BRIAN OVERLAND

PYTHON WITHOUT FEAR

```
x = ['Python','is','cool']
print(' '.join(x))
```

A Beginner's Guide That Makes You Feel SMART

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

Python Without Fear

This page intentionally left blank

Python Without Fear

A Beginner's Guide That Makes You Feel Smart

Brian Overland

Addison-Wesley

Boston • Columbus • Indianapolis • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact international@pearsoned.com.

Visit us on the Web: informit.com/aw

Library of Congress Catalog Number: 2017946292

Copyright © 2018 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit www.pearsoned.com/permissions/.

ISBN-13: 978-0-13-468747-6
ISBN-10: 0-13-468747-7
1 17

For all my beloved four-legged friends:
Skyler, Orlando, Madison, Cleo, and Pogo.

This page intentionally left blank

Contents

Preface xvii

Steering Around the "Gotchas" xvii

How to Think "Pythonically" xvii

Intermediate and Advanced Features xviii

Learning in Many Different Styles xviii

What's Going on "Under the Hood" xviii

Why Python? xix

Acknowledgments xxi

Author Bio xxiii

Chapter 1 *Meet the Python* 1

A Brief History of Python 1

How Python Is Different 2

How This Book Works 3

Installing Python 4

Begin Using Python with IDLE 6

Correcting Mistakes from Within IDLE 6

Dealing with Ends of Lines 7

Additional Help: Online Sources 8

Chapter 2 *A Python Safari: Numbers* 9

Python and Numbers 9

Interlude Why Doesn't C++ Support Infinite Integers? 11

Interlude How Big Is a Google? 13

Python and Floating-Point Numbers 14

Assigning Numbers to Variables 17

Interlude What Do Python Assignments Really Do? 21

Variable-Naming Conventions in This Book 23

Some Python Shortcuts 23

Chapter 2 Summary 26

Chapter 3 *Your First Programs* 29

Temperatures Rising? 29

Interlude Python's Use of Indentation 33

Putting in a Print Message 35

Syntax Summaries 36

Example 3.1. Quadratic Equation as a Function 38

How It Works 39

Getting String Input 41

Getting Numeric Input 43

Example 3.2. Quadratic Formula with I/O 44

How It Works 45

Formatted Output String 46

Example 3.3. Distance Formula in a Script 47

How It Works 48

Chapter 3 Summary 50

Chapter 4 *Decisions and Looping* 53

Decisions Inside a Computer Program 53

Conditional and Boolean Operators 55

The if, elif, and else Keywords 56

Interlude Programs and Robots in *Westworld* 56

Example 4.1. Enter Your Age 59

How It Works 60

while: Looping the Loop 60
Example 4.2. Factorials 63
How It Works 64
Optimizing the Code 65
Example 4.3. Printing Fibonacci Numbers 67
How It Works 69
"Give Me a break" Statement 70
Example 4.4. A Number-Guessing Game 71
How It Works 72
Interlude Binary Searches and "O" Complexity 74
Chapter 4 Summary 75

Chapter 5 *Python Lists* 77

The Python Way: The World Is Made of Collections 77
Processing Lists with for 80
Modifying Elements with for (You Can't!) 82
Example 5.1. A Sorting Application 83
How It Works 84
Optimizing the Code 84
Indexing and Slicing 85
Copying Data to Slices 88
Ranges 89
Example 5.2. Revised Factorial Program 91
How It Works 91
Optimizing the Code 92
Example 5.3. Sieve of Eratosthenes 93
How It Works 94
Optimizing the Code 96
List Functions and the in Keyword 97
Interlude Who Was Eratosthenes? 98
Chapter 5 Summary 99

Chapter 6 *List Comprehension and Enumeration* 101

Indexes and the enumerate Function 101
The Format String Method Revisited 103
Example 6.1. Printing a Table 104
How It Works 105

Simple List Comprehension 106
Example 6.2. Difference Between Squares 109
How It Works 110
Interlude Proving the Equation 111
"Two-Dimensional" List Comprehension 112
List Comprehension with Conditional 114
Example 6.3. Sieve of Eratosthenes 2 115
How It Works 116
Optimizing the Code: Sets 117
Example 6.4. Pythagorean Triples 118
How It Works 119
Interlude The Importance of Pythagoras 120
Chapter 6 Summary 123

Chapter 7 *Python Strings* 125
Creating a String with Quote Marks 125
Indexing and "Slicing" 127
String/Number Conversions 130
Example 7.1. Count Trailing Zeros 131
How It Works 132
Interlude Python Characters vs. Python Strings 135
Stripping for Fun and Profit 135
Example 7.2. Count Zeros, Version 2 137
How It Works 137
Let's Split: The split Method 138
Building Strings with Concatenation (+) 139
Example 7.3. Sort Words on a Line 141
How It Works 142
The join Method 143
Chapter 7 Summary 144

Chapter 8 *Single-Character Ops* 147
Naming Conventions in This Chapter 147
Accessing Individual Characters (A Review) 148
Getting Help with String Methods 148
Testing Uppercase vs. Lowercase 149
Converting Case of Letters 150

Testing for Palindromes 151
Example 8.1. Convert Strings to All Caps 152
How It Works 153
Optimizing the Code 154
Example 8.2. Completing the Palindrome Test 154
How It Works 156
Optimizing the Code 157
Interlude Famous Palindromes 158
Converting to ASCII Code 159
Converting ASCII to Character 160
Example 8.3. Encode Strings 161
How It Works 162
Interlude The Art of Cryptography 164
Example 8.4. Decode Strings 164
How It Works 165
Chapter 8 Summary 166

Chapter 9 *Advanced Function Techniques* 167

Multiple Arguments 167
Returning More Than One Value 168
Interlude Passing and Modifying Lists 170
Example 9.1. Difference and Sum of Two Points 172
How It Works 172
Arguments by Name 173
Default Arguments 174
Example 9.2. Adding Machine 176
How It Works 176
Optimizing the Code 177
Importing Functions from Modules 178
Example 9.3. Dice Game (Craps) 179
How It Works 180
Interlude Casino Odds Making 182
Chapter 9 Summary 185

Chapter 10 *Local and Global Variables* 187

Local Variables, What Are They Good For? 187
Locals vs. Globals 188
Introducing the global Keyword 190

The Python "Local Variable Trap" 190
Interlude Does C++ Have Easier Scope Rules? 191
Example 10.1. Beatles Personality Profile (BPP) 192
How It Works 195
Example 10.2. Roman Numerals 196
How It Works 197
Optimizing the Code 198
Interlude What's Up with Roman Numerals? 200
Example 10.3. Decode Roman Numerals 201
How It Works 202
Optimizing the Code 203
Chapter 10 Summary 204

