
<tr>
<td>locals()</td>
<td>565</td>
</tr>
<tr>
<td>map(function, iterable1 [, iterable2...])</td>
<td>565</td>
</tr>
<tr>
<td>max(arg1 [, arg2]...)</td>
<td>566</td>
</tr>
<tr>
<td>max(iterable)</td>
<td>566</td>
</tr>
<tr>
<td>min(arg1 [, arg2]...)</td>
<td>566</td>
</tr>
<tr>
<td>min(iterable)</td>
<td>567</td>
</tr>
<tr>
<td>oct(n)</td>
<td>567</td>
</tr>
<tr>
<td>open(file_name_str, mode='rt')</td>
<td>567</td>
</tr>
<tr>
<td>ord(char_str)</td>
<td>568</td>
</tr>
<tr>
<td>pow(x, y [, z])</td>
<td>569</td>
</tr>
<tr>
<td>print(objects, sep='', end='\n', file=sys.stdout)</td>
<td>569</td>
</tr>
<tr>
<td>range(n)</td>
<td>570</td>
</tr>
<tr>
<td>range(start, stop [, step])</td>
<td>570</td>
</tr>
<tr>
<td>repr(obj)</td>
<td>570</td>
</tr>
<tr>
<td>reversed(iterable)</td>
<td>571</td>
</tr>
<tr>
<td>round(x [, ndigits])</td>
<td>571</td>
</tr>
<tr>
<td>set([iterable])</td>
<td>572</td>
</tr>
<tr>
<td>setattr(obj, name_str, value)</td>
<td>573</td>
</tr>
<tr>
<td>sorted(iterable [, key] [, reverse])</td>
<td>573</td>
</tr>
<tr>
<td>str(obj=&quot;&quot;)</td>
<td>573</td>
</tr>
<tr>
<td>str(obj=b&quot; [, encoding='utf-8']&quot;)</td>
<td>574</td>
</tr>
<tr>
<td>sum(iterable [, start])</td>
<td>574</td>
</tr>
<tr>
<td>super(type)</td>
<td>575</td>
</tr>
<tr>
<td>tuple([iterable])</td>
<td>575</td>
</tr>
<tr>
<td>type(obj)</td>
<td>575</td>
</tr>
<tr>
<td>zip(*iterables)</td>
<td>575</td>
</tr>
</tbody>
</table>
Appendix C  Set Methods

- set_obj.add(obj)  577
- set_obj.clear()  577
- set_obj.copy()  578
- set_obj.difference(other_set)  578
- set_obj.difference_update(other_set)  578
- set_obj.discard(obj)  579
- set_obj.intersection(other_set)  579
- set_obj.intersection_update(other_set)  579
- set_obj.isdisjoint(other_set)  579
- set_obj.issubset(other_set)  579
- set_obj.issuperset(other_set)  580
- set_obj.pop()  580
- set_obj.remove(obj)  580
- set_obj.symmetric_difference(other_set)  580
- set_obj.symmetric_difference_update(other_set)  581
- set_obj.union(other_set)  581
- set_obj.union_update(other_set)  581

Appendix D  Dictionary Methods

- dict_obj.clear()  583
- dict_obj.copy()  584
- dict_obj.get(key_obj, default_val=None)  584
- dict_obj.items()  585
- dict_obj.keys()  585
- dict_obj.pop(key [, default_value])  585
- dict_obj.popitem()  585
- dict_obj.setdefault(key, default_value=None)  586
- dict_obj.values()  586
- dict_obj.update(sequence)  586

Appendix E  Statement Reference

Variables and Assignments  587
Spacing Issues in Python  589
Alphabetical Statement Reference
  assert Statement  590
  break Statement  591
  class Statement  591
continue Statement  593
def Statement  594
del Statement  594
elif Clause  595
else Clause  595
except Clause  595
for Statement  595
global Statement  596
if Statement  597
import Statement  598
nonlocal Statement  598
pass Statement  599
raise Statement  599
return Statement  599
try Statement  600
while Statement  602
with Statement  602
yield Statement  603

Index  605
This page intentionally left blank
Preface

Books on Python aimed for the absolute beginner have become a cottage industry these days. Everyone and their dog, it seems, wants to chase the Python.

We're a little biased, but one book we especially recommend is Python Without Fear. It takes you by the hand and explains the major features one at a time. But what do you do after you know a little of the language but not enough to call yourself an “expert”? How do you learn enough to get a job or to write major applications?

That's what this book is for: to be the second book you ever buy on Python and possibly the last.

What Makes Python Special?

It’s safe to say that many people are attracted to Python because it looks easier than C++. That may be (at least in the beginning), but underneath this so-called easy language is a tool of great power, with many shortcuts and software libraries called “packages” that—in some cases—do most of the work for you. These let you create some really impressive software, outputting beautiful graphs and manipulating large amounts of data.

For most people, it may take years to learn all the shortcuts and advanced features. This book is written for people who want to get that knowledge now, to get closer to being a Python expert much faster.
Paths to Learning: Where Do I Start?

This book offers different learning paths for different people.

- You’re rusty: If you’ve dabbled in Python but you’re a little rusty, you may want to take a look at Chapter 1, “Review of the Fundamentals.” Otherwise, you may want to skip Chapter 1 or only take a brief look at it.

- You know the basics but are still learning: Start with Chapters 2 and 3, which survey the abilities of strings and lists. This survey includes some advanced abilities of these data structures that people often miss the first time they learn Python.

- Your understanding of Python is strong, but you don’t know everything yet: Start with Chapter 4, which lists 22 programming shortcuts unique to Python, that most people take a long time to fully learn.

- You want to master special features: You can start in an area of specialty. For example, Chapters 5, 6, and 7 deal with text formatting and regular expressions. The two chapters on regular expression syntax, Chapters 6 and 7, start with the basics but then cover the finer points of this pattern-matching technology. Other chapters deal with other specialties. For example, Chapter 8 describes the different ways of handling text and binary files.

- You want to learn advanced math and plotting software: If you want to do plotting, financial, or scientific applications, start with Chapter 12, “The ‘numpy’ (Numeric Python) Package.” This is the basic package that provides an underlying basis for many higher-level capabilities described in Chapters 13 through 15.

Clarity and Examples Are Everything

Even with advanced technology, our emphasis is on clarity, short examples, more clarity, and more examples. We emphasize an interactive approach, especially with the use of the IDLE environment, encouraging you to type in statements and see what they do. Text in bold represents lines for you to type in, or to be added or changed.

```python
>>> print('Hello', 'my', 'world!')
Hello my world!
```

Several of the applications in this book are advanced pieces of software, including a Deck object, a fully functional “RPN” language interpreter, and a multifaceted stock-market program that presents the user with many choices. With these applications, we start with simple examples in the beginning, finally showing all the pieces in context. This approach differs from many
books, which give you dozens of functions all out of order, with no sense of architecture. In this book, architecture is everything.
You can download examples from brianoverland.com/books.

Learning Aids: Icons

This book makes generous use of tables for ease of reference, as well as conceptual art (figures). Our experience is that while poorly conceived figures can be a distraction, the best figures can be invaluable. A picture is worth a thousand words. Sometimes, more.

We also believe that in discussing plotting and graphics software, there’s no substitute for showing all the relevant screen shots.

The book itself uses a few important, typographical devices. There are three special icons used in the text.

Note We sometimes use Notes to point out facts you’ll eventually want to know but that diverge from the main discussion. You might want to skip over Notes the first time you read a section, but it’s a good idea to go back later and read them.

The Key Syntax Icon introduces general syntax displays, into which you supply some or all of the elements. These elements are called “placeholders,” and they appear in italics. Some of the syntax—especially keywords and punctuation—are in bold and intended to be typed in as shown. Finally, square brackets, when not in bold, indicate an optional item. For example:

```
set([iterable])
```

This syntax display implies that `iterable` is an iterable object (such as a list or a generator object) that you supply. And it’s optional.

Square brackets, when in bold, are intended literally, to be typed in as shown. For example:

```
list_name = [obj1, obj2, obj3, ...]
```

Ellipses (...) indicate a language element that can be repeated any number of times.

Performance Tip Performance tips are like Notes in that they constitute a short digression from the rest of the chapter. These tips address the question of how you can improve software performance. If you’re interested in that topic, you’ll want to pay special attention to these notes.
What You’ll Learn

The list of topics in this book that are not in *Python Without Fear* or other “beginner” texts is a long one, but here is a partial list of some of the major areas:

- List, set, and dictionary comprehension.
- Regular expressions and advanced formatting techniques; how to use them in lexical analysis.
- Packages: the use of Python’s advanced numeric and plotting software. Also, special types such as `Decimal` and `Fraction`.
- Mastering all the ways of using binary file operations in Python, as well as text operations.
- How to use multiple modules in Python while avoiding the “gotchas.”
- Fine points of object-oriented programming, especially all the “magic methods,” their quirks, their special features, and their uses.

Have Fun

When you master some or all of the techniques of this book, you should make a delightful discovery: Python often enables you to do a great deal with a relatively small amount of code. That’s why it’s dramatically increasing in popularity every day. Because Python is not just a time-saving device, it’s fun to be able to program this way . . . to see a few lines of code do so much.

We wish you the joy of that discovery.

Register your copy of *Supercharged Python* on the InformIT site for convenient access to updates and/or corrections as they become available. To start the registration process, go to informit.com/register and log in or create an account. Enter the product ISBN (9780135159941) and click Submit. Look on the Registered Products tab for an Access Bonus Content link next to this product, and follow that link to access any available bonus materials. If you would like to be notified of exclusive offers on new editions and updates, please check the box to receive email from us.
Acknowledgments

From Brian

I want to thank my coauthor, John Bennett. This book is the result of close collaboration between the two of us over half a year, in which John was there every step of the way to contribute ideas, content, and sample code, so his presence is there throughout the book. I also want to thank Greg Doench, acquisitions editor, who was a driving force behind the concept, purpose, and marketing of this book.

This book also had a wonderful supporting editorial team, including Rachel Paul and Julie Nahil. But I want to especially thank copy editor Betsy Hardinger, who showed exceptional competence, cooperation, and professionalism in getting the book ready for publication.

From John

I want to thank my coauthor, Brian Overland, for inviting me to join him on this book. This allows me to pass on many of the things I had to work hard to find documentation for or figure out by brute-force experimentation. Hopefully this will save readers a lot of work dealing with the problems I ran into.
This page intentionally left blank
About the Authors

Brian Overland started as a professional programmer back in his twenties, but also worked as a computer science, English, and math tutor. He enjoys picking up new languages, but his specialty is explaining them to others, as well as using programming to do games, puzzles, simulations, and math problems. Now he’s the author of over a dozen books on programming.

In his ten years at Microsoft he was a software tester, programmer/writer, and manager, but his greatest achievement was in presenting Visual Basic 1.0, as lead writer and overall documentation project lead. He believes that project changed the world by getting people to develop for Windows, and one of the keys to its success was showing it could be fun and easy.

He’s also a playwright and actor, which has come in handy as an instructor in online classes. As a novelist, he’s twice been a finalist in the Pacific Northwest Literary Contest but is still looking for a publisher.

John Bennett was a senior software engineer at Proximity Technology, Franklin Electronic Publishing, and Microsoft Corporation. More recently, he’s developed new programming languages using Python as a prototyping tool. He holds nine U.S. patents, and his projects include a handheld spell checker and East Asian handwriting recognition software.
This page intentionally left blank
Master crafters need many things, but, above all, they need to master the tools of the profession. This chapter introduces tools that, even if you’re a fairly experienced Python programmer, you may not have yet learned. These tools will make you more productive as well as increase the efficiency of your programs.

So get ready to learn some new tips and tricks.

### 4.1 Overview

Python is unusually gifted with shortcuts and time-saving programming techniques. This chapter begins with a discussion of twenty-two of these techniques.

Another thing you can do to speed up certain programs is to take advantage of the many packages that are available with Python. Some of these—such as `re` (regular expressions), `sys`, `random`, and `math`—come with the standard Python download, and all you have to do is to include an `import` statement. Other packages can be downloaded quite easily with the right tools.

### 4.2 Twenty-Two Programming Shortcuts

This section lists the most common techniques for shortening and tightening your Python code. Most of these are new in the book, although a few of them have been introduced before and are presented in greater depth here.

- Use Python line continuation as needed.
- Use `for` loops intelligently.
Understand combined operator assignment (+= etc.).
Use multiple assignment.
Use tuple assignment.
Use advanced tuple assignment.
Use list and string “multiplication.”
Return multiple values.
Use loops and the `else` keyword.
Take advantage of Booleans and `not`.
Treat strings as lists of characters.
Eliminate characters by using `replace`.
Don’t write unnecessary loops.
Use chained comparisons.
Simulate “switch” with a table of functions.
Use the `is` operator correctly.
Use one-line `for` loops.
Squeeze multiple statements onto a line.
Write one-line if/then/else statements.
Create Enum values with `range`.
Reduce the inefficiency of the `print` function within IDLE.
Place underscores inside large numbers.

Let’s look at these ideas in detail.

### 4.2.1 Use Python Line Continuation as Needed

In Python, the normal statement terminator is just the end of a physical line (although note the exceptions in Section 3.18). This makes programming easier, because you can naturally assume that statements are one per line.

But what if you need to write a statement longer than one physical line? This dilemma can crop up in a number of ways. For example, you might have a string to print that you can’t fit on one line. You could use literal quotations, but line wraps, in that case, are translated as newlines—something you might
not want. The solution, first of all, is to recognize that literal strings positioned next to other literal strings are automatically concatenated.

```python
>>> my_str = 'I am Hen-er-y the Eighth,' ' I am!
>>> print(my_str)
I am Hen-er-y the Eighth, I am!
```

If these substrings are too long to put on a single physical line, you have a couple of choices. One is to use the line-continuation character, which is a backslash (\).

```python
my_str = 'I am Hen-er-y the Eighth, \ ' I am!
```

Another technique is to observe that any open—and so far unmatched—parenthesis, square bracket, or brace automatically causes continuation onto the next physical line. Consequently, you can enter as long a statement as you want—and you can enter a string of any length you want—without necessarily inserting newlines.

```python
my_str = ('I am Hen-er-y the Eighth, ' 'I am! I am not just any Henry VIII, ' 'I really am!')
```

This statement places all this text in one string. You can likewise use open parentheses with other kinds of statements.

```python
length_of_hypotenuse = ( (side1 * side1 + side2 * side2) ** 0.5 )
```

A statement is not considered complete until all open parentheses [() have been matched by closing parentheses []). The same is true for braces and square brackets. As a result, this statement will automatically continue to the next physical line.

### 4.2.2 Use “for” Loops Intelligently

If you come from the C/C++ world, you may tend to overuse the `range` function to print members of a list. Here’s an example of the C way of writing a `for` loop, using `range` and an indexing operation.

```python
beat_list = ['John', 'Paul', 'George', 'Ringo']
for i in range(len(beat_list)):
    print(beat_list[i])
```
If you ever write code like this, you should try to break the habit as soon as you can. It’s better to print the contents of a list or iterator directly.

```
beat_list = ['John', 'Paul', 'George', 'Ringo']
for guy in beat_list:
    print(guy)
```

Even if you need access to a loop variable, it’s better to use the `enumerate` function to generate such numbers. Here’s an example:

```
beat_list = ['John', 'Paul', 'George', 'Ringo']
for i, name in enumerate(beat_list, 1):
    print(i, '. ', name, sep='')
```

This prints

1. John
2. Paul
3. George
4. Ringo

There are, of course, some cases in which it’s necessary to use indexing. That happens most often when you are trying to change the contents of a list in place.

### 4.2.3 Understand Combined Operator Assignment (+= etc.)

The combined operator-assignment operators are introduced in Chapter 1 and so are reviewed only briefly here. Remember that assignment (=) can be combined with any of the following operators: `+`, `-`, `/`, `//`, `%`, `**`, `&`, `^`, `|`, `<<`, `>>`.

The operators `&`, `|`, and `^` are bitwise “and,” “or,” and “exclusive or,” respectively. The operators `<<` and `>>` perform bit shifts to the left and to the right.

This section covers some finer points of operator-assignment usage. First, any assignment operator has low precedence and is carried out last.

Second, an assignment operator may or may not be in place, depending on whether the type operated on is mutable. *In place* refers to operations that work on existing data in memory rather than creating a completely new object. Such operations are faster and more efficient.

Integers, floating-point numbers, and strings are immutable. Assignment operators, used with these types, do not cause in-place assignment; they instead must produce a completely new object, which is reassigned to the variable. Here’s an example:
s1 = s2 = 'A string.'
s1 += '...with more stuff!'
print('s1:', s1)
print('s2:', s2)

The print function, in this case, produces the following output:

s1: A string...with more stuff!
s2: A string.

When s1 was assigned a new value, it did not change the string data in place; it assigned a whole new string to s1. But s2 is a name that still refers to the original string data. This is why s1 and s2 now contain different strings.

But lists are mutable, and therefore changes to lists can occur in place.

```python
a_list = b_list = [10, 20]
a_list += [30, 40]
print('a_list:', a_list)
print('b_list:', b_list)
```

This code prints

```
a_list: [10, 20, 30, 40]
b_list: [10, 20, 30, 40]
```

In this case, the change was made to the list in place, so there was no need to create a new list and reassign that list to the variable. Therefore, a_list was not assigned to a new list, and b_list, a variable that refers to the same data in memory, reflects the change as well.

In-place operations are almost always more efficient. In the case of lists, Python reserves some extra space to grow when allocating a list in memory, and that in turns permits append operations, as well as +=, to efficiently grow lists. However, occasionally lists exceed the reserved space and must be moved. Such memory management is seamless and has little or no impact on program behavior.

Non-in-place operations are less efficient, because a new object must be created. That’s why it’s advisable to use the join method to grow large strings rather than use the += operator, especially if performance is important. Here’s an example using the join method to create a list and join 26 characters together.

```python
str_list = []
n = ord('a')
for i in range(n, n + 26):
    str_list += chr(i)
alphabet_str = ''.join(str_list)
```
Figures 4.1 and 4.2 illustrate the difference between in-place operations and non-in-place operations. In Figure 4.1, string data seems to be appended onto an existing string, but what the operation really does is to create a new string and then assign it to the variable—which now refers to a different place in memory.

![Figure 4.1. Appending to a string (not in-place)](image)

But in Figure 4.2, list data is appended onto an existing list without the need to create a new list and reassign the variable.

![Figure 4.2. Appending to a list (in-place)](image)

Here’s a summary:

- Combined assignment operators such as `+=` cause in-place changes to data if the object is mutable (such as a list); otherwise, a whole new object is assigned to the variable on the left.

- In-place operations are faster and use space more efficiently, because they do not force creation of a new object. In the case of lists, Python usually allocates extra space so that the list can be grown more efficiently at run time.

### 4.2.4 Use Multiple Assignment

Multiple assignment is one of the most commonly used coding shortcuts in Python. You can, for example, create five different variables at once, assigning them all the same value—in this case, 0:

```python
a = b = c = d = e = 0
```
Consequently, the following returns `True`:

```
a is b
```

This statement would no longer return `True` if either of these variables was later assigned to a different object.

Even though this coding technique may look like it is borrowed from C and C++, you should not assume that Python follows C syntax in most respects. Assignment in Python is a statement and not an expression, as it is in C.

### 4.2.5 Use Tuple Assignment

Multiple assignment is useful when you want to assign a group of variables the same initial value.

But what if you want to assign different values to different variables? For example, suppose you want to assign 1 to `a`, and 0 to `b`. The obvious way to do that is to use the following statements:

```
a = 1
b = 0
```

But through *tuple assignment*, you can combine these into a single statement.

```
a, b = 1, 0
```

In this form of assignment, you have a series of values on one side of the equals sign (`=`) and another on the right. They must match in number, with one exception: You can assign a tuple of any size to a single variable (which itself now represents a tuple as a result of this operation).

```
a = 4, 8, 12   # a is now a tuple containing three values.
```

Tuple assignment can be used to write some passages of code more compactly. Consider how compact a Fibonacci-generating function can be in Python.

```python
def fibo(n):
a, b = 1, 0
while a <= n:
    print(a, end=' ')
    a, b = a + b, a
```

In the last statement, the variable `a` gets a new value: `a + b`; the variable `b` gets a new value—namely, the old value of `a`.
Most programming languages have no way to set $a$ and $b$ simultaneously. Setting the value of $a$ changes what gets put into $b$, and vice versa. So normally, a temporary variable would be required. You could do that in Python, if you wanted to:

```python
temp = a       # Preserve old value of a
a = a + b     # Set new value of a
b = temp      # Set b to old value of a
```

But with tuple assignment, there’s no need for a temporary variable.

```python
a, b = a + b, a
```

Here’s an even simpler example of tuple assignment. Sometimes, it’s useful to swap two values.

```python
x, y = 1, 25
print(x, y)   # prints 1 25
x, y = y, x
print(x, y)   # prints 25 1
```

The interesting part of this example is the statement that performs the swap:

```python
x, y = y, x
```

In another language, such an action would require three separate statements. But Python does not require this, because—as just shown—it can do the swap all at once. Here is what another language would require you to do:

```python
temp = x
x = y
y = temp
```

### 4.2.6 Use Advanced Tuple Assignment

Tuple assignment has some refined features. For example, you can unpack a tuple to assign to multiple variables, as in the following example.

```python
tup = 10, 20, 30
a, b, c = tup
print(a, b, c)   # Produces 10, 20, 30
```

It’s important that the number of input variables on the left matches the size of the tuple on the right. The following statement would produce a runtime error.
tup = 10, 20, 30
a, b = tup       # Error: too many values to unpack

Another technique that’s occasionally useful is creating a tuple that has one element. That would be easy to do with lists.

my_list = [3]

This is a list with one element, 3. But the same approach won’t work with tuples.

my_tup = (3)
print(type(my_tup))

This print statement shows that my_tup, in this case, produced a simple integer.

<class 'int'>

This is not what was wanted in this case. The parentheses were treated as a no-op, as would any number of enclosing parentheses. But the following statement produces a tuple with one element, although, to be fair, a tuple with just one element isn’t used very often.

my_tup = (3,)     # Assign tuple with one member, 3.

The use of an asterisk (*) provides a good deal of additional flexibility with tuple assignment. You can use it to split off parts of a tuple and have one (and only one) variable that becomes the default target for the remaining elements, which are then put into a list. Some examples should make this clear.

a, *b = 2, 4, 6, 8

In this example, a gets the value 2, and b is assigned to a list:

2
[4, 6, 8]

You can place the asterisk next to any variable on the left, but in no case more than one. The variable modified with the asterisk is assigned a list of whatever elements are left over. Here’s an example:

a, *b, c = 10, 20, 30, 40, 50

In this case, a and c refer to 10 and 50, respectively, after this statement is executed, and b is assigned the list [20, 30, 40].

You can, of course, place the asterisk next to a variable at the end.

big, bigger, *many = 100, 200, 300, 400, 500, 600
Printing these variables produces the following:

```python
>>> print(big, bigger, many, sep='\n')
100
200
[300, 400, 500, 600]
```

### 4.2.7 Use List and String “Multiplication”

Serious programs often deal with large data sets—for example, a collection of 10,000 integers all initialized to 0. In languages such as C and Java, the way to do this is to first declare an array with a large dimension.

Because there are no data declarations in Python, the only way to create a large list is to construct it on the right side of an assignment. But constructing a super-long list by hand is impractical. Imagine trying to construct a super-long list this way:

```python
my_list = [0, 0, 0, 0, 0, 0, 0, 0...]
```

As you can imagine, entering 10,000 zeros into program code would be very time-consuming! And it would make your hands ache.

Applying the multiplication operator provides a more practical solution:

```python
my_list = [0] * 10000
```

This example creates a list of 10,000 integers, all initialized to 0.

Such operations are well optimized in Python, so that even in the interactive development environment (IDLE), such interactions are handled quickly.

```python
>>> my_list = [0] * 10000
>>> len(my_list)
10000
```

Note that the integer may be either the left or the right operand in such an expression.

```python
>>> my_list = 1999 * [12]
>>> len(my_list)
1999
```

You can also “multiply” longer lists. For example, the following list is 300 elements long. It consists of the numbers 1, 2, 3, repeated over and over.

```python
>>> trip_list = [1, 2, 3] * 100
>>> len(trip_list)
300
```
The multiplication sign (*) does not work with dictionaries and sets, which require unique keys. But it does work with the string class (str); for example, you can create a string consisting of 40 underscores, which you might use for display purposes:

```python
divider_str = '_' * 40
```

Printing out this string produces the following:

```
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
```

### 4.2.8 Return Multiple Values

You can’t pass a simple variable to a Python function, change the value inside the function, and expect the original variable to reflect the change. Here’s an example:

```python
def double_me(n):
    n *= 2

a = 10
double_me(a)
print(a)        # Value of a did not get doubled!!
```

When `n` is assigned a new value, the association is broken between that variable and the value that was passed. In effect, `n` is a local variable that is now associated with a different place in memory. The variable passed to the function is unaffected.

But you can always use a return value this way:

```python
def double_me(n):
    return n * 2
```

```python
a = 10
a = double_me(a)
print(a)     # Value of a did get doubled
```

Therefore, to get an *out* parameter, just return a value. But what if you want more than one out parameter?

In Python, you can return as many values as you want. For example, the following function performs the quadratic equation by returning two values.

```python
def quad(a, b, c):
    determin = (b * b - 4 * a * c) ** .5
    x1 = (-b + determin) / (2 * a)
```
x2 = (-b - determin) / (2 * a)
return x1, x2

This function has three input arguments and two output variables. In calling the function, it’s important to receive both arguments:

x1, x2 = quad(1, -1, -1)

If you return multiple values to a single variable in this case, that variable will store the values as a tuple. Here’s an example:

```python
>>> x = quad(1, -1, -1)
>>> x
(1.618033988749895, -0.6180339887498949)
```

Note that this feature—returning multiple values—is actually an application of the use of tuples in Python.

### 4.2.9 Use Loops and the “else” Keyword

The `else` keyword is most frequently used in combination with the `if` keyword. But in Python, it can also be used with `try-except` syntax and with loops.

With loops, the `else` clause is executed if the loop has completed without an early exit, such as `break`. This feature applies to both `while` loops and `for` loops.

The following example tries to find an even divisor of \( n \), up to and including the limit, \( \text{max} \). If no such divisor is found, it reports that fact.

```python
def find_divisor(n, max):
    for i in range(2, max + 1):
        if n % i == 0:
            print(i, 'divides evenly into', n)
            break
        else:
            print('No divisor found')
```

Here’s an example:

```python
>>> find_divisor(49, 6)
No divisor found
>>> find_divisor(49, 7)
7 divides evenly into 49
```
4.2 Twenty-Two Programming Shortcuts

4.2.10 Take Advantage of Boolean Values and “not”

Every object in Python evaluates to True or False. For example, every empty collection in Python evaluates to False if tested as a Boolean value; so does the special value None. Here’s one way of testing a string for being length zero:

```python
if len(my_str) == 0:
    break
```

However, you can instead test for an input string this way:

```python
if not s:
    break
```

Here are the general guidelines for Boolean conversions.

- Nonempty collections and nonempty strings evaluate as True; so do nonzero numeric values.
- Zero-length collections and zero-length strings evaluate to False; so does any number equal to 0, as well as the special value None.

4.2.11 Treat Strings as Lists of Characters

When you’re doing complicated operations on individual characters and building a string, it’s sometimes more efficient to build a list of characters (each being a string of length 1) and use list comprehension plus `join` to put it all together.

For example, to test whether a string is a palindrome, it’s useful to omit all punctuation and space characters and convert the rest of the string to either all-uppercase or all-lowercase. List comprehension does this efficiently.

```python
test_str = input('Enter test string: ')  
a_list = [c.upper() for c in test_str if c.isalnum()]  
print(a_list == a_list[::-1])
```

The second line in this example uses list comprehension, which was introduced in Section 3.15, “List Comprehension.”

The third line in this example uses slicing to get the reverse of the list. Now we can test whether `test_str` is a palindrome by comparing it to its own reverse. These three lines of code have to be the shortest possible program for testing whether a string is a palindrome. Talk about compaction!

```python
Enter test string: A man, a plan, a canal, Panama!
True
```
4.2.12 **Eliminate Characters by Using “replace”**

To quickly remove all instances of a particular character from a string, use `replace` and specify the empty string as the replacement.

For example, a code sample in Chapter 10 asks users to enter strings that represent fractions, such as “1/2”. But if the user puts extra spaces in, as in “1   /   2”, this could cause a problem. Here's some code that takes an input string, `s`, and quickly rids it of all spaces wherever they are found (so it goes beyond stripping):

```python
s = s.replace(' ', '')
```

Using similar code, you can quickly get rid of all offending characters or substrings in the same way—but only one at a time. Suppose, however, that you want to get rid of all vowels in one pass. List comprehension, in that case, comes to your aid.

```python
a_list = [c for c in s if c not in 'aeiou']
s = ''.join(a_list)
```

4.2.13 **Don’t Write Unnecessary Loops**

Make sure that you don’t overlook all of Python’s built-in abilities, especially when you’re working with lists and strings. With most computer languages, you’d probably have to write a loop to get the sum of all the numbers in a list. But Python performs summation directly. For example, the following function calculates $1 + 2 + 3 \ldots + N$:

```python
def calc_triangle_num(n):
    return sum(range(n+1))
```

Another way to use the `sum` function is to quickly get the average (the mean) of any list of numbers.

```python
def get_avg(a_list):
    return sum(a_list) / len(a_list)
```

4.2.14 **Use Chained Comparisons ($n < x < m$)**

This is a slick little shortcut that can save you a bit of work now and then, as well as making your code more readable.

It’s common to write `if` conditions such as the following:

```python
if 0 < x and x < 100:
    print('x is in range. ')
```
But in this case, you can save a few keystrokes by instead using this:

```python
if 0 < x < 100:        # Use 'chained' comparisons.
    print('x is in range.')
```

This ability potentially goes further. You can chain together any number of comparisons, and you can include any of the standard comparison operators, including `==`, `<`, `<=`, `>`, and `>=`. The arrows don’t even have to point in the same direction or even be combined in any order! So you can do things like this:

```python
a, b, c = 5, 10, 15
if 0 < a <= c > b > 1:
    print('All these comparisons are true!')
    print('c is equal or greater than all the rest!')
```

You can even use this technique to test a series of variables for equality. Here’s an example:

```python
a = b = c = d = e = 100
if a == b == c == d == e:
    print('All the variables are equal to each other.')
```

For larger data sets, there are ways to achieve these results more efficiently. Any list, no matter how large, can be tested to see whether all the elements are equal this way:

```python
if min(a_list) == max(a_list):
    print('All the elements are equal to each other.')
```

However, when you just want to test a few variables for equality or perform a combination of comparisons on a single line, the techniques shown in this section are a nice convenience with Python. Yay, Python!

### 4.2.15 Simulate “switch” with a Table of Functions

This next technique is nice because it can potentially save a number of lines of code.

Section 15.12 offers the user a menu of choices, prompts for an integer, and then uses that integer to decide which of several functions to call. The obvious way to implement this logic is with a series of `if/elif` statements, because Python has no “switch” statement.

```python
if n == 1:
    do_plot(stockdf)
elif n == 2:
    do_highlow_plot(stockdf)
```
elif n == 3:
    do_volume_subplot(stockdf)
elif n == 4:
    do_movingavg_plot(stockdf)

Code like this is verbose. It will work, but it’s longer than it needs to be. But Python functions are objects, and they can be placed in a list just like any other kind of objects. You can therefore get a reference to one of the functions and call it.

    fn = [do_plot, do_highlow_plot, do_volume_subplot, do_movingavg_plot][n-1]
    fn(stockdf)                 # Call the function

For example, n-1 is evaluated, and if that value is 0 (that is, n is equal to 1), the first function listed, do_plot, is executed.

This code creates a compact version of a C++ switch statement by calling a different function depending on the value of n. (By the way, the value 0 is excluded in this case, because that value is used to exit.)

You can create a more flexible control structure by using a dictionary combined with functions. For example, suppose that “load,” “save,” “update,” and “exit” are all menu functions. We might implement the equivalent of a switch statement this way:

    menu_dict = {'load':load_fn, 'save':save_fn, 'exit':exit_fn, 'update':update_fn}
    (menu_dict[selector])()       # Call the function

Now the appropriate function will be called, depending on the string contained in selector, which presumably contains 'load', 'save', 'update', or 'exit'.

### 4.2.16 Use the “is” Operator Correctly

Python supports both a test-for-equality operator (==) and an \texttt{is} operator. These tests sometimes return the same result, and sometimes they don’t. If two strings have the same value, a test for equality always produces \texttt{True}.

    a = 'cat'
    b = 'cat'
    a == b    # This must produce True.