Chapter 11 *File Ops* 207
Text Files vs. Binary Files 207
The Op System (os) Module 208
Interlude Running on Other Systems 211
Open a File 211
Let's Write a Text File 213
Example 11.1. Write File with Prompt 214
How It Works 214
Read a Text File 216
Files and Exception Handling 217
Interlude Advantages of try/except 219
Example 11.2. Text Read with Line Numbers 220
How It Works 221
Other File Modes 223
Chapter 11 Summary 224

Chapter 12 *Dictionaries and Sets* 227
Why Do We Need Dictionaries, Ms. Librarian? 227
Adding and Changing Key-Value Pairs 229
Accessing Values 230
Searching for Keys 231
Interlude What Explains Dictionary "Magic"? 232
Example 12.1. Personal Phone Book 232
How It Works 234

Converting Dictionaries to Lists 235
Example 12.2. Reading Items by Prefix 236
How It Works 238
Example 12.3. Loading and Saving to a File 238
How It Works 240
All About Sets 241
Operations on Sets 242
Interlude What's So Important About Sets? 244
Example 12.4. Revised Sieve of Eratosthenes 244
How It Works 245
Chapter 12 Summary 246

Chapter 13 *Matrixes: 2-D Lists* 249
Simple Matrixes 249
Accessing Elements 250
Irregular Matrixes and Length of a Row 251
Multiplication (*) and Lists 252
The Python Matrix Problem 253
How to Create N*M Matrixes: The Solution 254
Interlude Why Isn't It Easier? 255
Example 13.1. Multiplication Table 256
How It Works 257
Example 13.2. User-Initialized Matrix 258
How It Works 259
Optimizing the Code 260
How to Rotate a Matrix 261
Interlude Pros and Cons of Garbage Collection 263
Example 13.3. Complete Rotation Example 264
How It Works 266
Optimizing the Code 267
Chapter 13 Summary 268

Chapter 14 *Winning at Tic-Tac-Toe* 271
Design of a Tic-Tac-Toe Board 271
Plan of This Chapter 273
Phase 1 273
Phase 2 273
Phase 3 273

Python One-Line if/else 274
Example 14.1. Simple Two-Player Game 274
How It Works 276
Interlude Variations on Tic-Tac-Toe 279
The count Method for Lists 279
Example 14.2. Two-Player Game with Win Detection 279
How It Works 282
Introducing the Computer Player 285
Example 14.3. Computer Play: The Computer Goes First 287
How It Works 290
Playing Second 291
Interlude The Art of Heuristics 292
Chapter 14 Summary 294

Chapter 15 *Classes and Objects I* 295
What's an Object? 295
Classes in Python 296
How Do I Define a Simple Class? 297
How Do I Use a Class to Create Objects? 297
How Do I Attach Data to Objects? 298
How Do I Write Methods? 300
The All-Important __init__ Method 301
Interlude Why This self Obsession? 302
Design for a Database Class 303
Interlude C++ Classes Compared to Python 304
Example 15.1. Tracking Employees 305
How It Works 307
Defining Other Methods 309
Design for a Point3D Class 310
Point3D Class and Default Arguments 312
Three-Dimensional Tic-Tac-Toe 312
Example 15.2. Looking for a 3-D Win 313
How It Works 314
Example 15.3. Calculating Ways of Winning 315
How It Works 317
Optimizing the Code 317
Chapter 15 Summary 318

Chapter 16 *Classes and Objects II* 321

Getting Help from Doc Strings 321

Function Typing and "Overloading" 323

Interlude What Is Duck Typing? 325

Variable-Length Argument Lists 326

Example 16.1. PointN Class 327

How It Works 329

Optimizing the Code 330

Inheritance 331

The Fraction Class 333

Example 16.2. Extending the Fraction Class 334

How It Works 335

Class Variables and Methods 337

Instance Variables as "Default" Values 339

Example 16.3. Polygon "Automated" Class 340

How It Works 342

Interlude OOPS, What Is It Good For? 343

Chapter 16 Summary 344

Chapter 17 *Conway's Game of Life* 347

Interlude The Impact of "Life" 347

Game of Life: The Rules of the Game 348

Generating the Neighbor Count 350

Design of the Program 352

Example 17.1. The Customized Matrix Class 352

How It Works 353

Moving the Matrix Class to a Module 354

Example 17.2. Printing a Life Matrix 355

How It Works 355

The Famous Slider Pattern 358

Example 17.3. The Whole Game of Life Program 358

How It Works 360

Interlude Does "Life" Create Life? 363

Chapter 17 Summary 364

Chapter 18 *Advanced Pythonic Techniques* 367

Generators 367

Exploiting the Power of Generators 369

Example 18.1. A Custom Random-Number Generator 370

How It Works 372

Interlude How Random Is "Random"? 373

Properties 375

Getter Methods 376

Setter Methods 377

Putting Getters and Setters Together 378

Example 18.2. Multilevel Temperature Object 379

How It Works 380

Decorators: Functions Enclosing Other Functions 382

Python Decoration 385

Example 18.3. Decorators as Debugging Tools 387

How It Works 388

Chapter 18 Summary 389

Appendix A *Python Operator Precedence Table* 391

Appendix B *Summary of Most Important Formatting Rules for Python 3.0* 393

1. Formatting Ordinary Text 393

2. Formatting Arguments 393

3. Specifying Order of Arguments 393

4. Right Justification Within Field of Size N 394

5. Left Justification Within Field of Size N 394

6. Truncation: Limit Size of Print Field 394

7. Combined Truncation and Justification 395

8. Length and Precision of Floating-Point Numbers 395

9. The Padding Character 395

Appendix C *Glossary* 397

Index 407

Preface

There's a lot of free programming instruction out there, and much of it's about Python. So for a book to be worth your while, it's got to be good…it's got to be really, really, *really* good.

I wrote this book because it's the book I wish was around when I was first learning Python a few years back. Like everybody else, I conquered one concept at a time by looking at almost a dozen different books and consulting dozens of web sites.

But this is Python, and *it's not supposed to be difficult!*

The problem is that not all learning is as easy or fast as it should be. And not all books or learning sites are *fun*. You can, for example, go from site to site just trying to find the explanation that really works.

Here's what this book does that I wish I'd had when I started learning.

Steering Around the "Gotchas"

Many things are relatively easy to do in Python, but a few things that ought to be easy are harder than they'd be in other languages. This is especially true if you have any prior background in programming. The "Python way" of doing things is often so different from the approach you'd use in any other language, you can stare at the screen for hours until someone points out the easy solution.

Or you can buy this book.