But the \texttt{is} operator isn’t guaranteed to produce \texttt{True} in string comparisons, and it’s risky to rely upon. A constructed string isn’t guaranteed to
match another string if you use `is` rather than test-for-equality (`==`). For example:

```python
>>> s1 = 'I am what I am and that is all that I am.'
>>> s2 = 'I am what I am' + ' and that is all that I am.'
>>> s1 == s2
True
>>> s1 is s2
False
```

What this example demonstrates is that just because two strings have identical contents does not mean that they correspond to the same object in memory, and therefore the `is` operator produces `False`.

If the `is` operator is unreliable in such cases, why is it in the language at all? The answer is that Python has some unique objects, such as `None`, `True`, and `False`. When you’re certain that you’re comparing a value to a unique object, then the `is` keyword works reliably; moreover, it’s preferable in those situations because such a comparison is more efficient.

```python
a_value = my_function()
if a_value is None:
    # Take special action if None is returned.
```

### 4.2.17 Use One-Line “for” Loops

If a `for` loop is short enough, with only one statement inside the loop (that is, the statement body), you can squeeze the entire `for` loop onto a single physical line.

```python
for var in sequence: statement
```

Not all programmers favor this programming style. However, it’s useful as a way of making your program more compact. For example, the following one-line statement prints all the numbers from 0 to 9:

```python
>>> for i in range(10): print(i, end=' ')  
0 1 2 3 4 5 6 7 8 9
```

Notice that when you’re within IDLE, this `for` loop is like any other: You need to type an extra blank line in order to terminate it.
4.2.18 Squeeze Multiple Statements onto a Line

If you have a lot of statements you want to squeeze onto the same line, you can do it—if you’re determined and the statements are short enough.

The technique is to use a semicolon (;) to separate one statement on a physical line from another. Here’s an example:

```python
>>> for i in range(5): n=i*2; m = 5; print(n+m, end=' ')
5 7 9 11 13
```

You can squeeze other kinds of loops onto a line in this way. Also, you don’t have to use loops but can place any statements on a line that you can manage to fit there.

```python
>>> a = 1; b = 2; c = a + b; print(c)
3
```

At this point, some people may object, “But with those semicolons, this looks like C code!” (Oh, no—anything but that!)

Maybe it does, but it saves space. Keep in mind that the semicolons are statement separators and not terminators, as in the old Pascal language.

4.2.19 Write One-Line if/then/else Statements

This feature is also called an *in line* if conditional. Consider the following if/else statement, which is not uncommon:

```python
turn = 0
...
if turn % 2:
    cell = 'X'
else:
    cell = 'O'
```

The book *Python Without Fear* uses this program logic to help operate a tic-tac-toe game. On alternate turns, the cell to be added was either an “X” or an “O.” The turn counter, advanced by 1 each time, caused a switch back and forth (a *toggle*) between the two players, “X” and “O.”

That book replaced the if/else block just shown with the more compact version:

```python
cell = 'X' if turn % 2 else 'O'
```
true_expr if conditional else false_expr

If the conditional is true, then the true_expr is evaluated and returned; otherwise the false_expr is evaluated and returned.

4.2.20 Create Enum Values with “range”

Many programmers like to use enumerated (or “enum”) types in place of so-called magic numbers. For example, if you have a color_indicator variable, in which the values 1 through 5 represent the values red, green, blue, back, and white, the code becomes more readable if you can use the color names instead of using the literal numbers 1 through 5.

You could make this possible by assigning a number to each variable name.

```python
red = 0
blue = 1
green = 2
black = 3
white = 4
```

This works fine, but it would be nice to find a way to automate this code. There is a simple trick in Python that allows you to do that, creating an enumeration. You can take advantage of multiple assignment along with use of the range function:

```python
red, blue, green, black, white = range(5)
```

The number passed to range in this case is the number of settings. Or, if you want to start the numbering at 1 instead of 0, you can use the following:

```python
red, blue, green, black, white = range(1, 6)
```

Note For more sophisticated control over the creation and specification of enumerated types, you can import and examine the enum package.

```python
import enum
def enum
help(enum)
```

You can find information on this feature at

4.2.21 Reduce the Inefficiency of the “print” Function Within IDLE

Within IDLE, calls to the `print` statement are incredibly slow. If you run programs from within the environment, you can speed up performance dramatically by reducing the number of separate calls to `print`.

For example, suppose you want to print a 40 × 20 block of asterisks (*). The slowest way to do this, by far, is to print each character individually. Within IDLE, this code is painfully slowly.

```python
for i in range(20):
    for j in range(40):
        print('*', end='')
    print()
```

You can get much better performance by printing a full row of asterisks at a time.

```python
row_of_asterisks = '*' * 40
for i in range(20):
    print(row_of_asterisks)
```

But the best performance is achieved by revising the code so that it calls the `print` function only once, after having assembled a large, multiline output string.

```python
row_of_asterisks = '*' * 40
s = ''
for i in range(20):
    s += row_of_asterisks + '
'
print(s)
```

This example can be improved even further by utilizing the string class `join` method. The reason this code is better is that it uses in-place appending of a list rather than appending to a string, which must create a new string each time.

```python
row_of_asterisks = '*' * 40
list_of_str = []
for i in range(20):
    list_of_str.append(row_of_asterisks)
print('
'.join(list_of_str))
```

Better yet, here is a one-line version of the code!

```python
print('
'.join(['*' * 40] * 20))
```
4.2.22 Place Underscores Inside Large Numbers

In programming, you sometimes have to deal with large numeric literals. Here’s an example:

```python
CEO_salary = 1500000
```

Such numbers are difficult to read in programming code. You might like to use commas as separators, but commas are reserved for other purposes, such as creating lists. Fortunately, Python provides another technique: You can use underscores (\_) inside a numeric literal.

```python
CEO_salary = 1_500_000
```

Subject to the following rules, the underscores can be placed anywhere inside the number. The effect is for Python to read the number as if no underscores were present. This technique involves several rules.

- You can’t use two underscores in a row.
- You can’t use a leading or trailing underscore. If you use a leading underscore (as in \_1), the figure is treated as a variable name.
- You can use underscores on either side of a decimal point.

This technique affects only how numbers appear in the code itself and not how anything is printed. To print a number with thousands-place separators, use the `format` function or method as described in Chapter 5, “Formatting Text Precisely.”

4.3 Running Python from the Command Line

If you’ve been running Python programs from within IDLE—either as commands entered one at a time or as scripts—one way to improve execution speed is to run programs from a command line instead; in particular, doing so greatly speeds up the time it takes to execute calls to the `print` function.

Some of the quirks of command-line operation depend on which operating system you’re using. This section covers the two most common operating systems: Windows and Macintosh.

4.3.1 Running on a Windows-Based System

Windows systems, unlike Macintosh, usually do not come with a version of Python 2.0 preloaded, a practice that actually saves you a good deal of fuss as long as you install Python 3 yourself.
To use Python from the command line, first start the DOS Box application, which is present as a major application on all Windows systems. Python should be easily available because it should be placed in a directory that is part of the PATH setting. Checking this setting is easy to do while you’re running a Windows DOS Box.

In Windows, you can also check the PATH setting by opening the Control Panel, choose Systems, and select the Advanced tab. Then click Environment Variables.

You then should be able to run Python programs directly as long as they’re in your PATH. To run a program from the command line, enter `python` and the name of the source file (the main module), including the `.py` extension.

```
python test.py
```

### 4.3.2 Running on a Macintosh System

Macintosh systems often come with a version of Python already installed; unfortunately, on recent systems, the version is Python 2.0 and not Python 3.0.

To determine which version has been installed for command-line use, first bring up the Terminal application on your Macintosh system. You may need to first click the Launchpad icon.

You should find yourself in your default directory, whatever it is. You can determine which command-line version of Python you have by using the following command:

```
python -V
```

If the version of Python is 2.0+, you’ll get a message such as the following:

```
python 2.7.10
```

But if you’ve downloaded some version of Python 3.0, you should have that version of Python loaded as well. However, to run it, you’ll have to use the command `python3` rather than `python`.

If you do have `python3` loaded, you can verify the exact version from the command line as follows:

```
python3 -V
```

```
python 3.7.0
```

For example, if the file `test.py` is in the current directory, and you want to compile it as a Python 3.0 program, then use the following command:

```
python3 test.py
```

The Python command (whether `python` or `python3`) has some useful variations. If you enter it with `-h`, the “help” flag, you get a printout on all the
possible flags that you can use with the command, as well as relevant environment variables.

```
python3 -h
```

### 4.3.3 Using `pip` or `pip3` to Download Packages

Some of the packages in this book require that you download and install the packages from the Internet before you use those packages. The first chapter that requires that is Chapter 12, which introduces the `numpy` package.

All the packages mentioned in this book are completely free of charge (as most packages for Python are). Even better, the `pip` utility—which is included with the Python 3 download—goes out and finds the package that you name; thus all you should need is an Internet connection!

On Windows-based systems, use the following command to download and install a desired package.

```
pip install package_name
```

The package name, incidentally, uses no file extension:

```
pip install numpy
```

On Macintosh systems, you may need to use the `pip3` utility, which is downloaded with Python 3 when you install it on your computer. (You may also have inherited a version of pip, but it will likely be out-of-date and unusable.)

```
pip3 install package_name
```

### 4.4 Writing and Using Doc Strings

Python doc strings enable you to leverage the work you do writing comments to get free online help. That help is then available to you while running IDLE, as well as from the command line, when you use the `pydoc` utility.

You can write doc strings for both functions and classes. Although this book has not yet introduced how to write classes, the principles are the same. Here’s an example with a function, showcasing a doc string.

```python
def quad(a, b, c):
    '''Quadratic Formula function.
    This function applies the Quadratic Formula to determine the roots of x in a quadratic equation of the form ax^2 + bx + c = 0.
    '''
```
determin = (b * b - 4 * a * c) ** 0.5
x1 = (-b + determin) / (2 * a)
x2 = (-b - determin) / (2 * a)
return x1, x2

When this doc string is entered in a function definition, you can get help from within IDLE:

```python
>>> help(quad)
Help on function quad in module __main__:

quad(a, b, c)
   Quadratic Formula function.

   This function applies the Quadratic Formula
to determine the roots of x in a quadratic
equation of the form ax^2 + bx + c = 0.
```

The mechanics of writing a doc string follow a number of rules.

- The doc string itself must immediately follow the heading of the function.
- It must be a literal string utilizing the triple-quote feature. (You can actually use any style quote, but you need a literal quotation if you want to span multiple lines.)
- The doc string must also be aligned with the “level-1” indentation under the function heading: For example, if the statements immediately under the function heading are indented four spaces, then the beginning of the doc string must also be indented four spaces.
- Subsequent lines of the doc string may be indented as you choose, because the string is a literal string. You can place the subsequent lines flush left or continue the indentation you began with the doc string. In either case, Python online help will line up the text in a helpful way.

This last point needs some clarification. The doc string shown in the previous example could have been written this way:

```python
def quad(a, b, c):
    '''Quadratic Formula function.

    This function applies the Quadratic Formula
to determine the roots of x in a quadratic
equation of the form ax^2 + bx + c = 0.'''
```
determin = (b ** 2 - 4 * a * c) ** .5
x1 = (-b + determin) / (2 * a)
x2 = (-b - determin) / (2 * a)
return x1, x2

You might expect this doc string to produce the desired behavior—to print help text that lines up—and you’d be right. But you can also put in extra spaces so that the lines also align within program code. It might seem this shouldn’t work, but it does.

For stylistic reasons, programmers are encouraged to write the doc string this way, in which the subsequent lines in the quote line up with the beginning of the quoted string instead of starting flush left in column 1:

```python
def quad(a, b, c):
    '''Quadratic Formula function.

    This function applies the Quadratic Formula to determine the roots of x in a quadratic equation of the form ax^2 + bx + c = 0.
    '''
```

As part of the stylistic guidelines, it’s recommended that you put in a brief summary of the function, followed by a blank line, followed by more detailed description.

When running Python from the command line, you can use the `pydoc` utility to get this same online help shown earlier. For example, you could get help on the module named `queens.py`. The `pydoc` utility responds by printing a help summary for every function. Note that “py” is not entered as part of the module name in this case.

```
python -m pydoc queens
```

### 4.5 Importing Packages

Later sections in this chapter, as well as later chapters in the book, make use of packages to extend the capabilities of the Python language.

A package is essentially a software library of objects and functions that perform services. Packages come in two varieties:

- Packages included with the Python download itself. This includes `math`, `random`, `sys`, `os`, `time`, `datetime`, and `os.path`. These packages are especially convenient, because no additional downloading is necessary.
- Packages you can download from the Internet.
The syntax shown here is the recommended way to an import a package. There are a few variations on this syntax, as we’ll show later.

```
import package_name
```

For example:

```
import math
```

Once a package is imported, you can, within IDLE, get help on its contents. Here’s an example:

```
>>> import math
>>> help(math)
```

If you type these commands from within IDLE, you’ll see that the math package supports a great many functions.

But with this approach, each of the functions needs to be qualified using the dot (.) syntax. For example, one of the functions supported is `sqrt` (square root), which takes an integer or floating-point input.

```
>>> math.sqrt(2)
1.4142135623730951
```

You can use the `math` package, if you choose, to calculate the value of `pi`. However, the math package also provides this number directly.

```
>>> math.atan(1) * 4
3.141592653589793
>>> math.pi
3.141592653589793
```

Let’s look at one of the variations on the `import` statement.

```
import package_name [as new_name]
```

In this syntax, the brackets indicate that the `as new_name` clause is optional. You can use it, if you choose, to give the package another name, or alias, that is referred to in your source file.

This feature provides short names if the full package name is long. For example, Chapter 13 introduces the `matplotlib.pyplot` package.

```
import matplotlib.pyplot as plt
```

Now, do you want to use the prefix `matplotlib.pyplot`, or do you want to prefix a function name with `plt`? Good. We thought so.

Python supports other forms of syntax for the `import` statement. With both of these approaches, the need to use the package name and the dot syntax is removed.
from package_name import symbol_name
from package_name import *

In the first form of this syntax, only the symbol_name gets imported, and not the rest of the package. But the specified symbol (such as pi in this next example) can then be referred to without qualification.

```python
>>> from math import pi
>>> print(pi)
3.141592653589793
```

This approach imports only one symbol—or a series of symbols separated by commas—but it enables the symbolic name to be used more directly. To import an entire package, while also gaining the ability to refer to all its objects and functions directly, use the last form of the syntax, which includes an asterisk (*).

```python
>>> from math import *
>>> print(pi)
3.141592653589793
>>> print(sqrt(2))
1.4142135623730951
```

The drawback of using this version of import is that with very large and complex programs, it gets difficult to keep track of all the names you're using, and when you import packages without requiring a package-name qualifier, name conflicts can arise.

So, unless you know what you're doing or are importing a really small package, it's more advisable to import specific symbols than use the asterisk (*).

### 4.6 A Guided Tour of Python Packages

Thousands of other packages are available if you go to python.org, and they are all free to use. The group of packages in Table 4.1 is among the most useful of all packages available for use with Python, so you should be sure to look them over.

The re, math, random, array, decimal, and fractions packages are all included with the standard Python 3 download, so you don't need to download them separately.

The numpy, matplotlib, and pandas packages need to be installed separately by using the pip or pip3 utility. Later chapters, starting with Chapter 12, cover those utilities in depth.
### Table 4.1. Python Packages Covered in This Book

<table>
<thead>
<tr>
<th>NAME TO IMPORT</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>re</td>
<td>Regular-expression package. This package lets you create text patterns that can match many different words, phrases, or sentences. This pattern-specification language can do sophisticated searches with high efficiency. This package is so important that it’s explored in both Chapters 6 and 7.</td>
</tr>
<tr>
<td>math</td>
<td>Math package. Contains helpful and standard math functions so that you don’t have to write them yourself. These include trigonometric, hyperbolic, exponential, and logarithmic functions, as well as the constants ( e ) and ( \pi ). This package is explored in Chapter 11.</td>
</tr>
<tr>
<td>random</td>
<td>A set of functions for producing pseudo-random values. Pseudo-random numbers behave as if random—meaning, among other things, it’s a practical impossibility for a user to predict them. This random-number generation package includes the ability to produce random integers from a requested range, as well as floating-point numbers and normal distributions. The latter cluster around a mean value to form a “bell curve” of frequencies. This package is explored in Chapter 11.</td>
</tr>
<tr>
<td>decimal</td>
<td>This package supports the ( \text{Decimal} ) data type, which (unlike the ( \text{float} ) type) enables you to represent dollars-and-cents figures precisely without any possibility of rounding errors. ( \text{Decimal} ) is often preferred for use in accounting and financial applications. This package is explored in Chapter 10.</td>
</tr>
<tr>
<td>fractions</td>
<td>This package supports the ( \text{Fraction} ) data type, which stores any fractional number with absolute precision, provided it can be represented as the ratio of two integers. So, for example, this data type can represent the ratio 1/3 absolutely, something that neither the ( \text{float} ) nor ( \text{Decimal} ) type can do without rounding errors. This package is explored in Chapter 10.</td>
</tr>
<tr>
<td>array</td>
<td>This package supports the ( \text{array} ) class, which differs from lists in that it holds raw data in contiguous storage. This isn’t always faster, but sometimes it’s necessary to pack your data into contiguous storage so as to interact with other processes. However, the benefits of this package are far exceeded by the ( \text{numpy} ) package, which gives you the same ability, but much more. This package is briefly covered in Chapter 12.</td>
</tr>
<tr>
<td>numpy</td>
<td>This package supports the ( \text{numpy} ) (numeric Python) class, which in turn supports high-speed batch operations on one-, two-, and higher-dimensional arrays. The class is useful not only in itself, as a way of supercharging programs that handle large amounts of data, but also as the basis for work with other classes. This package is explored in Chapters 12 and 13. ( \text{numpy} ) needs to be installed with ( \text{pip} ) or ( \text{pip3} ).</td>
</tr>
</tbody>
</table>
### 4.7 Functions as First-Class Objects

Another productivity tool—which may be useful in debugging, profiling, and related tasks—is to treat Python functions as *first-class* objects. That means taking advantage of how you can get information about a function at runtime. For example, suppose you’ve defined a function called `avg`.

```python
def avg(a_list):
    '''This function finds the average val in a list.''
    x = (sum(a_list) / len(a_list))
    print('The average is:', x)
    return x
```

The name `avg` is a symbolic name that refers to a function, which in Python lingo is also a callable. There are a number of things you can do with `avg`, such as verify its type, which is `function`. Here’s an example:

```python
>>> type(avg)
<class 'function'>
```

We already know that `avg` names a function, so this is not new information. But one of the interesting things you can do with an object is assign it to a

<table>
<thead>
<tr>
<th>Name To Import</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>numpy.random</code></td>
<td>Similar to <code>random</code>, but designed especially for use with <code>numpy</code>, and ideally suited to situations in which you need to generate a large quantity of random numbers quickly. In head-to-head tests with the standard <code>random</code> class, the <code>numpy</code> random class is several times faster when you need to create an array of such numbers. This package is also explored in Chapter 12.</td>
</tr>
<tr>
<td><code>matplotlib.pyplot</code></td>
<td>This package supports sophisticated plotting routines for Python. Using these routines, you can create beautiful looking charts and figures—even three-dimensional ones. This package is explored in Chapter 13. It needs to be installed with <code>pip</code> or <code>pip3</code>.</td>
</tr>
<tr>
<td><code>pandas</code></td>
<td>This package supports data frames, which are tables that can hold a variety of information, as well as routines for going out and grabbing information from the Internet and loading it. Such information can then be combined with the <code>numpy</code> and plotting routines to create impressive-looking graphs. This package is explored in Chapter 15. It also needs to be downloaded.</td>
</tr>
</tbody>
</table>
new name. You can also assign a different function altogether to the symbolic name, \texttt{avg}.

```python
def new_func(a_list):
    return (sum(a_list) / len(a_list))
```

```text
old_avg = avg
avg = new_func
```

The symbolic name \texttt{old_avg} now refers to the older, and longer, function we defined before. The symbolic name \texttt{avg} now refers to the newer function just defined.

The name \texttt{old_avg} now refers to our first averaging function, and we can call it, just as we used to call \texttt{avg}.

```python
>>> old_avg([4, 6])
The average is 5.0
5.0
```

The next function shown (which we might loosely term a “metafunction,” although it’s really quite ordinary) prints information about another function—specifically, the function argument passed to it.

```python
def func_info(func):
    print('Function name:', func._name_)
    print('Function documentation: help(func)')
```

If we run this function on \texttt{old_avg}, which has been assigned to our first averaging function at the beginning of this section, we get this result:

```text
Function name: avg
Function documentation:
Help on function avg in module __main__:

    avg(a_list)
    This function finds the average val in a list.
```

We’re currently using the symbolic name \texttt{old_avg} to refer to the first function that was defined in this section. Notice that when we get the function’s name, the information printed uses the name that \textit{the function was originally defined with}.

All of these operations will become important when we get to the topic of “decorating” in Section 4.9, “Decorators and Function Profilers.”
4.8 Variable-Length Argument Lists

One of the most versatile features of Python is the ability to access variable-length argument lists. With this capability, your functions can, if you choose, handle any number of arguments—much as the built-in `print` function does.

The variable-length argument ability extends to the use of named arguments, also called “keyword arguments.”

4.8.1 The *args List

The *args syntax can be used to access argument lists of any length.

```python
def func_name([ordinary_args,] *args):
    statements
```

The brackets are used in this case to show that *args may optionally be preceded by any number of ordinary positional arguments, represented here as `ordinary_args`. The use of such arguments is always optional.

In this syntax, the name `args` can actually be any symbolic name you want. By convention, Python programs use the name `args` for this purpose.

The symbolic name `args` is then interpreted as a Python list like any other; you expand it by indexing it or using it in a `for` loop. You can also take its length as needed. Here’s an example:

```python
def my_var_func(*args):
    print('The number of args is', len(args))
    for item in args:
        print(items)
```

This function, `my_var_func`, can be used with argument lists of any length.

```python
>>> my_var_func(10, 20, 30, 40)
The number of args is 4
10
20
30
40
```

A more useful function would be one that took any number of numeric arguments and returned the average. Here’s an easy way to write that function.

```python
def avg(*args):
    return sum(args)/len(args)
```
Now we can call the function with a different number of arguments each time.

```python
>>> avg(11, 22, 33)
22.0
>>> avg(1, 2)
1.5
```

The advantage of writing the function this way is that no brackets are needed when you call this function. The arguments are interpreted as if they were elements of a list, but you pass these arguments without list syntax.

What about the ordinary arguments we mentioned earlier? Additional arguments, not included in the list `*args`, must either precede `*args` in the argument list or be keyword arguments.

For example, let’s revisit the `avg` example. Suppose we want a separate argument that specifies what units we’re using. Because `units` is not a keyword argument, it must appear at the beginning of the list, in front of `*args`.

```python
def avg(units, *args):
    print((sum(args)/len(args)), units)

Here’s a sample use:

```python
>>> avg('inches', 11, 22, 33)
22.0 inches
```

This function is valid because the ordinary argument, `units`, precedes the argument list, `*args`.

**Note** The asterisk (*) has a number of uses in Python. In this context, it’s called the *splat* or the *positional expansion* operator. Its basic use is to represent an “unpacked list”; more specifically, it replaces a list with a simple sequence of separate items.

The limitation on such an entity as `*args` is that there isn’t much you can do with it. One thing you can do (which will be important in Section 4.9, “Decorators and Function Profilers”) is pass it along to a function. Here’s an example:

```python
>>> ls = [1, 2, 3]  # Unpacked list.
>>> print(*ls)      # Print unpacked version
 1 2 3
>>> print(ls)       # Print packed (ordinary list).
[1, 2, 3]
```
4.8 Variable-Length Argument Lists

The other thing you can do with \*args or \*ls is to pack it (or rather, repack it) into a standard Python list; you do that by dropping the asterisk. At that point, it can be manipulated with all the standard list-handling abilities in Python.

4.8.2 The \"**kwargs\" List

The more complete syntax supports keyword arguments, which are named arguments during a function call. For example, in the following call to the \texttt{print} function, the \texttt{end} and \texttt{sep} arguments are named.

\begin{verbatim}
print(10, 20, 30, end='.', sep=',')
\end{verbatim}

The more complete function syntax recognizes both unnamed and named arguments.

\begin{verbatim}
def func_name([ordinary_args], \*args, **kwargs):
    statements
\end{verbatim}

As with the symbolic name \texttt{args}, the symbolic name \texttt{kwargs} can actually be any name, but by convention, Python programmers use \texttt{kwargs}.

Within the function definition, \texttt{kwargs} refers to a dictionary in which each key-value pair is a string containing a named argument (as the key) and a value, which is the argument value passed.

An example should clarify. Assume you define a function as follows:

\begin{verbatim}
def pr_named_vals(**kwargs):
    for k in kwargs:
        print(k, ':', kwargs[k])
\end{verbatim}

This function cycles through the dictionary represented by \texttt{kwargs}, printing both the key values (corresponding to argument names) and the corresponding values, which have been passed to the arguments.

For example:

\begin{verbatim}
>>> pr_named_vals(a=10, b=20, c=30)
a : 10
b : 20
c : 30
\end{verbatim}

A function definition may combine any number of named arguments, referred to by \texttt{kwargs}, with any number of arguments that are not named, referred to by \texttt{args}. Here is a function definition that does exactly that.
The following example defines such a function and then calls it.

```python
def pr_vals_2(*args, **kwargs):
    for i in args:
        print(i)
    for k in kwargs:
        print(k, ':', kwargs[k])

pr_vals_2(1, 2, 3, -4, a=100, b=200)
```

This miniprogram, when run as a script, prints the following:

1
2
3
-4
a : 100
b : 200

**Note** Although `args` and `kwargs` are expanded into a list and a dictionary, respectively, these symbols can be passed along to another function, as shown in the next section.

**4.9 Decorators and Function Profilers**

When you start refining your Python programs, one of the most useful things to do is to time how fast individual functions run. You might want to know how many seconds and fractions of a second elapse while your program executes a function generating a thousand random numbers.

Decorated functions can profile the speed of your code, as well as provide other information, because functions are first-class objects. Central to the concept of decoration is a *wrapper* function, which does everything the original function does but also adds other statements to be executed.

Here’s an example, illustrated by Figure 4.3. The decorator takes a function F1 as input and returns another function, F2, as output. This second function, F2, is produced by including a call to F1 but adding other statements as well. F2 is a wrapper function.
Here's an example of a decorator function that takes a function as argument and wraps it by adding calls to the `time.time` function. Note that `time` is a package, and it must be imported before `time.time` is called.

```python
import time
def make_timer(func):
    def wrapper():
        t1 = time.time()
        ret_val = func()
        t2 = time.time()
        print('Time elapsed was', t2 - t1)
        return ret_val
    return wrapper
```

There are several functions involved with this simple example (which, by the way, is not yet complete!), so let's review.

- There is a function to be given as input; let's call this the original function (`F1` in this case). We'd like to be able to input any function we want, and have it decorated—that is, acquire some additional statements.
- The wrapper function is the result of adding these additional statements to the original function. In this case, these added statements report the number of seconds the original function took to execute.
- The decorator is the function that performs the work of creating the wrapper function and returning it. The decorator is able to do this because it internally uses the `def` keyword to define a new function.
Ultimately, the wrapped version is intended to replace the original version, as you'll see in this section. This is done by reassigning the function name.

If you look at thisdecorator function, you should notice it has an important omission: The arguments to the original function, `func`, are ignored. The wrapper function, as a result, will not correctly call `func` if arguments are involved.

The solution involves the `*args` and `**kwargs` language features, introduced in the previous section. Here's the full decorator:

```python
import time

def make_timer(func):
    def wrapper(*args, **kwargs):
        t1 = time.time()
        ret_val = func(*args, **kwargs)
        t2 = time.time()
        print('Time elapsed was', t2 - t1)
        return ret_val
    return wrapper
```

The new function, remember, will be `wrapper`. It is `wrapper` (or rather, the function temporarily named `wrapper`) that will eventually be called in place of `func`; this wrapper function therefore must be able to take any number of arguments, including any number of keyword arguments. The correct action is to pass along all these arguments to the original function, `func`. Here's how:

```python
ret_val = func(*args, **kwargs)
```

Returning a value is also handled here; the wrapper returns the same value as `func`, as it should. What if `func` returns no value? That's not a problem, because Python functions return `None` by default. So the value `None`, in that case, is simply passed along. (You don’t have to test for the existence of a return value; there always is one!)

Having defined this decorator, `make_timer`, we can take any function and produce a wrapped version of it. Then—and this is almost the final trick—we reassign the function name so that it refers to the wrapped version of the function.

```python
def count_nums(n):
    for i in range(n):
        for j in range(1000):
            pass

count_nums = make_timer(count_nums)
```
The wrapper function produced by `make_timer` is defined as follows (except that the identifier `func` will be reassigned, as you’ll see in a moment).

```python
def wrapper(*args, **kwargs):
    t1 = time.time()
    ret_val = func(*args, **kwargs)
    t2 = time.time()
    print('Time elapsed was', t2 - t1)
    return ret_val
```

We now reassign the name `count_nums` so that it refers to *this* function—`wrapper`—which will call the original `count_nums` function but also does other things.

Confused yet? Admittedly, it’s a brain twister at first. But all that’s going on is that (1) a more elaborate version of the original function is being created at run time, and (2) this more elaborate version is what the name, `count_nums`, will hereafter refer to. Python symbols can refer to any object, including functions (callable objects). Therefore, we can reassign function names all we want.

```python
count_nums = wrapper
```

Or, more accurately,

```python
count_nums = make_timer(count_nums)
```

So now, when you run `count_nums` (which now refers to the wrapped version of the function), you’ll get output like this, reporting execution time in seconds.

```bash
>>> count_nums(33000)
Time elapsed was 1.063697338104248
```

The original version of `count_nums` did nothing except do some counting; this wrapped version reports the passage of time in addition to calling the original version of `count_nums`.

As a final step, Python provides a small but convenient bit of syntax to automate the reassignment of the function name.

```python
@decorator
def func(args):
    statements
```

This syntax is translated into the following:

```python
def func(args):
    statements
    func = decorator(func)
```
In either case, it’s assumed that *decorator* is a function that has already been defined. This decorator must take a function as its argument and return a wrapped version of the function. Assuming all this has been done correctly, here’s a complete example utilizing the `@` sign.

```python
@make_timer
def count_nums(n):
    for i in range(n):
        for j in range(1000):
            pass
```

After this definition is executed by Python, `count_num` can then be called, and it will execute `count_num` as defined, but it will also add (as part of the wrapper) a `print` statement telling the number of elapsed seconds.

Remember that this part of the trick (the final trick, actually) is to get the name `count_nums` to refer to the new version of `count_nums`, after the new statements have been added through the process of decoration.

## 4.10 Generators

There’s no subject in Python about which more confusion abounds than generators. It’s not a difficult feature once you understand it. Explaining it’s the hard part.

But first, what does a generator do? The answer: It enables you to deal with a sequence one element at a time.

Suppose you need to deal with a sequence of elements that would take a long time to produce if you had to store it all in memory at the same time. For example, you want to examine all the Fibonacci numbers up to 10 to the 50th power. It would take a lot of time and space to calculate the entire sequence. Or you may want to deal with an infinite sequence, such as all even numbers.

The advantage of a generator is that it enables you to deal with one member of a sequence at a time. This creates a kind of “virtual sequence.”

### 4.10.1 What’s an Iterator?

One of the central concepts in Python is that of *iterator* (sometimes confused with *iterable*). An iterator is an object that produces a stream of values, one at a time.
All lists can be iterated, but not all iterators are lists. There are many functions, such as `reversed`, that produce iterators that are not lists. These cannot be indexed or printed in a useful way, at least not directly. Here’s an example:

```python
>>> iter1 = reversed([1, 2, 3, 4])
>>> print(iter1)
<list_reverseiterator object at 0x1111d7f28>
```

However, you can convert an iterator to a list and then print it, index it, or slice it:

```python
>>> print(list(iter1))
[4, 3, 2, 1]
```

Iterators in Python work with `for` statements. For example, because `iter1` is an iterator, the following lines of code work perfectly well.