How to Think "Pythonically"

Closely related to the issue of "gotchas" is the understanding of how to *think* in Python. Until you understand Python's unique way of modeling the world, you might end up writing a program the way a C programmer would. It runs, but it doesn't use any of the features that make Python such a fast development tool.

```
a_list = ['Don\'t', 'do', 'this', 'the' ,'C', 'way']
for x in a_list:
    print(x, end=' ')
```

This little snippet prints

```
Don't do this the C way
```

Intermediate and Advanced Features

Again, although Python is generally easier than other languages, that's not universally true. Some of the important intermediate features of Python are difficult to understand unless well explained. This book pays a lot of attention to intermediate and even advanced features, including list comprehension, generators, multidimensional lists (matrixes), and decorators.

Learning in Many Different Styles

In this book, I present a more varied teaching style than you'll likely find elsewhere. I make heavy use of examples, of course, but sometimes it's the right conceptual figure or analogy that makes all the difference. Or sometimes it's working on exercises that challenge you to do variations on what's just been taught. But all of the book's teaching styles reinforce the same ideas.

What's Going on "Under the Hood"

Although this book is for people who may be new to programming altogether, it also caters to people who want to know how Python works and how it's fundamentally different "under the hood." That is, how does Python carry out the operations internally? If you want more than just a simplistic introduction, this book is for you.

Why Python?

Of course, if you're trying to decide between programming languages, you'll want to know why you should be using Python in the first place.

Python is quickly taking over much of the programming world. There are some things that still require the low-level capabilities of C or C++, but you'll find that Python is a *rapid application development tool*; it multiplies the effort of the programmer. Often, in a few lines of code, you'll be able to do amazing things.

More specifically, a program that might take 100 lines in Python could potentially take 1,000 or 2,000 lines to write in C. You can use Python as "proof of concept": write a Python program in an afternoon to see whether it fulfills the needs of your project; then after you're convinced the program is useful, you can rewrite it in C or C++, if desired, to make more efficient use of computer resources.

With that in mind, I'll hope you'll join me on this fun, exciting, entertaining journey. And remember this:

```
x = ['Python', 'is', 'cool']
print(' '.join(x))
```

Register your copy of *Python Without Fear* on the InformIT site for convenient access to updates and/or corrections as they become available. To start the registration process, go toinformit.com/register and log in or create an account. Enter the product ISBN (9780134687476) 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.

This page intentionally left blank

Acknowledgments

It's customary for authors to write an acknowledgments page, but in this case, there's a particularly good reason for one. There is no chapter in this book that wasn't strongly influenced by one of the collaborators: retired Microsoft programmer (and software development engineer) John Bennett.

John, who has used Python for a number of years—frequently to help implement his own high-level script languages—was particularly helpful in pointing out that this book should showcase "the Python way of doing things." So the book covers not just how to transcribe a Python version of a C++ solution but rather how to take full advantage of Python concepts—that is, how to "think in Python."

I should also note that this book exists largely because of the moral support of two fine acquisition editors: Kim Boedigheimer, who championed the project early on, and Greg Doench, whom she handed the project off to.

Developmental and technical editors Michael Thurston and John Wargo made important suggestions that improved the product. My thanks go to them, as well as the editorial team that so smoothly and cheerfully saw the manuscript through its final phases: Julie Nahil, Kim Wimpsett, Angela Urquhart, and Andrea Archer.

This page intentionally left blank

Author Bio

At one time or another, Brian Overland was in charge of, or at least influential in, documenting all the languages that Microsoft Corporation ever sold: Macro Assembler, FORTRAN, COBOL, Pascal, Visual Basic, C, and C++. Unlike some people, he wrote a lot of code in all these languages. He'd never document a language he couldn't write decent programs in.

For years, he was Microsoft's "go to" man for writing up the use of utilities needed to support new technologies, such as RISC processing, linker extensions, and exception handling.

The Python language first grabbed his attention a few years ago, when he realized that he could write many of his favorite applications—the Game of Life, for example, or a Reverse Polish Notation interpreter—in a smaller space than any computer language he'd ever seen.

When he's not exploring new computer languages, he does a lot of other things, many of them involving writing. He's an enthusiastic reviewer of films and writer of fiction. He's twice been a finalist in the Pacific Northwest Literary Contest.

This page intentionally left blank

3 Your First Programs

Programming is like writing a script, creating a predetermined list of words and actions for actors to perform night after night. A Python function is not so different. From within the interactive environment, you can execute a function as often as you like, and it will execute the same predefined "script." (The term *script* can also refer to an entire program.)

Within the Python interactive development environment (IDLE), writing functions is the beginning of true programming. In this chapter, I explore how to write functions, including the following:

- Using functions to calculate formulas
- Getting string and numeric input
- Writing formatted output

Temperatures Rising?

I happen to live in the Northwest corner of the United States, and I have Canadian relatives. When they discuss the weather, they're always talking Celsius. They might say, "Temperature's all the way up to 25 degrees. Gettin' pretty warm, eh?"

For people accustomed to the Fahrenheit scale, 25 is cold enough to freeze your proverbial hockey stick. So I have to mentally run a conversion.

```
fahr = cels * 1.8 + 32
```

If you have the Python interactive environment running, this is an easy calculation. I can convert 20 degrees in my head, but what about 25? Let's use Python! The following statements assign a value to the name `cels` (a *variable*), use that value to assign another value to the name `fahr`, and then finally display what the `fahr` value is.

```
>>>cels = 25
>>>fahr = cels * 1.8 + 32
>>>fahr
77.0
```

So, 25 "Canadian" degrees are 77.0 degrees on the "real" (that is, the American) temperature scale. That's comfortably warm, isn't it? For those living north of the border, it's practically blistering.

Python prints the answer with a decimal point: 77.0. That's because when the interactive environment combined my input with the floating-point value 1.8, it promoted all the data to floating-point format.

Let's try another one. What is the Fahrenheit value of 32 degrees Celsius? Actually, there's a faster way to do this calculation. We don't have to use variables unless we want to do so.

```
>>>32 * 1.8 + 32.0
89.6
```

Thirty-two degrees on the Celsius scale is 89.6 Fahrenheit. For a Canadian, that's practically burning up.

But I'd like to make this calculation even easier. What I'd really like to do is just enter a function name followed by a value to convert.

```
>>>convert(32)
89.6
```

And—here is the critical part—if this function worked generally, as if it were part of Python, I could use it to convert any number from Celsius to Fahrenheit. All I'd have to do is enter a different argument.

```
>>>convert(10)
50.0
>>>convert(20)
68.0
>>>convert(22.5)
72.5
```

But Python lets me create my own such function. This is what the **def** keyword does: define a new function. We could write it this way from within the interactive environment:

```
>>>def convert(fahr):
    cels = fahr * 1.8 + 32.0
    return cels

>>>
```

Notice that these statements by themselves don't seem to do anything. Actually, they do quite a bit. They associate the symbolic name `convert` with something referred to as a *callable* in Python, that is, a function.