```python
>>> iter1 = reversed([1, 2, 3, 4])
>>> for i in iter1:
...     print(i, end=' ')
4 3 2 1
```

Iterators have `state information`; after reaching the end of its series, an iterator is exhausted. If we used `iter1` again without resetting it, it would produce no more values.

### 4.10.2 Introducing Generators

A generator is one of the easiest ways to produce an iterator. But the generator function is not itself an iterator. Here’s the basic procedure.

1. Write a generator function. You do this by using a `yield` statement anywhere in the definition.
2. Call the function you completed in step 1 to get an iterator object.
3. The iterator created in step 2 is what yields values in response to the `next` function. This object contains state information and can be reset as needed.

Figure 4.4 illustrates the process.
A generator function is really a generator factory!

**Figure 4.4.** Returning a generator from a function

Here’s what almost everybody gets wrong when trying to explain this process: It looks as if the `yield` statement, placed in the generator function (the thing on the left in Figure 4.4), is doing the yielding. That’s “sort of” true, but it’s not really what’s going on.

The generator function defines the behavior of the iterator. But the iterator object, the thing to its right in Figure 4.4, is what actually executes this behavior.

When you include one or more `yield` statements in a function, the function is no longer an ordinary Python function; `yield` describes a behavior in which the function does not return a value but sends a value back to the caller of `next`. State information is saved, so when `next` is called again, the iterator advances to the next value in the series without starting over. This part, everyone seems to understand.

But—and this is where people get confused—it isn’t the generator function that performs these actions, even though that’s where the behavior is defined. Fortunately, you don’t need to understand it; you just need to use it. Let’s start with a function that prints even numbers from 2 to 10:

```python
def print_evens():
    for n in range(2, 11, 2):
        print(n)
```

Now replace `print(n)` with the statement `yield n`. Doing so changes the nature of what the function does. While we’re at it, let’s change the name to `make_evens_gen` to have a more accurate description.
def make_evens_gen():
    for n in range(2, 11, 2):
        yield n

The first thing you might say is “This function no longer returns anything; instead, it yields the value n, suspending its execution and saving its internal state.”

But this revised function, make_evens_gen, does indeed have a return value! As shown in Figure 4.4, the value returned is not n; the return value is an iterator object, also called a “generator object.” Look what happens if you call make_evens_gen and examine the return value.

```python
>>> make_evens_gen()
<generator object make_evens_gen at 0x1068bd410>
```

What did the function do? Yield a value for n? No! Instead, it returned an iterator object, and that’s the object that yields a value. We can save the iterator object (or generator object) and then pass it to `next`.

```python
>>> my_gen = make_evens_gen()
>>> next(my_gen)
2
>>> next(my_gen)
4
>>> next(my_gen)
6
```

Eventually, calling `next` exhausts the series, and a `StopIteration` exception is raised. But what if you want to reset the sequence of values to the beginning? Easy. You can do that by calling `make_evens_gen` again, producing a new instance of the iterator. This has the effect of starting over.

```python
>>> my_gen = make_evens_gen()  # Start over
>>> next(my_gen)
2
>>> next(my_gen)
4
>>> next(my_gen)
6
```
What happens if you call `make_evens_gen` every time? In that case, you keep starting over, because each time you’re creating a new generator object. This is most certainly not what you want.

```python
>>> next(make_evens_gen())
2
>>> next(make_evens_gen())
2
>>> next(make_evens_gen())
2
```

Generators can be used in `for` statements, and that’s one of the most frequent uses. For example, we can call `make_evens_gen` as follows:

```python
for i in make_evens_gen():
    print(i, end='  ')
```

This block of code produces the result you’d expect:

```plaintext
2  4  6  8  10
```

But let’s take a look at what’s really happening. The `for` block calls `make_evens_gen` one time. The result of the call is to get a generator object. That object then provides the values in the `for` loop. The same effect is achieved by the following code, which breaks the function call onto an earlier line.

```python
>>> my_gen = make_evens_gen()
>>> for i in my_gen:
    print(i, end='  ')
```

Remember that `my_gen` is an iterator object. If you instead referred to `make_evens_gen` directly, Python would raise an exception.

```python
for i in make_evens_gen:       # ERROR! Not an iterable!
    print(i, end='  ')
```

Once you understand that the object returned by the generator function is the generator object, also called the iterator, you can call it anywhere an `iterable` or `iterator` is accepted in the syntax. For example, you can convert a generator object to a list, as follows.

```python
>>> my_gen = make_evens_gen()
>>> a_list = list(my_gen)
>>> a_list
[2, 4, 6, 8, 10]
```
The problem with the last few statements in this example is that each time you iterate through a sequence using a generator object, the iteration is exhausted and needs to be reset.

```python
>>> my_gen = make_evens_gen()  # Reset!
>>> a_list = list(my_gen)
>>> a_list
[2, 4, 6, 8, 10]
```

You can of course combine the function call and the `list` conversion. The list itself is stable and (unlike a generator object) will retain its values.

```python
>>> a_list = list(make_evens_gen())
>>> a_list
[2, 4, 6, 8, 10]
```

One of the most practical uses of an iterator is with the `in` and `not in` keywords. We can, for example, generate an iterator that produces Fibonacci numbers up to and including N, but not larger than N.

```python
def make_fibo_gen(n):
    a, b = 1, 1
    while a <= n:
        yield a
        a, b = a + b, a
```

The `yield` statement changes this function from an ordinary function to a generator function, so it returns a generator object (iterator). We can now determine whether a number is a Fibonacci by using the following test:

```python
n = int(input('Enter number: '))
if n in make_fibo_gen(n):
    print('number is a Fibonacci. ')
else:
    print('number is not a Fibonacci. ')
```

This example works because the iterator produced does not yield an infinite sequence, something that would cause a problem. Instead, the iterator terminates if n is reached without being confirmed as a Fibonacci.

Remember—and we state this one last time—by putting `yield` into the function `make_fibo_gen`, it becomes a generator function and it returns the
generator object we need. The previous example could have been written as follows, so that the function call is made in a separate statement. The effect is the same.

```python
n = int(input('Enter number: '))
my_fibo_gen = make_fibo_gen(n)
if n in my_fibo_gen:
    print('number is a Fibonacci. ')
else:
    print('number is not a Fibonacci. ')
```

As always, remember that a generator function (which contains the `yield` statement) is not a generator object at all, but rather a generator factory. This is confusing, but you just have to get used to it. In any case, Figure 4.4 shows what’s really going on, and you should refer to it often.

## 4.11 Accessing Command-Line Arguments

Running a program from the command lets you provide the program an extra degree of flexibility. You can let the user specify *command-line arguments*; these are optional arguments that give information directly to the program on start-up. Alternatively, you can let the program prompt the user for the information needed. But use of command-line arguments is typically more efficient.

Command-line arguments are always stored in the form of strings. So—just as with data returned by the `input` function—you may need to convert this string data to numeric format.

To access command-line arguments from within a Python program, first import the `sys` package.

```python
import sys
```

You can then refer to the full set of command-line arguments, including the function name itself, by referring to a list named `argv`.

```python
argv     # If 'import sys.argv' used
sys.argv  # If sys imported as 'import sys'
```

In either case, `argv` refers to a list of command-line arguments, all stored as strings. The first element in the list is always the name of the program itself. That element is indexed as `argv[0]`, because Python uses zero-based indexing.
4.11 Accessing Command-Line Arguments

For example, suppose that you are running quad (a quadratic-equation evaluator) and input the following command line:

```
python quad.py -1 -1 1
```

In this case, `argv` will be realized as a list of four strings. Figure 4.5 illustrates how these strings are stored, emphasizing that the first element, `argv[0]`, refers to a string containing the program name.

```
len(argv) = 4
```

```
| "quad.py" | "-1" | "-1" | "1"
|------------|------|------|------
```

*Figure 4.5. Command-line arguments and argv*

In most cases, you’ll probably ignore the program name and focus on the other arguments. For example, here is a program named silly.py that does nothing but print all the arguments given to it, including the program name.

```
import sys
for thing in sys.argv:
    print(thing, end='  ')
```

Now suppose we enter this command line:

```
python silly.py arg1 arg2 arg3
```

The Terminal program (in Mac) or the DOS Box prints the following:

```
silly.py arg1 arg2 arg3
```

The following example gives a more sophisticated way to use these strings, by converting them to floating-point format and passing the numbers to the quad function.

```
import sys

def quad(a, b, c):
    '''Quadratic Formula function.''
    determin = (b * b - 4 * a * c) ** .5
    x1 = (-b + determin) / (2 * a)
```
def main():
    '''Get argument values, convert, call quad.'''

    s1, s2, s3 = sys.argv[1], sys.argv[2], sys.argv[3]
    a, b, c = float(s1), float(s2), float(s3)
    x1, x2 = quad(a, b, c)
    print('x values: {}, {}'.format(x1, x2))

main()

The interesting line here is this one:

    s1, s2, s3 = sys.argv[1], sys.argv[2], sys.argv[3]

Again, the `sys.argv` list is zero-based, like any other Python list, but the program name, referred to as `sys.argv[0]`, typically isn’t used in the program code. Presumably you already know what the name of your program is, so you don’t need to look it up.

Of course, from within the program you can’t always be sure that argument values were specified on the command line. If they were not specified, you may want to provide an alternative, such as prompting the user for these same values.

Remember that the length of the argument list is always N+1, where N is the number of command-line arguments—beyond the program name, of course.

Therefore, we could revise the previous example as follows:

```python
import sys

def quad(a, b, c):
    '''Quadratic Formula function.'''

determin = (b * b - 4 * a * c) ** .5
    x1 = (-b + determin) / (2 * a)
    x2 = (-b - determin) / (2 * a)
    return x1, x2

def main():
    '''Get argument values, convert, call quad.'''

    s1, s2, s3 = sys.argv[1], sys.argv[2], sys.argv[3]
    a, b, c = float(s1), float(s2), float(s3)
    x1, x2 = quad(a, b, c)
    print('x values: {}, {}'.format(x1, x2))

main()
```
if len(sys.argv) > 3:
    s1, s2, s3 = sys.argv[1], sys.argv[2], sys.argv[3]
else:
    s1 = input('Enter a: ')
    s2 = input('Enter b: ')
    s3 = input('Enter c: ')
    a, b, c = float(s1), float(s2), float(s3)
    x1, x2 = quad(a, b, c)
    print('x values: {}, {}.'.format(x1, x2))

main()

The key lines in this version are in the following if statement:

if len(sys.argv) > 3:
    s1, s2, s3 = sys.argv[1], sys.argv[2], sys.argv[3]
else:
    s1 = input('Enter a: ')
    s2 = input('Enter b: ')
    s3 = input('Enter c: ')
    a, b, c = float(s1), float(s2), float(s3)

If there are at least four elements in sys.argv (and therefore three command-line arguments beyond the program name itself), the program uses those strings. Otherwise, the program prompts for the values.

So, from the command line, you'll be able to run the following:

    python quad.py 1 -9 20

The program then prints these results:

    x values: 4.0 5.0

Chapter 4  Summary

A large part of this chapter presented ways to improve your efficiency through writing better and more efficient Python code. Beyond that, you can make your Python programs run faster if you call the print function as rarely as possible from within IDLE—or else run programs from the command line only.

A technique helpful in making your code more efficient is to profile it by using the time and datetime packages to compute the relative speed of the code, given different algorithms. Writing decorators is helpful in this respect, because you can use them to profile function performance.
One of the best ways of supercharging your applications, in many cases, is to use one of the many free packages available for use with Python. Some of these are built in; others, like the `numpy` package, you’ll need to download.

**Chapter 4 Questions for Review**

1. Is an assignment operator such as `+=` only a convenience? Can it actually result in faster performance at run time?

2. In most computer languages, what is the minimum number of statements you’d need to write instead of the Python statement `a, b = a + b, a`?

3. What’s the most efficient way to initialize a list of 100 integers to 0 in Python?

4. What’s the most efficient way of initializing a list of 99 integers with the pattern `1, 2, 3` repeated? Show precisely how to do that, if possible.

5. If you’re running a Python program from within IDLE, describe how to most efficiently print a multidimensional list.

6. Can you use list comprehension on a string? If so, how?

7. How can you get help on a user-written Python program from the command line? From within IDLE?

8. Functions are said to be “first-class objects” in Python but not in most other languages, such as C++ or Java. What is something you can do with a Python function (callable object) that you cannot do in C or C++?

9. What’s the difference between a wrapper, a wrapped function, and a decorator?

10. When a function is a generator function, what does it return, if anything?

11. From the standpoint of the Python language, what is the one change that needs to be made to a function to turn it into a generator function?

12. Name at least one advantage of generators.

**Chapter 4 Suggested Problems**

1. Print a matrix of 20 × 20 stars or asterisks (*). From within IDLE, demonstrate the slowest possible means of doing this task and the fastest possible means. (Hint: Does the fastest way utilize string concatenation of the `join`
method?) Compare and contrast. Then use a decorator to profile the speeds of the two ways of printing the asterisks.

2 Write a generator to print all the perfect squares of integers, up to a specified limit. Then write a function to determine whether an integer argument is a perfect square if it falls into this sequence—that is, if \( n \) is an integer argument, the phrase \( n \text{ in square_iter}(n) \) should yield True or False.
Symbols

& (ampersands) for Boolean arrays, 416–417
intersection set operator, 579
* (asterisks). See Asterisks (*)
@ (at signs)
    function name reassignment, 131–132
    native mode, 277
\ (backslashes). See Backslashes (\)
^ (caret)
    regular expressions, 194, 196–197
    shape characters, 444
    symmetric difference set operator, 580
    text justification, 165
: (colons). See Colons (:) 
, (commas)
    arguments, 16
    lists, 22
    thousands place separator, 152, 154, 168–170
{} (curly braces). See Curly braces ({})
$ (dollar signs) in regular expressions, 185–187, 194
. (dots). See Decimal points (.), Dots (.)
= (equal signs). See Equal signs (=)
! (exclamation points). See Exclamation points (!)
> (greater than signs). See Greater than signs (>)

# (hashtags)
    comments, 3
    regular expressions, 216
< (less than signs). See Less than signs (<)
- (minus signs). See Minus signs (-)
() (parentheses). See Parentheses ()
% (percent sign operator). See Percent sign operator (%)
+ (plus signs). See Plus signs (+)
? (question marks)
    format specifier, 270
    jump-if-not-zero structure, 502–504
    regular expressions, 215–216
" ' (quotation marks) in strings, 19–20
; (semicolons) for multiple statements, 112, 590
/ (slashes)
    arrays, 407
    division, 5
    fractions, 352
[] (square brackets). See Square brackets ([]) 
_ (underscores)
    inside large numbers, 115
    magic methods, 295
    variables, 4, 292, 487–488
| (vertical bars)
    alteration operator in regular expressions, 191–192
Index

OR operator for Boolean arrays, 416–417
OR operator in regular expressions flags, 192–193
union set operator, 581

A
\A character in regular expressions, 194
Abbreviations in regular expressions, 224
abort function, 248
abs function
description, 550
numpy package, 432
__abs__ method, 296, 308
acos function, 376
add_func function definition, 82
__add__ method
binary operator, 296
description, 305–307, 311
Money class, 339–340, 344
add method for sets, 28, 577
Addition
arrays, 407, 416
complex numbers, 356
in-place operation, 283
lists, 82
magic methods, 305–307
money calculator, 343–344
operators, 5
reflection, 311–312
RPN application, 79
Algebra, linear, 456–463
Aliases for packages, 120
Align characters
strings, 172–173
text justification, 164–166
all function, 550
__all__ symbol, 484–487
Alpha characters, testing for, 48
Alphanumeric characters, testing for, 48
Alteration operator (|) in regular expressions, 191–192
Ampersands (&) for Boolean arrays, 416–417
__and__ method, 296
AND operator (\&) for Boolean arrays, 416–417
and operator, logical, 15
any function, 550
append function
description, 432
lists, 22–23, 73, 78
rows and columns, 474
arange function
description, 393
numpy, 468
overview, 396
arccos function, 432
arcsin function, 432
arctan function, 432
*args list, 125–127
Arguments
command-line, 138–141
default values of, 17
functions, 16–19
multiple types, 320–322
named (keyword), 19
passing through lists, 89–90
print, 8
range, 24
variable-length lists, 125–128
Arithmetic operators
magic methods, 296, 304–308
summary, 5
array function, 394–395
array package
description, 122
overview, 387–390
Arrays
arange function, 396
batch operations, 406–409
Boolean, 415–417
copy function, 402
Index

creating, 392–395
dot product, 456–460
empty function, 397–398
eye function, 398–399
fromfunction function, 403–405
full function, 401–402
linear algebra functions, 462–463
linspace function, 396–397
mixed-data records, 469–471
multiplication table, 405–406
ones function, 399–400
outer product, 460–462
review questions, 429–430
rows and columns, 424–428
Sieve of Eratosthenes, 417–419
slicing, 410–415
standard deviation, 419–424
suggested problems, 430
summary, 429
zeros function, 400–401

as np clause, 391
as_tuple method, 333
ascii function, 551
ASCII values
   binary files, 246
   character codes, 43
   conversions, 36, 43–44
   point order, 37
   regular expressions, 193
text files, 246
type specifier, 173
asin function, 376
Aspect ratio for circles, 452–454
assert statement, 590–591
assign_op function definition, 263–265
Assignment operators, 4–5
Assignments
   arrays, 377, 379
   combined operators, 4–5, 98–100
description, 2
   list slices, 67

magic methods, 296, 313
multiple, 100–101
overview, 587–588
precedence, 548
RPN application operators, 262–268
strings, 21
tuples, 14–15, 101–104

Asterisks (*)
   arrays, 407
   Boolean arrays, 416
   exponentiation, 5
   import statement, 121, 484–485
   lists, 67–69, 104–105
magic methods, 307
   multiplication, 5
   regular expressions, 182–183, 215
   shape characters, 444
   strings, 37–38, 104–105
tuples, 103
   variable-length argument lists, 125–128
   variable-length print fields, 151–152

At signs (@)
   function name reassignment, 131–132
   native mode, 277
   use with decorators, 131
atan function, 376
Attributes, setting and getting, 322–323
Axes adjustments, 467–468
axis function, 454–455

B
   \B character in regular expressions, 194
   \b character in regular expressions, 194
b color character, 443
%b format specifier, 155
b type specifier, 173–174, 176
Backslashes (\)
   line continuation, 97
   regular expressions, 182–185, 187,
   194–196, 212
   strings, 20
Backtracking in regular expressions, 199–200
Bases in logarithmic functions, 381–382
Batch operations for arrays, 406–409
Bell curves, 370–373
benchmarks function definition, 392
Big endian bytes vs. little endian, 276–278
bin function
  conversion functions, 48
  description, 551
bin_op function definition, 239–242, 264–265
Binary files
  approaches to, 247–249
  description, 246–248
  read/write operations, 252–254, 268–278
  review questions, 283
  suggested problems, 283–284
  summary, 282
  vs. text, 245–246
Binary numbers
  conversion functions, 47–48
  type specifier, 173–174, 176
Binary operators for magic methods, 296
Binomial Theorem, 370
Bins for histograms, 446–452
Bitwise operators
  Boolean arrays, 416–417
  combined, 98
  inversion, 308
  listed in order of precedence, 547
  magic methods, 296
  shift, 296, 313
bool dtype value, 394
bool function, 551–552
__bool__ method
  conversions, 296, 314–315
  description, 308
  object representation, 299
Boolean values
  arrays, 415–419
  benefits, 107
  conversions, 296, 299, 314–315
dtype, 394
  format specifier, 270
  operators, 15–16
  string methods, 48–49
break statement
  overview, 591
  while statements, 13
Building strings, 44–46
Built-in functions, 549–576
Bytes
  big endian vs. little endian, 276–278
  format specifier, 270
  bytes function, 552
  bytes package, 247–248, 268
C
c color character, 443
c format specifier, 270
c type specifier, 173
calc_binary function definition, 176
calcsize function, 270–271
Calculator in Money class, 342–345
__call__ method, 296
callable function, 552
Canonical number example for regular expressions, 217–218
Canonical representation in source code, 161–162
Cards in Deck class, 365–370
Carets (^)
  regular expressions, 194, 196–197
  shape characters, 444
  symmetric difference set operator, 580
  text justification, 165
Case conversion methods, 49–50
Case sensitivity
  Boolean values, 13
  comparisons, 43
  list sorts, 77
  names, 4
  regular expressions, 192–193
  strings, 36
casefold method, 36, 77
ceil function, 377
__ceil__ method, 296, 308
center method, 54–55
Centering text, 55–56, 165–166
Chained comparisons, 108–109
Character codes, 42–44
Character sets for regular expressions
description, 185–186, 193
working with, 195–197
Characters
format specifier, 270
replacing, 108
strings as lists of, 107
Charts. See Graphs and charts
chflags function, 249
chmod function, 249
chown function, 249
chr function
description, 552
strings, 43
chroot function, 248
Circles, aspect ratio, 452–454
class statement, 591–593
Classes
attributes, 322–323
complex, 353–357
Decimal, 329–332
doc strings, 117–119
floating-point limitations, 328–329
forward reference limitation, 289–290
Fraction, 349–353
inheritance, 293–295
__init__ and __new__ methods, 288–289
instance variables, 286–288
magic methods, 295–298
methods, 290–292
Money. See Money class
multiple argument types, 320–322
numeric, 327–328
review questions, 324–325
suggested problems, 325
summary, 323–324
syntax, 285–287
variables, 292
clear method
dict_obj, 583
lists, 73
set_obj, 578
close method, 254
__cmp__ method, 302
Code, module-level, 19, 30
Code point order
lists, 72
strings, 37
Code refactoring
RPN application, 268
stock-market application, 525–526
Collections
dictionaries and sets, 87–89
lists. See Lists
magic methods, 316–318
Colons (:)
dictionaries, 88
for statements, 23
format method, 163
function definitions, 9
if statements, 11
lists, 65
multidimensional slicing, 413
print-field width, 163–164
regular expressions, 215
text justification, 165
Color characters, plotting, 443
Columns in arrays, 424–428
Combined assignment operators
description, 4–5
precedence, 548
working with, 98–100
Comma-separated value (CSV) files, 472
Command line
arguments, 138–141
Macintosh systems, 116–117
Command line (continued)
  pip utility, 117
  review questions, 142
  suggested problems, 142–143
  summary, 141–142
Windows-based systems, 115–116

Commas (,)
  arguments, 16
  lists, 22
  thousands place separator, 152, 154, 168–170

Comments, 3

Comparisons
  chained, 108–109
  lists, 68–69, 72
  magic methods, 296, 300–304
  operator summary, 15–16
  strings, 37

compile function
  description, 553
  regular expressions, 190

Compiling regular expressions, 188–192
complex class
  description, 328
  overview, 353–357
complex functions, 553–555
__complex__ method, 296, 314–315

Compound interest, plotting, 444–446

Concatenation
  join method, 44–46
  lists, 67
  strings, 21, 36–38

Consonants, testing for, 43

Containers in magic methods, 296
Containment in Money class, 336–338
__contains__ method, 296, 317

continue statement
  overview, 593
  while statements, 13

Control structures
  for statements, 23–25
  if statements, 11–12

RPN application, 502–504
  while statements, 12–14

Conversions
  characters, 43–44
  complex numbers, 354
  degree and radian functions, 376
  to fixed-length fields, 269–271
  magic methods, 296, 314–315
  numeric, 34–36
  repr vs. string, 161–162
  string functions, 47–48

  copy function
    description, 393
    overview, 402

  copy method
    dict_obj, 584
    set_obj, 578

Copying
  arrays, 363, 402
  lists, 61, 69–71
  corrcoef function, 474

Correlation coefficients, 474

  cos function
    math package, 376
    numpy package, 431, 437
    tree height calculation, 379–380
  cosh function, 377

Cosine waves
  plotting, 437–439
  vs. sine, 442–443

  count method
    lists, 75
    strings, 51

  count_nums function definition, 130–131

CSV (comma-separated value) files, 472

Curly braces ({}),
  dictionaries, 26
  format method, 157–158
  line continuation, 97
  sets, 87
  variable-size fields, 177

Currency. See Money class
D
\d character in regular expressions, 183–184, 194–195
%d format specifier, 147
d format specifier, 270
D shape character for plotting lines, 444
d shape character for plotting lines, 444
d type specifier, 173
Data dictionaries for symbol tables, 262–265
Data frames, 123
Data reader for stock-market application, 519–521
Data types
    arguments, 18–19, 320–322
dictionary keys, 27
    elementary, 6–7
    lists, 60
type specifiers, 173–174
deal function definition, 366–370
DEBUG flag, 192–193
Decimal class, 329–332
    application, 335–336
description, 327
    overview, 329–332
    review questions, 357–358
    special operations, 332–334
    suggested problems, 358
    summary, 357
Decimal numbers
    testing for, 48
type specifier, 173
decimal package, 122
Decimal points (.)
    floating point data, 6
    format specifiers, 154–156
    precision field, 149, 170–171
    regular expressions, 224–225, 230
Deck class
    objects, 365–368
    pictograms, 368–370
decode function, 271
Decorator functions, 128–132
Deep copying of lists, 69–71
def statement
    function definitions, 9
    overview, 594
Default arguments, 18
Degree and radian conversion functions, 376
degrees function, 376
de del statement, 594
delattr function, 555
Delimiter
    split, 53–54
    strings, 161
text files, 472–473
__delitem__ method, 296, 316
Denominators in fractions, 349–353
DFAs (deterministic finite automata), 189
dictionaries
    dictionary comprehension, 87–89
    methods, 583–586
    overview, 26–27
    symbol tables, 262–265
difference method for sets, 578
difference_update method for sets, 578
Digits in regular expressions, 183–184, 194–195
dir function, 555–556
discard method, 579
Distance operator for magic methods, 307
Distributions for random numbers, 359
Division
    arrays, 407
    floating point and integer values, 6–7
    magic methods, 305
    operators, 5
    reflection, 311
divmod function
    description, 556
tuples, 7
__divmod__ method, 305
_do_duo_plot function definition, 527–529
_do_highest_plot function definition, 531
do_input function definition, 498–499
do_movingavg_plot function definition, 539
do_plot function definition, 521–522, 526
do_println function definition, 498
do_prints function definition, 498
do_printvar function definition, 498–499
do_split_plot function definition, 537
do_trials function definition, 361–363
do_volume_plot function definition, 534
Doc strings, 117–119
Dollar signs ($) in regular expressions, 185–187, 194
DOS Box application, 116
dot linear-algebra function
arrays, 457–460
description, 463
Dot product, 456–460
DOTALL flag in regular expressions, 193, 224, 226
Dots (.), decimal points. See Decimal points (.)
function qualifiers, 120
functions, 46
instance variables, 287
regular expressions, 194
shape characters, 444
dump method, 254, 279
dumps method, 254
Dunder methods, 295
Dynamic attributes, 322–323

E
e constant, 376–377, 432
%E format specifier, 147
%e format specifier, 147
E type specifier, 173
e type specifier, 173
Elementary data types, 6–7
elif statements
  example, 12
  overview, 595
else clause
  exceptions, 250
  if statements, 11–12
  loops, 106
  overview, 595
empty function
  description, 393
  overview, 397–398
end attribute in regular expressions, 204
endswith method, 50–51
enum values with range, 113
enumerate function
  description, 556–557
  lists, 64
EOFError exceptions, 280
__eq__ method, 296, 302
Equal signs (=)
  Boolean arrays, 416
  equality tests, 15
  lists, 68–69
  magic methods, 300
  precedence, 548
  regular expressions, 216
  strings, 37
  text justification, 165–166
Equality
  Boolean operators, 15
  decimal objects, 333
  lists, 68–69
  sets, 28
  strings, 36
eval function
  command strings, 81
  description, 557–558
eval_scores function definition, 74
except statements
  overview, 595
  try blocks, 250
Exclamation points (!)
inequality tests, 15
lists, 68–69
magic methods, 300
regular expressions, 216
RPN application, 504–508
strings, 37
exec functions, 249, 558
exp function, 432
Expectations in random behavior, 361
Exponentiation operators, 5, 432
Exponents
format specifier, 147
logarithmic functions, 381–382
Expressions, regular. See Regular expressions
extend function, 73
eye function
description, 393
overview, 398–399

F

f dtype value, 394
%F format specifier, 147
%f format specifier, 147
f format specifier, 270
f type specifier, 173
False keyword, 15–16, 107
fibo function definition, 101
Fibonacci sequence
generating, 137–138
printing, 14
RPN application, 504–505
tuples application, 101