If you display the "value" of the function, by itself, you get a cryptic message.

```
>>>convert
<function convert at 0x1040667b8>
```

This message tells you that `convert` has been successfully associated with a function. There were no syntax errors; however, runtime errors are always possible.

Not until we execute `convert` do we know whether it runs without errors. But this is easy. To execute a function, just follow it with parentheses—enclosing any *arguments*, if any.

```
>>>convert(5)
41.0
```

So, 5 degrees Celsius is actually 41.0 Fahrenheit…cool but not quite freezing.

If you enter this example as shown—using the bold font to indicate what you should type as opposed to what Python prints—and if everything goes right, then congrats, you've just written your first Python function!

If instead you get a syntax error, remember that you can easily edit a function by 1) moving the cursor to any line of the function and 2) pressing Enter. The entire function definition will reappear, and you can edit it by moving the cursor up and down. Finally, you can reenter it again. (To reenter, put your cursor on the end of the last line and press Enter twice.)

Before resubmitting the function definition, review the following rules:

- The definition of `convert` is followed by parentheses and the name of an *argument*. This name stands in for the value to be converted. In this case, the argument name is `fahr`.
- You must type a colon (:) at the end of the first line.
- The environment then automatically indents the next lines. Use this indentation. Don't try to modify it—at least not yet.
- The **`return`** statement determines what value the function produces.
- Remember that in Python all names are case-sensitive.
- In the interactive environment, you terminate the function by typing an extra blank line after you're done.

Note From within the interactive environment, you should use whatever indentations the environment creates for you. Doing otherwise may cause Python to report errors and fail to run the program.

However, when you write Python scripts in separate text files, the preferred convention is to use four spaces (and no tab characters). This is somewhat arbitrary, because almost any indentation scheme works if you hold to it consistently. But four spaces is the style preferred according to the PEP-8 standard that is observed by many Python programmers.

As much as possible, this book tries to hold to this PEP-8 standard. You can read more about this typographic standard for Python programming by searching for *PEP-8* online.

Note

Let's take another example. Let's define another function and this time give it the name `inch_to_cent`. This function is even simpler than the `convert` function: it changes inches to centimeters, according to the formula 1 inch = 2.54 centimeters.

```
>>>def inch_to_cent(inches):
    cent = inches * 2.54
    return cent

>>>
```

As with the earlier function, entering a syntactically correct definition doesn't immediately do anything, but it does create a *callable* that you can then use to perform the inches-to-centimeter conversation whenever you want.

Here's an example:

```
>>>inch_to_cent(10)
25.4
>>>inch_to_cent(7.5)
19.05
```

Note that the `inch_to_cent` function definition uses its own variable—a local variable—named `cent`. Because it is local, it doesn't affect what happens to any variable named `cent` outside of the function.

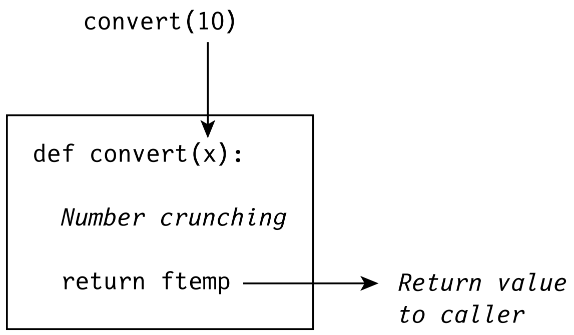
But the use of this variable in this case isn't really necessary. You could define the same function more succinctly, as follows. But the effect is the same in either case.

```
>>>def inch_to_cent(x):
    return x * 2.54

>>>
```

You can conceptualize the action of a function call as follows. Each call to the inch_to_cent function passes a particular value in parentheses. This value is passed to the name x inside the function definition, and the return statement produces the output after operating on the x value passed to it.

Here's an illustration of how this works:



Remember, a function must be defined before a call to that function is executed.

Interlude

Python's Use of Indentation

Syntactically, Python is fundamentally different from all the languages in the C-language family—including C++, Java, and C#—as well as other languages such as BASIC. The single biggest difference is that spacing matters, particularly indentation.

In the interactive environment, Python automatically indents statements inside a control structure, such as a **def**, **if**, or **while** statement block. Until you terminate that block, you should accept the indentation and not try to "fix" it.

When you learn later in this chapter to compose text files as Python scripts, you can indent any number of spaces you want, but you must do it consistently. If the first statement within a block of statements is indented four spaces, the next statement must be indented four spaces as well—no more, no less.

Note that the PEP-8 specification states that four-space indentation is the preferred standard.

A pitfall awaits you in the form of invisible tab characters. You can use tabs, but the danger is that a tab may look like four blank spaces when in fact it is only one character. And if you indent with a tab on one line and use spaces to indent on the next, Python gets confused and issues a syntax error.

▼ *continued on next page*

Interlude

▼ continued

If possible, then, always use either one technique or the other: a single tab or multiple blank spaces. The safest policy is to have your text editor follow the rule of replacing a tab with blank spaces.

Indentation is an area in which C++ programmers are bound to feel superior. Take the following Python function:

```
def  convert_temp(x):
    cels = x * 1.8 + 32.0
    return cels
```

In Python, you must indent this way or Python gets horribly confused. In C and C++, you are freed from spacing issues for the most part, because statement blocks and function definitions are controlled by curly braces. Here's how you might write this function in C++:

```
float convert_temp(float x) {
    float cels = x * 1.8 + 32.0;
    return cels;
}
```

There are similarities between these two versions—the Python and the C/C++ version—but the latter gives you a lot more freedom to space things as you choose.

```
float convert_temp(float x)
{cels = x * 1.8 + 32.0; return cels; }
```

With a little optimization, you can even put all the code on a single line.

```
float convert_temp(float x){return x * 1.8 + 32.0;}
```

What C and C++ programmers tend to like about this is that the compiler is largely indifferent to spacing issues—as long as some whitespace appears where needed to separate variable names and keywords. C++ will never complain because you intended three spaces rather than four, which to a C++ programmer seems fussy, if not petty.

But the Python way has its own advantages. To beginning and intermediate programmers especially, Python indentation allows you to see how "deep" you are in the program. It makes relationships between different statements more obvious. And it closely echoes the indentation of pseudocode I use throughout this book.