FileNotFoundException exception, 250
Files
binary. See Binary files
converting data to fixed-length fields, 269–271
file/directory system, 248–249
file-opening exceptions, 249–252
file pointer, 257–258
numpy package, 471–474
read/write operations. See Read/write operations
review questions, 283
suggested problems, 283–284
summary, 282
text. See Text files
with keyword, 252
Fill characters
strings, 56, 172–173
text justification, 164–166
filter function
description, 558–559
lists, 81
finally clause, 250, 600
Financial applications, 464–467
Financial data from Internet. See Stock-market application
find_divisor function definition, 106
find method, 52
findall function, 206–209
First-class objects, functions as, 123–124
Fixed-length fields, converting to, 269–271
Fixed-length strings in read/write operations, 273–274
Flags in regular expressions, 192–193
float dtype value, 394
float function, 559
__float__ method, 296, 314–315
float32 dtype value, 394
float64 dtype value, 394
Floating-point numbers
conversions, 35
dividing, 6–7
format specifier, 147, 270
problems with rounding errors, 328–329
overview, 6
type specifier, 173
floor function, 377
__floor__ method, 296, 308
__floordiv__ method, 305
for statements
intelligent use, 97–98
one-line, 111
overview, 595–596
working with, 23–25
fork function, 248
format function
  description, 559–560
  working with, 152–156
format method
  format specifiers. See Format specifiers
  working with, 156–158
__format__ method, 153, 295, 297–298
Format specifiers
  leading-zero character, 167–168
  overview, 162–163
  precision, 170–173
  print-field width, 163–164
  sign character, 166–167
  text formatting, 147–150
  text justification, 164–166
  thousands place separator, 168–170
  type specifiers, 173–176
Formatting text, 145
  format function, 152–156
  format method, 156–158
  format specifiers. See Format specifiers
  ordering by position, 158–161
  percent sign operator, description,
   145–146
  percent sign operator, format specifiers,
   147–150
  percent sign operator, variable-length
   print fields, 150–152
  repr conversions, 161–162
  review questions, 179
  suggested problems, 179–180
  summary, 178–179
  variable-size fields, 176–178
Forward reference problem, 19, 289–290
Fraction class, 328, 349–353
Fractions, floating-point limitations, 328
fractions package, 122, 306
from syntax for import statement, 483
fromfunction function
  description, 393
  overview, 403–405
frozenset function, 560
full function
  description, 393
  overview, 401–402
fullmatch function, 187
Functions
  arguments and return values, 16–19
  built-in, 549–576
  decorators and profilers, 128–132
  default arguments, 17
  definitions, 9–11
  doc strings, 117–119
  as first-class objects, 123–124
  forward reference problem, 19
  lists, 71–73
  named arguments, 19
  strings, 46–47
  switch simulation, 109–110
  variable-length argument lists, 125–128
functools package, 82
Fundamentals
  arithmetic operators, 5
  Boolean operators, 15–16
  combined assignment operators, 4–5
  data types, 6–7
  dictionaries, 26–27
  Fibonacci sequence application, 14–15
  for statements, 23–25
  forward reference problem, 19
  function arguments and return values,
   16–19
  function definitions, 9–11
  global and local variables, 29–31
  if statements, 11–12
  input and output, 7–9
  lists, 21–23
  quick start, 1–4
  review questions, 31–32
  sets, 28–29
  strings, 19–21
  suggested problems, 32
  summary, 31
tuples, 25–26
variables and names, 4
while statements, 12–14
Future values in financial applications, 464–465

G
g color character in plotting lines, 443
%G format specifier, 147
%g format specifier, 147
G type specifier, 173
g type specifier, 173
Game of Life simulation, 414–415
Garbage collection, 46
__ge__ method, 302
gen_rand function definition, 375
Generators, 132
    iterators, 132–133
    overview, 133–138
get_circ function definition, 378
get_circle_area function definition, 483
Generator, 132
get_height function definition, 380
get method, 27, 584
Get-random-number operator, 504–508
get_square_area function definition, 483
get_std1 function definition, 423
g get_std2 function definition, 423
get_std3 function definition, 423
get_str function definition, 498
gt get_triangle_area function definition, 484
gattr function
    description, 560
    dynamic attributes, 323
    stock-market application, 527–530
getcontext function, 334
getenv function, 249
getenvb function, 249
__getitem__ method, 296, 316
__getstate__ method, 296
getwd function, 248
global statement
    overview, 596–597
    RPN application, 500–501
    variables, 30–31
Global variables, 29–31
globals function, 560
Golden ratio, 378
Googleplex, 6
Graphs and charts
    axes adjustments, 467–468
    high and low data, 530–533
    moving-average lines, 538–540
    pie, 455–456
    splitting, 536–537
    stock-market application, 521–523, 527–530
    subplots, 536–537
    time periods, 534–536
    titles and legends, 524–525
Greater-than operator in RPN application, 504–508
Greater than signs (>)
    big endian mode, 277
    Boolean arrays, 416
    equality tests, 15
    lists, 68–69
    magic methods, 300
    RPN application, 504–508
    shape characters, 444
    strings, 37
    text justification, 165
Greedy vs. non-greedy matching in regular expressions, 219–224
Ground division
    integers, 7
    operators, 5
    group attribute for regular expressions, 203–205
groupdict attribute for regular expressions, 204
Grouping problem in regular expressions, 208–209

Groups
- named, 231–234
- noncapture, 217–219
- regular expressions, 193, 198–199

groups attribute for regular expressions, 204

_groups_ method, 296, 302

Guessing game
- logarithmic functions, 382–384
- RPN application, 507–508

H

H format specifier, 270
h format specifier, 270
H shape character in plotting lines, 444
h shape character in plotting lines, 444
hasattr function, 561
hash function, 561
__hash__ method, 299

Hashtags (#)
- comments, 3
- regular expressions, 216
hcos function, 432
Height of tree calculations, 378–380
help function, 561
help statement for strings, 19
hex function
- conversion functions, 48
  description, 561
__hex__ method, 296, 314–315

Hexadecimal numbers
- conversions, 35, 47–48
  format specifier, 147–148
  type specifier, 174–175
High and low data in stock-market application, 530–533
hist function, 447
histogram function, 449–452
Histograms, 446–452
hsin function, 432
htan function, 432

Hyperbolic functions, 377

I

i dtype value, 394
%i format specifier, 147
I/O directives in RPN application, 496–499
__iadd__ method, 296, 313
__iand__ method, 313
id function, 561
Identifiers, testing for, 48
__idiv__ method, 313
IDLE (interactive development environment), 1
idle3 command, 391
if statements
  indentation, 11–12
  introduction to, 11
  one-line, 112–113
  overview, 597–598
ignore_case function definition, 77
IGNORECASE flag, 192–193, 196
__igrounddiv__ method, 313
__ilshift__ method, 313
 Imaginary portion of complex numbers, 353–357
Imag portion in complex numbers, 354

Immutability
  defined, 21
  dictionary keys, 27
  set members, 28
  strings, 21, 33–34
  tuples, 26
__imod__ method, 313
import statement
  modules, 478–479
  overview, 598
  packages, 119–121
  variations, 482–484
Importing packages, 119–121
__imul__ method, 313
__imult__ method, 296
in operator
  lists, 68
  sets, 577
  strings, 37, 43–44
In-place operators
  assignments, 98–100
  magic methods, 312–314
Indentation
  doc strings, 118
  for statements, 23
  function definitions, 9–10
  if statements, 11–12
  overview, 589–590
__index__ method, 296, 314–315
index method
  lists, 75
  strings, 52
IndexError exception, 62
Indexes
  characters, 20
  lists, 24, 61–65, 73–74
  ordering by position, 159–161
  strings, 39–42
Inequality tests, 15
inf value, 377
Infix notation, 79
info function, 433
Inheritance
  classes, 293–295
  collections, 318
  Money class, 336–337, 347–349
__init__ method
  Deck class, 366–367, 369
  description, 295
  Money class, 341, 345, 347–348
  overview, 288–289
inner linear-algebra function, 463
INPUT directive in RPN application,
  496–497
input function
  description, 562
  strings, 47
  working with, 8–9
Input operations, 562
  overview, 7–9
  splitting input, 53–54
  text files, 254–257
insert function, 73
install command in pip utility, 434
Instances, 286–288
Instantiating classes, 289–290
int dtype value, 394
int function
  conversions, 35
  description, 562
__int__ method, 296, 314–315
int8 dtype value, 394
int16 dtype value, 394
int32 dtype value, 394
int64 dtype value, 394
Integers
  conversions, 35
  dividing, 6–7
  format specifier, 147, 270
  Fraction class, 349
  overview, 6
  random-integer game, 363–364
Interactive development environment
  (IDLE), 1
Interest and interest rates
  financial applications, 464–465
  plotting, 444–446
Internet financial data. See Stock-market
application
intersection method, 28–29, 579
intersection_update method, 579
Inverse trigonometric functions, 376
__invert__ method, 308
Inverting dictionaries, 89
__ior__ method, 313
ipmt function, 466
__ipow__ method, 314
### IQ score histogram example, 446–452

- irshift__ method, 313
- is operator, 37–38
- is not operator
  - correct use, 110–111
  - strings, 37–38
- isalnum method, 48
- isalpha method, 48
- isdecimal method, 48
- isdigit method, 48
- isdisjoint method, 579
- isfile function, 249
- isidentifier method, 48
- isinstance function, 321–322
  - description, 562
  - variable type, 17
- islower method, 48
- isprintable method, 48
- isspace method, 49
- issubclass function, 563
- issubset method, 579
- issuperset method, 580
- istitle method, 49
- __isub__ method, 296, 313
- isupper method, 49
- items method, 89, 585
- iter function, 563–564
- __iter__ method, 296, 316, 319–320
- Iterative searches in regular expressions, 206–208

#### Iterators
- description, 132–133
- random-number generators, 375
- __ixor__ method, 314

### K

- k color character in plotting lines, 443
- Key-value pairs in dictionaries, 26–27
- KeyError exceptions
  - Money class, 341
  - RPN application, 502
- Keys
  - immutable, 33
  - list sorts, 76–77
- keys method, 585
- Keyword arguments (named arguments), 17
- Keywords, 4
- kill function, 248
- kron function, 463
- **kwargs list, 127–128

### L

- L format specifier, 270
- l format specifier, 270
- Lambda functions
  - overview, 83–84
  - RPN application, 239–240
- Large numbers, underscores inside, 115
- Last-in-first-out (LIFO) devices, 78
- lastindex attribute for regular expressions, 203–204
- Law of Large Numbers, 361, 371–372
- Lazy vs. greedy matching in regular expressions, 219–224
- __le__ method, 302
- Leading spaces, stripping, 54–55
- Leading-zero character, 167–168
- Left justification
  - format specifier, 148
  - text, 55, 165–166
- legend function, 524
- Legends in charts, 524–525
len function
   description, 564
   lists, 71–72
   strings, 47
__len__ method, 296, 316
Less than signs (<)
   Boolean arrays, 416
   lists, 68–69
   little endian mode, 277
   magic methods, 300
   regular expressions, 216
   shape characters, 444
   strings, 37
   text justification, 165–166
LIFO (last-in-first-out) devices, 78
limit_denominator method, 351
linalg.det function, 463
Line continuation, 96–97
Line-number checking in RPN application, 500–502
Linear algebra, 456–463
Lines, plotting, 435–444
linspace function
   description, 393
   line plotting, 437–438
   numpy, 433, 445
   overview, 396–397
List comprehension, 84–87
list function, 564–565
listdir function, 248–249
Lists
   vs. arrays, 388–389
   bytes, 268
   of characters, strings as, 107
   contents, 75
   copying, 61, 69–71
   creating, 59–60
   for statements, 24
   functions, 71–73
   in-place operations, 99
   indexing, 61–64
   lambda functions, 83–84
list comprehension, 84–87
   modifying, 73–74
   multidimensional, 90–93
   multiplication, 104–105
   negative indexing, 63
   operators, 67–69
   overview, 21–23
   passing arguments through, 89–90
   reduce function, 81–83
   reorganizing, 75–77
   review questions, 93–94
   RPN application, 78–81
   slicing, 64–67
   as stacks, 78–81
   vs. strings, 39
   suggested problems, 94
   summary, 93
Literal characters in regular expressions, 182
Little endian bytes vs. big endian, 276–278
ljust method, 54–55
load method, 254, 279
load_stock function definition, 519–520
Local variables, 29–31
LOCALE flag, 193
locals function, 565
log function
   description, 381
   math package, 377
   numpy package, 432
log2 function
   description, 381
   math package, 377
   numpy package, 432
log10 function
   description, 381
   math package, 377
   numpy package, 432
Logarithmic functions
   math package, 377
   numpy package, 432
   working with, 381–384
Logical and operation, 15
Logical not operation, 16
Logical or operation, 16
Look-ahead feature in regular expressions
  multiple patterns, 227–228
  negative, 229–231
  overview, 224–227
Loops
  else statements, 106
  for statements, 23–25
  unnecessary, 108
  while statements, 12–14
lower method, 36, 49–50
Lowercase characters
  converting to, 48
  testing for, 48
  `__lshift__` method, 296
`lstrip` method, 54–55
  `__lt__` method, 296, 302

M
m color character in plotting lines, 443
Macintosh systems, command line, 116–117
Magic methods, 285
  arithmetic operators, 304–307
  collections, 316–318
  comparisons, 300–304
  conversion, 314–315
  in-place operators, 312–314
  `__iter__` and `__next__`, 319–320
  objects, 298–300
  overview, 295–296
  reflection, 310–312
  review questions, 324–325
  strings, 297–298
  suggested problems, 325
  summary, 323–324
  unary arithmetic operators, 308–310
  `__main__` module, 477–478, 488–490
make_evens_gen function definition, 135–137
make_timer function definition, 129–130
makedir function, 248
makeplot function, 525–529, 532
map function
  description, 565–566
  lists, 81
match function in regular expressions,
  184–188
match object in regular expressions, 203–204
Matching, greedy vs. non-greedy, 219–224
Math operations in numpy package, 431–433
math package, 120
  description, 122
  function categories, 376–377
  logarithmic functions, 381–384
  overview, 376
  review questions, 385
  special values, 377–378
  suggested problems, 386
  summary, 385
  trigonometric functions, 378–380
matplotlib package
  circles and aspect ratio, 452–454
  compound interest plotting, 444–446
  downloading, 434
  histograms, 446–452
  line plotting, 435–444
  overview, 388
  pie charts, 455–456
  review questions, 475
  suggested problems, 476
  summary, 475
matplotlib.pyplot package
  description, 123
  overview, 388
Matrixes
  large, 91–93
  multidimensional lists, 90–91
  unbalanced, 91
  See also numpy package
max function
  arrays, 420
  description, 566
  lists, 71–72
  strings, 47
mean function for arrays, 420, 422, 425
Mean value
arrays, 421–422	normal distribution, 370–371
median function, 420
Median value
description, 94
numpy, 420–421
Meta characters in regular expressions, 194–195
Metafunctions, 124
Methods
magic. See Magic methods
overview, 290–291
public and private, 292
strings, 46–47
min function
arrays, 420
description, 566–567
lists, 71–72
strings, 47
Minus signs (-)
arrays, 407
difference set operator, 578
format specifiers, 148, 167
magic methods, 307
regular expressions, 186, 196
subtraction, 5
text justification, 165
__missing__ method, 317
Mixed-data records in arrays, 469–471
mkdir function, 248
Module-level code, 19, 30
ModuleNotFoundError exception, 390
Modules
__all__ symbol, 484–487
import statement, 482–484
__main__, 488–490
mutual importing, 490–492
overview, 477–478
review questions, 514
RPN application. See Reverse Polish Notation (RPN) application
suggested problems, 514–515
summary, 513–514
two-module example, 478–482
variables, 487–488
Modulus division
description, 7
operators, 5
reflection, 311
money_calc function definition, 343, 346
Money class, 327
calculator, 342–345
containment, 336–338
default currency, 345–346
designing, 336–337
displaying objects, 338
inheritance, 347–349
operations, 339–342
review questions, 357–358
suggested problems, 358
summary, 357
monthly_payment function definition, 465–467
Mortgage payments, 464–467
Moving-average lines, 538–540
mpl_toolkits package, 463–464
mul_func function definition, 82
__mul__ method, 305–307, 321–322
__mult__ method, 296
Multidimensional array slicing, 412–415
Multidimensional lists, 90–93
MULTILINE flag for regular expressions, 193, 225–226
Multiple argument types for classes, 320–322
Multiple assignment, 100–101
Multiple charts in stock-market application, 527–530
Multiple inheritance, 294–295
Multiple lines, plotting, 441–444
Multiple patterns in regular expressions, 227–228
Multiple statements on one line, 112
Multiple values, returning, 105–106
Multiplication
arrays, 407, 409, 456–462
complex numbers, 355
in-place, 313
lists, 69, 92, 104–105
magic methods, 305, 307
operators, 5
reduce function, 82
reflection, 311
regular expressions, 182
strings, 37–39, 104–105
Multiplication table, 405–406
Mutual importing of modules, 490–492

N
type specifier, 174
__name__ module, 477–478
Named groups in regular expressions, 231–234
Names
mangling, 292
variables, 4
nan value, 377
ndarray class
math operations, 431
statistical-analysis functions, 420
ndarray data type, 391
__ne__ method, 302
__neg__ method, 296, 308–309
Negative indexes
lists, 63
strings, 39
Negative look-ahead in regular expressions, 229–231
Nested blocks, 10
new_hand function definition, 368
__new__ method, 295
Money class, 347–348
overview, 288–289
Newlines, printing, 8
next function for generators, 133–135
__next__ method, 296, 317, 319–320
NFAs (nondeterministic finite automata), 189
Non-greedy vs. greedy matching in regular expressions, 219–224
Non-overlapping searches in regular expressions, 206
Noncapture groups in regular expressions, 217–219
Nondeterministic finite automata (NFAs), 189
None value
Boolean value, 107
dictionaries, 27
equality tests, 38
return value, 17
split, 54
nonlocal statement, 598
Nonnegative indexes for lists, 61–62
__nonzero__ method, 299
Normal distribution, 370–373
normalize method, 332–333
normalvariate function, 360, 371–372
not operation
Boolean value, 107
logical, 16
not in operator
lists, 68
strings, 37, 43
NotImplemented return value, 307, 310, 321–322
np_sieve function definition, 419
Numerator in fractions, 349–353
Numeric classes, 327–328
Numeric conversions with strings, 34–36
numpy package
arrays. See Arrays
axes adjustments, 467–468
circles and aspect ratio, 452–454
compound interest plotting, 444–446
description, 122
downloading and importing, 390–391
Index

financial applications, 464–467
line plotting, 435–444
linear algebra, 456–463
math operations, 431–433
mixed-data records, 469–471
overview, 387–388
pie charts, 455–456
read/write operations, 471–474
review questions, 429–430, 475
suggested problems, 430, 476
sum example, 391–392
summary, 429, 475
three-dimensional plotting, 463–464
numpy.random package
description, 123
overview, 388

O
%o format specifier, 147–148, 155
o shape character in plotting lines, 444
o type specifier, 174–175
Objects
magic methods, 298–300
syntax, 285–287
oct function
conversion functions, 48
description, 567
__oct__ method, 314–315
Octal numbers
conversions, 35, 47–48
format specifier, 147–148
type specifier, 174–175
One-line statements
for loops, 111
if/then/else, 112–113
ones function
description, 393
overview, 399–400
open function, 567–568
open method
description, 252–253
shelve package, 280
open_rpn_file function definition, 261, 266, 495
Operating system package, 248
operator package, 505
Operators
arithmetic, 5
Boolean and comparison, 15–16
combined assignment, 4–5, 98–100
lists, 67–69
magic methods, 304–312
precedence and summary, 547–548
strings, 36–38
__or__ method, 296
or operation, logical, 16
OR operator (|)
Boolean arrays, 416–417
regular expressions flags, 192–193
__orc__ method, 296
ord function
description, 568
strings, 36, 43
Order
lists, 22, 60
by position, 158–161
sets, 28
os package, 248–249
os.path package, 249
Outer linear-algebra function
arrays, 460–462
description, 463
Outer product for arrays, 460–462
Output operations
overview, 7–9
text files, 254–257

P
p format specifier, 270
p shape character in plotting lines, 444
pack function, 269, 271–273, 275
Packages
common, 121–123
importing, 119–121
Packages (continued)
- pip utility, 117
- review questions, 142
- suggested problems, 142–143
- summary, 141–142
Padding in text justification, 165–166
pandas package
- description, 123
- stock-market application, 518
pandas_datareader package, 518
Parentheses ()
- Boolean arrays, 416
- complex numbers, 355
- function definitions, 9–11
- line continuation, 97
- operator precedence, 5, 548
- print statement, 8, 146
- regular expressions, 191, 198–199
- tuples, 25
pass statement
- if statements, 12
- overview, 599
Passing arguments through lists, 89–90
Passwords and regular expressions, 200–203, 227–228
PATH setting in Windows-based systems, 116
Pattern quantifiers in regular expressions, 197–199
Pattern searches in regular expressions, 205–206
Payment periods in financial applications, 464
Pearson correlation coefficient, 474
Pentagrams, plotting, 439–440
Percent sign operator (%)
- format specifiers, 147–150
- percentages display, 174–175
- remainder division, 5, 7
- text formatting, 145–146
- type specifier, 175–176
- variable-length print fields, 150–152
phi value, 378
Phone number example for regular expressions, 183–185
pi constant, 376–378, 432
pickle package, 247–248, 278–280
Pictograms with Deck class, 368–370
Pie charts, 455–456
pie function, 455–456
pip utility
- description, 390
- downloading, 434
- package downloads, 117
pip3 utility
- description, 390
- downloading, 434
play_the_game function definition, 364
plot function
- matplotlib package, 435
- multiple lines, 441–443
- pentagrams, 439–440
- stock-market application, 523
Plotting
- compound interest, 444–446
- lines, 435–444
- three-dimensional, 463–464
Plus signs (+)
- addition, 5
- arrays, 407
- Boolean arrays, 416
- complex numbers, 354
- format specifiers, 166–167
- lists, 67
- regular expressions, 182–183, 190, 215
- shape characters, 444
- strings, 21, 36–37
- pmt function, 464–467
Point class, 306–307
Polymorphism, 293
pop method
- dict_obj, 585
- lists, 75, 78–80
- set_obj, 580
popitem method, 585
__pos__ method, 296, 308
Position, ordering by, 158–161
Positional expansion operator, 126
Positive indexes for lists, 62
Postfix languages, 79
pow function
description, 569
math package, 377
__pow__ method, 296, 305
power function in numpy, 432–433
Power functions
arrays, 407
magic methods, 296, 305
math package, 377
reflection, 311
ppmt function, 466
pr_normal_chart function definition, 371–372
pr_vals_2 function definition, 128
Precedence of operators, 5, 547–548
Precision
format function, 154–156
format specifiers, 148–150, 170–173
Fraction class, 349
strings, 172–173
Present values in financial applications, 464
Prime numbers with Sieve of Eratosthenes, 410–412
Print fields
print-field width, 163–164
variable-length, 150–152
print function
Decimal class, 331–332
description, 569
format specifiers, 147–150
with IDLE, 114
text files, 256
text formatting, 145–146
working with, 8–9
print_me function definition, 291
print_nums function definition, 13
Printable characters, testing for, 48
PRINTLN directive, 496–497
PRINTS directive, 496–497
PRINTVAR directive, 496–497
Private variables and methods, 292, 487–488
Processes, functions for, 248
Profilers, function, 128–132
Pseudo-random sequences, 359, 374
Public variables and methods, 292, 487–488
putenvb function, 249
pydoc utility, 117–119
pyplot function, 435
Q
Q format specifier, 270
q format specifier, 270
quad function definition, 117–119, 139–140
Quantifiers in regular expressions, 193–194, 197–199
Question marks (?)
format specifier, 270
jump-if-not-zero structure, 502–504
regular expressions, 215–216
Quotation marks (") in strings, 19–20
R
r character as raw-string indicator, 184
r color character used in plotting lines, 443
%r format specifier, 147, 150
__radd__ method, 296, 311
radians function
math package, 376
numpy package, 432
tree height calculation, 380
raise statement, 599
randint function
description, 359–360
in do_trials, 362
RPN application, 506
random function, 360
Random numbers in RPN application, 504–508
random package
Deck class, 365–370
description, 122
Fibonacci sequence application, 15
functions, 360
normal distribution, 370–373
overview, 359
random-integer game, 363–364
random-number generator, 374–376
review questions, 385
suggested problems, 386
summary, 385
testing random behavior, 361–363
range function
description, 570
enum values with, 113
for statements, 23–25
list enumeration, 63–64
raw_input function, 8
Raw strings, used in regular expressions, 184
__rdivmod__ method, 311
re package
description, 122
regular expressions, 184
re.Scanner class, 236–243
re.split function, 234–236
read_fixed_str function definition, 273
read_floats function definition, 272
read method, 253, 255–256
read_num function definition, 272
read_rec function definition, 275
read_var_str function definition, 274
Read/write operations
big endian vs. little endian, 276–278
binary files, 268–278
fixed-length strings, 273–274
numpy package, 471–474
one number at a time, 272
pickle package, 278–280
several numbers at a time, 272–273
shelve package, 280–282
strings and numerics together, 275–276
summary, 252–254
text files, 254–257
variable-length strings, 274
readline method, 253, 255–256
readlines method, 253, 255–257
real function, 354
Real portion in complex numbers, 353–357
Reciprocal functions, plotting, 437
reduce function, 81–83
Refactoring code
RPN application, 268
stock-market application, 525–526
Refining matches for regular expressions,
185–188
Reflection magic methods, 296, 301, 310–312
Regex flags in regular expressions, 192–193
regex package, 189
Regular expressions, 181
advanced grammar, 215–216
backtracking, 199–200
basic syntax, 193
countact sets, 195–197
compiling vs. running, 188–192
flags, 192–193
greedy vs. non-greedy matching, 219–224
grouping problem, 208–209
introduction, 181–182
iterative searches, 206–208
look-ahead feature, 224–227
look-ahead feature, multiple patterns, 227–228
look-ahead feature, negative, 229–231
match object, 203–204
meta characters, 194–195
named groups, 231–234
noncapture groups, 217–219
password example, 200–203
pattern quantifiers, 197–199
pattern searches, 205–206
phone number example, 183–185
re.split function, 234–236
refining matches, 185–188
repeated patterns, 210–211
replacing text, 211–213
review questions, 213–214, 243–244
scanner class, 236–243
suggested problems, 214, 244
summary, 213, 243

Remainder division
integers, 7
operators, 5
reflection, 311

remove method
lists, 73–74
sets, 28, 580
removedirs function, 248
rename function, 248

Reorganizing lists, 75–77
Repeated patterns in regular expressions, 210–211
replace method
characters, 108
strings, 53

Replacing
characters, 108
substrings, 53
text, 211–213

repr function
description, 570–571
repr conversions vs. string, 161–162

__repr__ method
description, 297–298
Money class, 342
reset_index method, 521, 534
reshape function, 408
return statement
functions, 16–19
overview, 599

Return values
functions, 16–19
multiple, 105–106
reverse function for lists, 75–77

Reverse Polish Notation (RPN) application
assignment operator, 262–268
changes, 499–500
final structure, 508–513
greater-than and get-random-number operators, 504–508
I/O directives, 496–499
jump-if-not-zero structure, 502–504
line-number checking, 500–502
lists, 78–81
modules, 493–496
regular expressions, 234–236
review questions, 514
scanner class, 236–243
suggested problems, 514–515
summary, 513–514
text files, 258–268

reversed function
description, 571
lists, 71–72
strings, 47
__reversed__ method, 317
rfind method, 52
__rfloordiv__ method, 311
Right justification, 55–56, 165–166
rjust method, 54–56
rmdir function, 248
__rmod method, 311
__rmul__ method, 311
__rmult__ method, 296
rolling function, 538–539
round function
description, 571–572
floating-point numbers, 329
ROUND_HALF_EVEN rounding, 334
__round__ method, 296, 308
Rounding
decimal objects, 332–334
floating-point limitations, 328–330
magic methods, 296, 308
precision specifier, 170
reflection, 311
Rows in arrays, 424–428
RPN application. See Reverse Polish Notation (RPN) application
__rpow__ method, 311
rstrip method, 54–55
__rsub__ method, 296, 311
__rtruediv__ method, 311
Run Module command, 480
Running regular expressions, 188–192

S
\S character in regular expressions, 194
\s character in regular expressions, 194
%s format specifier, 147–148, 150
s shape character in plotting lines, 444
sample function, 360
davetxt function, 474
Scanner class, 236–243, 503–504
Scope of global and local variables, 29–31
Search-and-replace methods for strings, 50–53
search function, 205–206
Searches in regular expressions. See Regular expressions
Seed values for random-number generators, 374
seek method, 254, 257
seekable method, 254, 257
self argument, 288, 290–291
Semicolons (;) for multiple statements, 112, 590
Separator strings, printing, 8
set function, 572
set_list_vals function definition, 89–90
set methods, 577–581
setattr function
  description, 573
  dynamic attributes, 323
setdefault method, 586
__setitem__ method, 296, 316
Sets
  overview, 28–29
  set comprehension, 87–89
  __setstate__ method, 296
  Shallow copying of lists, 69–71
  Shape characters, plotting, 444
  shelve package, 247–248, 280–282
  Shift operators, 296, 313
  Short-circuit logic, 15, 548
  Shortcuts
    Boolean values, 107
    chained comparisons, 108–109
    combined operator assignments, 98–100
    common techniques, 95–96
    else statements, 106
    enum values with range, 113
    for loops, 97–98
    is operator, 110–111
    line continuation, 96–97
    list and string multiplication, 104–105
    loops, 108
    multiple assignment, 100–101
    multiple statements onto a line, 112
    one-line for loops, 111
    one-line if/then/else statements, 112–113
    overview, 95
    print function with IDLE, 114
    replace method, 108
    returning multiple values, 105–106
    review questions, 142
    strings as lists of characters, 107
    suggested problems, 142–143
    summary, 141–142
    switch simulation, 109–110
    tuple assignment, 101–104
    underscores inside large numbers, 115
    show function
      histograms, 447
      line plotting, 439
      matplotlib package, 435–436
      multiple lines, 442
      stock-market application, 524
    shuffle function, 360, 365
Sieve of Eratosthenes
  array slicing for, 410–412
  Boolean arrays, 417–419
Sigma in normal distribution, 370–371
Sign bits for decimal objects, 334
Sign character in format specifiers, 166–167
sin function
  math package, 376
  numpy package, 432–433, 435–436
tree height calculation, 379–380
Sine waves
  vs. cosine, 442–443
  plotting, 435–437
Single-character functions, 42–44
sinh function, 377
size function for arrays, 420
Slashes (/)
  arrays, 407
  division, 5
  fractions, 352
Slicing
  arrays, 410–415
  lists, 64–67
  strings, 39–42
sort function for lists, 22, 75–76
sorted function
  description, 573
  lists, 71–72
  strings, 47
Space characters, testing for, 49
Spaces
  fractions, 352
  issues, 589–590
  stripping, 54–55
span attribute, 204–205
span method, 205
spawn function, 248
Special characters in regular expressions, 182
Special operations for decimal objects, 332–334
Spheres, plotting, 463–464
Splat operator, 126
split method
  dictionaries, 27
  Money calculator, 342
  regular expressions, 234–236
  strings, 53
Splitting
  charts, 536–537
  strings, 53–54
sqrt function
  math package, 120, 377
  numpy package, 432
Square brackets ([])
  line continuation, 97
  lists, 22, 65, 67
  regular expressions, 183, 185–186, 195–197
Stack class, 317–318
Stacks, lists as, 78–81
Standard deviation
  arrays, 419–424
  normal distribution, 370–371
start attribute for regular expressions, 204
startswith method, 50–51
State information for iterators, 133
State machines for regular expressions, 188–189
Statements reference, 590–603
std function, 420, 422
stderr file, 254
stdin file, 254
stdout file, 254
stock_demo module, 517
stock_load module, 517
Stock-market application
  charts, 521–523, 527–530
  data reader, 519–521
  high and low data, 530–533
  makeplot function, 525–526
  moving-average lines, 538–540
  overview, 517
  pandas package, 518
  review questions, 545
Stock-market application (continued)
  subplots, 536–537
  suggested problems, 545–546
  summary, 544–545
  time periods, 534–536
  titles and legends, 524–525
  user choices, 540–544
stock_plot module, 517
StopIteration exception, 135
str class, 19
str function, 573–574
__str__ method, 153
description, 298
Money class, 341, 349
representation, 295, 297
Strings
  Boolean methods, 48–49
  building, 44–46
  case conversion methods, 49–50
  conversions, 47–48, 161–162
doc, 117–119
  format specifier, 147, 270
  functions, 46–47
  immutability, 33–34
  indexing and slicing, 39–42
  justifying, 55–56
  as lists of characters, 107
  magic methods, 297–298
  multiplication, 104–105
  negative indexes, 39
  numeric conversions, 34–36
  operators, 36–38
  overview, 19–21
  precision, 172–173
  read/write operations, 273–276
  regular expressions, 184
  review questions, 57
  search-and-replace methods, 50–53
  single-character functions, 42–44
  splitting, 53–54
  stripping, 54–55
  suggested problems, 57
  summary, 56–57
  strip method, 54–55
Striping strings, 54–55
struct package, 247–248, 269–271
sub function for replacing text, 211–213
__sub__ method, 296, 305–307
Subclassing, 293–295
Sublists, 64–67
subplot function, 536–537
Subplots in stock-market application, 536–537
Subtraction
  arrays, 407
  distance operator, 307
  magic methods, 305, 313
  operators, 5
  reflection, 311
Sum example, 391–392
sum function
  arrays, 420, 425
  description, 574–575
  lists, 71, 73
  super function, 575
Superclass initialization methods, 294
swapcase method, 49–50
Switch simulation, 109–110
Symbol tables in data dictionaries, 262–265
symmetric_difference method, 580
symmetric_difference_update method, 581
sys package, 138–139, 254
System time for random-number generators, 374

T
Tab characters
  function definitions, 9
  vs. spaces, 589
Tagged groups in regular expressions, 210–211
Tagging in regular expressions, 217–219
tan function
   math package, 376
   numpy package, 432
   tree height calculation, 379–380
tanh function, 377
tau value, 377
tell method, 254, 257–258
tensordot function, 463
Text
   formatting. See Formatting text
      justification fill and align characters, 164–166
      replacing, 211–213
Text files
   vs. binary, 245–246
   description, 245–247
   read/write operations, 252–257
   review questions, 283
   RPN application, 258–268
   suggested problems, 283–284
   summary, 282
Thousands place separator, 152, 168–170
Three-dimensional plotting, 463–464
Ticks in graphs axes, 467–468
time package, 392, 419
Time periods in stock-market application, 534–536
Title characters, testing for, 49
title method
   stock-market application, 524
   strings, 49
Titles for charts, 524–525
Trailing spaces, stripping, 54–55
Tree height calculations, 378–380
Trigonometric functions, 376, 378–380
True keyword
   Boolean operators, 15–16, 107
   while statements, 13
   __truediv__ method, 305
   __trunc__ method, 296, 308–309
Truncation
   magic methods, 308–309
   precision specifier, 170
   strings, 172–173
try/except syntax
   else statements, 106
   exceptions, 250–251
   overview, 600–601
tuple function, 575
Tuples
   assignment, 14–15, 101–104
   dictionary keys, 27
   divmod function, 7
   immutability, 33
   overview, 25–26
Two-module example, 478–482
type function
   description, 575
   object type, 278
   type testing, 320
   variable type, 17
Type specifiers, 173–176
TypeError exception, 34, 43
Types. See Data types
U
   U dtype value, 394
   %u format specifier, 147
   uint8 dtype value, 394
   uint16 dtype value, 394
   uint32 dtype value, 394
   uint64 dtype value, 394
Unary arithmetic operators for magic
   methods, 308–310
Unbalanced matrixes, 91
Underscores (_)
   inside large numbers, 115
   magic methods, 295
   variables, 4, 292, 487–488
UNICODE flag in regular expressions, 193
Unicode values
   character codes, 43
text files, 246
uniform function, 360
union method, 28–29, 581
union_update method, 581
Unnecessary loops, 108
unpack function, 270–271
Unsigned integers format specifier, 147
update method for dict_obj, 586
upper method, 36, 49–50
Uppercase characters
   converting to, 48
testing for, 49

V
   v shape character in plotting lines, 444
ValueError exception
   lists, 60, 74
   strings, 34, 52
Values in dictionaries, 26–27
values method, 586
Variable-length argument lists, 125–128
Variable-length print fields, 150–152
Variable-length string formatting, 274
Variable-size field formatting, 176–178
Variables
   creating, 2
data types, 6
global and local, 29–31
instance, 286–288
lists, 59–61
module-level, 478, 487–488
names, 4
   overview, 587–588
public and private, 292
Variation in random behavior, 361
vdot linear-algebra function, 463
VERBOSE flag in regular expressions, 193
Vowels, testing for, 43

W
   \W character in regular expressions, 194
   \w character in regular expressions, 194
   w color character in plotting lines, 443
   while statements, 12–14, 601
Width
   format function, 154–156
   format specifier, 148–149
   print-field, 163–164
   strings, 172–173
Wildcards in regular expressions, 193
Windows-based systems, command line, 115–116
   with statements
      overview, 601–602
      working with, 252
Word boundaries in regular expressions, 194
Wrapper function definition, 131
Wrapper functions, 128–132
writable method, 253
write_fixed_str function definition, 273
write_floats function definition, 272–273
write method, 253, 255
write_num function definition, 272
write_rec function definition, 275
write_var_str function definition, 274
writelines method, 253, 255

X
   %X format specifier, 147
   %x format specifier, 147–148, 155
   X type specifier, 174–175
   x type specifier, 174–175
   xticks function, 467–468

Y
   y color character, 443
yield statement
   generators, 133–135, 137–138
   overview, 602
   random-number generators, 374–375
   yticks function, 467–468
Z

\z character in regular expressions, 194
zeros function
  description, 393
  overview, 400–401

Zeros, leading-zero character, 167–168
zfll method, 55–56
zip function
  description, 575–576
  lists, 88