Once you get used to Python's reliance on indented statements, you'll love it. Just be careful that your text editor doesn't let you confuse tab characters with blank spaces.

Putting in a Print Message

What if I want the function to not just return a number but to instead print out a user-friendly message such as the following:

```
7.5 inches are equal to 19.05 centimeters.
```

I can easily do that in Python. All I need to do is add a call to the built-in **print** function. Because **print** is a built-in function of Python, it's one that you do not define yourself; it's already defined for you. Here's a sample of this function in action:

```
>>>print('My name is Brian O.')
My name is Brian O.
```

Version Python version 2.0 features a version of **print** that does not expect parentheses around the argument list, because it is not a function. Starting with Python version 3.0, **print** becomes a function and therefore requires the parentheses.

```
print 'My name is Brian O.'  # Python 2.0 version
```

Version

Why did I place single-quotation marks around the message to be printed? I did that because this information is text, not numeric data or Python code; it indicates that the words are to be printed out exactly as shown. Here are some more examples:

```
>>>print('To be or not to be.')
To be or not to be.
>>>print('When we are born, we cry,')
When we are born, we cry,
>>>print('That we are come'
    ' to this great stage of fools.')
That we are come to this great stage of fools.
```

The ability to use **print** pays off in a number of ways: I can intermix text—words placed in quotation marks—with variables.

```
>>>x = 5
>>>y = 25
>>>print('The value of', x, 'squared is', y)
The value of 5 squared is 25
```

By default, the **print** function inserts an extra blank space between one item and the next. Also, after a call to the **print** function is finished, then by default it prints a newline character, which causes the terminal to advance to the next line.

Now let's combine the printing ability with the power to define functions.

```
>>>def convert(x):
    c = x * 2.54
    print(x, 'inches equal', c, 'centimeters.')

>>>convert(5)
5 inches equal 12.2 centimeters.
>>>convert(10)
10 inches equal 25.4 centimeters.
```

Do you now see why the **print** function is useful? I can call this built-in function from within a definition of one of my functions; that enables my functions to print nice output messages rather than just producing a number.

Syntax Summaries

Throughout this book I use summaries to summarize parts of Python syntax. These are the grammatical rules of the language, and—although they are generally easier and more natural than syntax rules for human language—they must be followed precisely. If you're required to use a colon at the end of the line, you must not forget it.

Here is the syntax summary for function definitions:

```
def function_name(argument) :
    indented_statements
```

There actually is more to function syntax than this, as you'll see in Chapters 9 and 10. As I'll show later in this chapter, you can have more than one *argument*; if you do, use commas to separate them.

In a syntax display—such as the one shown previously—items in bold must be typed in as shown; the items in italics are items you supply, such as names.

Here's another example you can compare to the syntax summary:

```
>>>def print_age(n):
    print('Happy birthday.')
    print('I see that you are', n)
    print('years old.')

>>>
```

Remember, as always, that to end the statement block from within the interactive environment, type an extra blank line at the end.

Remember, also, that certain errors are not detected until the function is executed. Suppose a function does not contain syntax errors, but it tries to refer to a variable that is not yet recognized. Executing the function will generate an error unless the variable is created before the function is executed.

For a variable to be recognized, one of several things must happen.

- The function creates a variable by assigning it a value during an assignment (=).
- The variable must already exist because of an earlier assignment.
- Or, the variable exists because it represents a value passed to the function (for example, `n` in the previous function-definition example).

Here's a sample session that executes the `print_age` function. It assumes that this function has already been defined through the use of a **`def`** statement, as shown earlier.

```
>>>print_age(29)
Happy birthday.
I see that you are 29
years old.
```

Here is the same function, this time called with the value 45 rather than 29:

```
>>>print_age(45)
Happy birthday.
I see that you are 45
years old.
```

The built-in **`print`** function has a simple syntax—although there are some special features I'll introduce later.

Key Syntax

```
print(items)
```

When **`print`** is executed, it displays the items on the console, with an extra blank space used to separate one item from the next.

During the call to **`print`**, you use commas to separate arguments if there is more than one.

```
>>>i = 10
>>>j = 5
>>>print(i, 'is greater than', j)
10 is greater than 5
```

Example 3.1. *Quadratic Equation as a Function*

Now let's do something a little more interesting: take the quadratic formula example from Chapter 2 and place it in a function definition, by using the **def** keyword.

The quadratic formula computes the value of x, given the following relationship to arguments a, b, and c.

$0 = ax^2 + bx + c$

The following interactive session defines quad as a function taking three arguments and returning a value, which is the solution for x.

```
>>>def quad(a, b, c):
    determ = (b * b - 4 * a * c) ** 0.5
    x = (-b + determ) / (2 * a)
    return x

>>>
```

With this definition entered into the environment, you can then call the quad function with any values you like. For example, a simple quadratic equation is as follows:

$0 = x^2 + 2x + 1$

In this statement, a, b, and c correspond to the values 1, 2, and 1, respectively. Therefore, by giving the values 1, 2, and 1 as arguments to the quad function, we will get the value of x that satisfies the equation.

```
>>>quad(1, 2, 1)
-1.0
```

This means that we should be able to plug the value –1.0 in for x and get the quadratic equation to come out right. Let's try it.

$$\begin{aligned} 0 &= (-1)^2 + 2(-1) + 1 \\ &= 1 - 2 + 1 \end{aligned}$$

It works! Everything checks out nicely, because plugging –1.0 in for x does indeed produce 0. But a more interesting equation involves the numbers 1, –1, and –1, which give us the golden ratio. That ratio has the following property:

$x/1 = (x + 1)/x,$

That equation, in turn, implies the following:

$x^2 = x + 1$

This in turn yields a quadratic equation, as shown here:

$0 = x^2 - x - 1$

Finally, that gives us values for a, b, and c of 1, –1, and –1, which we can evaluate with the quad function. Let's try it!

```
>>>quad(1, -1, -1)
1.618033988749895
```

And this turns out to be correct to the 15th decimal place. This is the special number "phi." One of its many special properties is phi squared minus 1 produces phi itself. This is the golden ratio.

You can verify it this way:

```
>>>phi = quad(1, -1, -1)
>>>phi
1.618033988749895
>>>phi * phi - 1
1.618033988749895
```

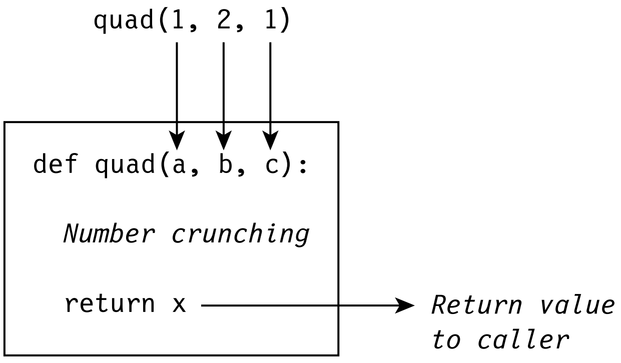
A-ha! Phi squared, minus 1, gives us phi again! This is indeed the golden ratio or, rather, a close approximation of it.

How It Works

Although the quad function may look more complicated than the other, more elementary examples in this chapter, at the bottom it's doing the same thing: taking in some input, doing some number crunching, and returning a result. The one true innovation in this example is that here I've introduced the use of *three* arguments rather than just one.

The order of arguments is significant. Because the quad definition takes three arguments, a, b, and c, each call to quad must specify three values, and these are passed to those variable names: a, b, and c, in that order.

The following illustration shows how this works for the function call quad(1, 2, 1), assigning 1, 2, and 1 to the values a, b, and c:



Now it's simply a matter of doing the correct number crunching to get an answer, and that means applying the quadratic formula.

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

We can use pseudocode to express what this function does. A pseudocode description of a program or function uses sentences that are very close to human language but lists the steps explicitly.

Here is the pseudocode description of the `quad` function:

```
For inputs a, b, and c:
    Set determ equal to the square root of (b * b) – (4 * a * c).
    Set x equal to (–b + determ) divided by (2 * a).
    Return the value x.
```

The quadratic formula actually produces two answers, not one. The plus or minus sign indicates that –b plus the determinant (the quantity under the radical sign) divided by 2a is one answer; but –b *minus* the determinant divided by 2a is the other answer. In Example 3.1, the function returns only the first answer.

EXERCISES

Exercise 3.1.1. Revise the `quad` function by replacing the name `determ` with the name `dt` and by replacing the name `x` with the name `x1`; then verify that the function still works.

Exercise 3.1.2. Revise the `quad` function so that instead of returning a value, it prints two values using the Python **`print`** statement in a user-friendly manner: "The `x1` value is…" and "The `x2` value is…" (Hint: The use of the plus/minus sign in the quadratic formula indicates what these two—not one—values should be. Review this formula closely if you need to do so.) Print each answer out on a separate line.

Exercise 3.1.3. The mathematical number "phi" represents the golden ratio, more specifically, the ratio of the long side of a golden rectangle to the short side. Try to predict what the reciprocal (1/phi) is; then use the Python interactive environment to see whether you're right. How would you express the relationship between phi and 1/phi?

Getting String Input

Before you can finally write "real programs," you'll need to be able to write scripts that can query the user for more information. Fortunately, Python includes a powerful built-in function, the **input** function, which makes this easy.

I'll give you the syntax first and then show examples.

Key Syntax

```
string_var = input(prompt_string)
```

Version ▶ If you're using Python 2.0, use the function name **raw_input** instead of **input**. In 2.0, the **input** function works, but it does something different: it evaluates the string input as a Python statement rather than just passing it back as a string.

◀ Version

The essence of this syntax is that the built-in **input** statement both takes and produces a text string. In one way, the concept of text string is easy to understand; it's just "words" for the most part—or more accurately, letters and other characters.

For example, we might write a function, `main`, which we're going to use as a script.

```
>>>def main():
    name1_str = input('Enter your name: ')
    name2_str = input('Enter another: ')
    name3_str = input('And another: ')
    print('Here are all the candidates: ')
    print(name1_str, name2_str, name3_str)

>>>main()
Enter your name: Brian
Enter another: Hillary
And another: Donald
Here are all the candidates:
Brian Hillary Donald
```

This by itself is not a very exciting program. It takes some text and displays it on the console. But this little program demonstrates an important ability of Python: the ability to prompt the user for a text string and then assign it to a variable.

While other variables up until now have referred to numeric values, these variables—name1, name2, and name3—all refer to text strings in this case.

What exactly can go into a text string? Basically, anything you can type can go in a text string. Here's an example:

```
>>>in_str = input('Enter input line: ')
Enter input line: When I'm 64...
>>>in_str
'When I'm 64...'
```

As you can see, text strings can contain numerals (digit characters). But until they're converted, they're just numerals. They are text-string representations of numbers, not numbers you can perform arithmetic on.

If this isn't obvious, just remember that the *numeral* 5 is just a character on a keyboard or on the screen. But the *number* 5 can be doubled or tripled to produce 10 or 15 and has little to do with characters on a keyboard.

Here's an example:

```
in_str = '55'
```

But assigning 55 with no quote marks around it does something different.

```
n = 55
```

The difference is that 55 is an actual number, meaning that you can add, subtract, multiply, and divide it. But when enclosed in quotation marks, '55' is a text string. That means it is a string consisting of two numerals, each a 5, strung together.

A simple program should help illustrate the difference.

```
>>>def main():
    in_str = input('Enter your age: ')
    print ('Next year you'll be', in_str + 1)

>>>main()
Enter your age: 29
Error! Incompatible types.
```

Oops! What happened? The characters 29 were entered at the prompt and stored as a text string, that is, a numeral 2 followed by a numeral 9—a string two characters long. But that's not the same as a number, even though it looks like one.

Python complains as soon as you try to add a text string to a number.

```
in_str + 1
```

No error is reported until you execute the function. Python variables don't have types; only data objects do. Consequently, Python syntax seems lax at first. But the types of data objects—which are not checked for syntax errors, as there are no "declarations"—are checked whenever a Python statement is actually executed.

This means, among other things, that you cannot perform arithmetic on a string of numerals such as 100, until that string is first converted to numeric format. If this doesn't make sense now, don't worry; it will make sense when you read the next section.

The next section shows how to get input and store it as a number rather than text string.

Getting Numeric Input

As the previous section demonstrated, if you write a program that takes numeric input and does number crunching on it, you need to first convert to a numeric format.

To do that, use one of the following statements, depending on whether you are dealing with integer (**int**) or floating-point data (**float**):

Key Syntax

```
var_name = int(input(prompt_message))
var_name = float(input(prompt_message))
```

These statements combine the input and conversion operations. You can, if you prefer, do them separately, but this is less efficient. For example, you could use this approach:

```
in_str = input('Enter a number: ')
n = int(in_str)
```

These two statements work fine together, but there's no reason not to combine the operations.

```
n = int(input('Enter a number: '))
```

Here's an interactive session that uses a number entered at the keyboard and then multiplies it by 2:

```
>>>def main():
    n = int(input('Enter a number: '))
    print('Twice of that is:', n * 2)

>>>main()
Enter a number: 10
Twice of that is: 20
```

So, to get actual numeric input, as opposed to storing input in a text string, use the **int** and **float** conversions.

But what are **int** and **float**, exactly? Here I'm using them like functions, but they're actually the names of built-in data types, integer and floating point, respectively. In Python, there's a general rule that type names can be used in this fashion, to perform conversions (assuming the appropriate conversion exists).

Key Syntax

```
type_name(data)
```

Example 3.2. *Quadratic Formula with I/O*

This next example takes the quadratic-formula example another step further, by placing all the statements in a `main` function and then relying on Python input and output statements to communicate with the end user.

```
>>>def main():
    a = float(input('Enter value for a: '))
    b = float(input('Enter value for b: '))
    c = float(input('Enter value for c: '))
    determ = (b * b - 4 * a * c) ** 0.5
    x1 = (-b + determ) / (2 * a)
    x2 = (-b - determ) / (2 * a)
    print('Answers are', x1, 'and', x2)

>>>main()
```

Here is a sample session that might follow after you type **main()**:

```
Enter value for a: 1
Enter value for b: -1
Enter value for c: -1
Answers are 1.618033988749895 and -0.6180339887498948
```

There are two different answers in this case, not equal to each other, because the golden ratio is either phi (the ratio of the large side to the small) or 1/phi (the ratio of the small side to the large), depending on how you look at it. The negative sign in the second answer is necessary for the math to come out right.

Nearly all the digits are identical in this case, except for a small difference due to rounding errors. The actual values of phi and 1/phi are irrational (which means you would need an infinite number of digits to represent them precisely).

Note Remember that the interactive environment supports cut-and-paste operations, as well as a "magic" technique for revising blocks of code.

So, if you enter a long function definition and realize you've made a mistake, you can save a great deal of time by doing the following:

1. Scroll up to the block of code.
2. Place your cursor on any line of code in this block.
3. Press Enter.
4. The block of code will appear in the window—at the new cursor position—ready for you to edit it.

Then go ahead and make your changes, scrolling up and down if you need. When done, type an extra blank line after the new block of code.

◀ Note

How It Works

In this chapter, we're still dealing with programs that are relatively short and translate into simple pseudocode.

```
Prompt the user for the values of a, b, and c.
Apply the quadratic formula to get x1 and x2.
Print the values of x1 and x2.
```

Because `a`, `b`, and `c` all need to refer to numeric data, the program applies a **`float`** conversion combined with the built-in **`input`** function. If these numbers are not converted to **`float`** format, you won't be able to do math with them.

```
a = float(input('Enter value for a: '))
b = float(input('Enter value for b: '))
c = float(input('Enter value for c: '))
```

Next, the quadratic formula is applied to get the two solutions for `x`. Remember that the operation `** 0.5` has the same effect as taking the square root.

```
determ = (b * b – 4 * a * c) ** 0.5
x1 = (-b + determ) / (2 * a)
x2 = (-b - determ) / (2 * a)
```

Finally, the program displays the output, featuring `x1` and `x2`.

```
print('The answers are', x1, 'and', x2)
```

EXERCISES

Exercise 3.2.1. In Example 3.2, instead of using the prompt messages "Enter the value of a," etc., prompt the user with the following messages:

"Enter the value of the x-square coefficient."

"Enter the value of the x coefficient."

"Enter the value of the constant."

Do you have to change the variable names as a result? Note that the user never sees the names of variables inside the code, unless you deliberately print those names.

Exercise 3.2.2. Modify Example 3.2 so that it restricts input to integer values.

Exercise 3.2.3. Write a program to calculate the area of a right triangle, based on height and width inputs. Apply the triangle area formula: A = w * h * 0.5. Prompt for width and height separately and print a nice message on the display saying, "The area of the triangle is…."

Exercise 3.2.4. Do the same for producing the volume of a sphere based on the radius of the sphere. I'll invite you to look up the formula for volume of a sphere. For the value pi, you can insert the following statement into your program:

```
pi = 3.14159265
```

Formatted Output String

In Example 3.2, typical output looked like this:

```
The answers are 3.0 and 4.0
```

(This is produced, incidentally, when the inputs to the quadratic formula are 1, –7, and 12.)

But we might like to place a period at the end, making the output read as a nice sentence. We'd like to get the following:

```
The answers are 3.0 and 4.0.
```

But the **print** function puts a space between each print field so that you end up getting the following, which has an unnecessary space before the last character.

```
The answers are 3.0 and 4.0 .
```

There are at least two solutions. One is to include the special **sep** (separator) argument to the **print** function. By default, print uses a single space as a separator. But we can use **sep=''** (this consists of two single quotes in a row) to indicate that **print** shouldn't put in any separator at all.

This is fine, because we just take on responsibility for putting in space separators ourselves. The output statement then becomes the following:

```
print('The answers are ', x1, ' and ', x2, '.', sep='')
```

And this works, although it's a fair amount of extra work. Not only do we have to add **sep=''**, but we have to add all those extra spaces.

But there's a better way. Python provides a way to create a formatted-output string. To use this approach, follow these steps:

1 Create a format specification string that includes print fields denoted with the characters {}. A print field is an indication of where output characters, produced by arguments, are to be inserted into the resulting string.

2 Apply the **format** method to this format-specification string, specifying the values to be printed as arguments.

3 Print the resulting output string.

For example, you can set up a format specification string (fss) as follows:

```
fss = 'The numbers are {} and {}.'
```

Then you apply the **format** method to this string. The result produces an output string.

```
output_str = fss.format(10, 20)
print(output_str)
```

And here's what the result looks like:

```
The numbers are 10 and 20.
```

Example 3.3. *Distance Formula in a Script*

Sooner or later, you'll want to write and permanently save your Python programs. The steps are as follows:

1 From within IDLE, choose the New File command from the File menu.

2 Enter a program (or copy text) into the window that appears, which serves as a text editor for your programs.

3 To save the program, choose Save or Save As from the File menu. The first time you save a program this way, the environment will prompt you to enter a name with a `.py` extension. (It will add this extension for you automatically.)

4 To run the program, make sure the program window has the focus. Then either press F5 or select Run Module from the Run menu.

5 After the program begins running, you may need to shift focus back to IDLE's main window (the *shell*). [An exception is that with tkinter (graphical) programs, you'll need to shift focus to the window *generated* by the program.]

Alternatively, you can write a program with any text editor you want, but be sure you save the file in plain-text format and give it a `.py` extension. Then you can load it into Python by using the Open command from IDLE's File menu.

Although the Python environment is still extremely useful for experimenting with, and getting help with, individual commands and features, the text-editor approach is usually better for writing and executing long programs.

This next example shows how to use the Pythagorean distance formula to calculate the distance between any two points on a Cartesian plane. Here's the formula:

$$distance = \mathbf{square\_root(}horizontal\_dist^2 + vertical\_dist^2\mathbf{)}$$

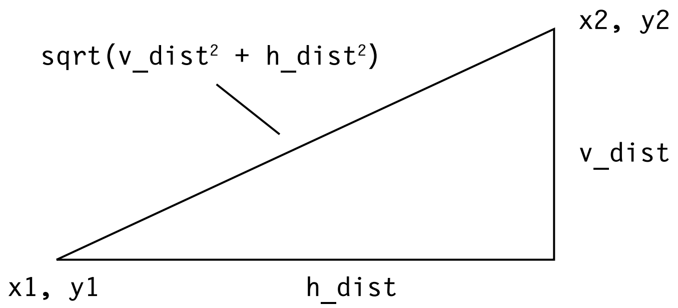
Here's the program listing:

dist.py

```
x1 = float(input('Enter x1: '))
y1 = float(input('Enter y1: '))
x2 = float(input('Enter x2: '))
y2 = float(input('Enter y2: '))
h_dist = x2 - x1
v_dist = y2 - y1
dist = (h_dist ** 2 + v_dist ** 2) ** 0.5
print('The distance is ', dist)
```

How It Works

The Pythagorean distance formula is derived from the Pythagorean theorem, which I'll have more to say about in Chapter 6. By applying this theorem, you can see that the distance between two points is equivalent to the hypotenuse of a right triangle, in which the vertical distance (`v_dist`) and horizontal distance (`h_dist`) are the two other sides.



The square of the hypotenuse is equal to the sums of the squares of the other sides. Therefore, the hypotenuse itself is equal to the square root of this sum. (See the figure.)

Remember that the exponentiation operator in Python is **. Therefore, the following

```
amount ** 2
```

means to produce the square of `amount` (multiply itself by itself), whereas this next expression

```
b ** 0.5
```

is equivalent to taking the square root of `b`. Therefore, the distance formula is

```
dist = (h_dist ** 2 + v_dist ** 2) ** 0.5
```

EXERCISES

Exercise 3.3.1. As I just mentioned, the syntax `x ** 2` translates as `x` to the second power, in other words, `x` squared. There is another, slightly more verbose, way of expressing the same operation. Revise Example 3.3 so that it uses this other means of calculating a square. Also, replace `h_dist`, `v_dist`, and `dist` in the program with `h`, `v`, and `d`. Then rerun and make sure everything works. For example, if you input the points 0, 0 and 3, 4, the program should say that the distance between the points is 5.0.

Exercise 3.3.2. Revise Example 3.3 so that it outputs the result and puts a period (.) at the end of the sentence, without any superfluous blank spaces. Use the format-specification-string technique I outlined in the previous section.

Exercise 3.3.3. Write a program that calculates the area of a triangle after prompting for the values of the triangle's height and width. Use the formula height * width * 0.5. Use the format-specification-string technique to print a period at the end of the output.

Exercise 3.3.4. Write a program that calculates the area of a circle after prompting for the value of the radius. (I'll leave it to you to look up the formula for area of a circle if you don't remember it.) Use the format-specification-string technique to print a period at the end of the output. Also, to get the value of pi, place the following statement at the beginning of your program:

```
from math import pi
```

With this statement at the beginning of your program, you can use `pi` to refer to a good approximation of pi.

Chapter 3 *Summary*

Here are the main points of Chapter 3:

- A function definition lets you perform a series of calculations over and over, without having to reenter all the steps in number crunching. At least this is a simple way to understand the concept.
- The syntax of a function definition has this form:

```
def function_name(arguments):
    indented_statements
```

- The `arguments` may be blank, may have one argument name, or may be a series of argument names separated by commas.
- If you enter the function-definition heading correctly, the Python interactive environment automatically creates indentation. Remember that a correct function-definition heading ends with a colon (:).
- From within the Python interactive environment, you complete a function definition by typing an extra blank line after you've entered all the indented statements.
- To call a function, enter the name of the function followed by parentheses and argument values. These values are then passed to the function-definition code. Here's an example:

```
>>>convert(10)
```

- You can prompt the user for string input by using the **`input`** statement. The prompt message is a string printed on the console to prompt the user for input.

```
string_var = input(prompt_message)
```

- To get numeric input, use an **`int`** or **`float`** conversion, as appropriate.

```
var = int(input(prompt_message))
var = float(input(prompt_message))
```

- The built-in **`print`** function prints all its arguments in order. By default, arguments are printed with a single blank space separating them. You can use the optional **`sep`** argument to specify another separator character. **`sep=''`** specifies that no separator character should be used.
- You can use a format-specification string, in which {} indicates a print field. Here's an example:

```
fss = 'The square root of {} is {}.'
```

- You can then apply the format method to a format specification string to produce an output string.

```
format_spec_string.format(arguments)
```

- Here's an example:

```
fss.format(25, 5)
```

This page intentionally left blank

Index

Numbers

O(log n) notation, binary searches, 74–75
1/phi number, as golden ratio, 44–46
2D list comprehension, 112–114
2D lists. *See* Matrixes (2D lists)
3D Tic-Tac-Toe game
- calculating ways of winning, 315–318
- looking for win, 313–315
- overview of, 312–313

A

ABC language, Python derived from, 1
__add__ method
- extending Fraction class, 335–336
- as magic method, 402
- PointN class, 328–330

add method, sets, 243, 247
add_entries, personal phone book, 233, 235
Addition operator (+), shortcut with assignment and, 24–25
Advanced features
- decorators, 382–389
- generators, 367–375
- properties, 375–382
- summary, 389–390

Aliases, Python variables as, 79
amt variable, 197–201
and operator
- Boolean, 55
- counting trailing zeros, 133

append method
- modifying elements, 83–84
- optimizing code, 96–97
- preparing for join, 154
- sorting information with lists, 77–78

*args
- customized Matrix class for Game of Life, 353–354
- decorators, 386–387, 389
- defined, 326
- random-number generator, 371–372
- randomization via system time, 374
- variable-length argument lists, 326–327

Arguments
- calling by name, 173–174, 175
- calling functions with incompatible, 171
- customized Matrix class for Game of Life, 353–354
- default, 174–178
- defining functions with, 31, 50, 185
- method, 309–310
- ordering in functions, 39–40, 393
- passing and modifying lists, 170–171
- passing multiple, 36, 37–40, 167–168

Arguments (*continued*)
  rules for formatting, 393
  slicing, 157
  variable-length argument lists, 326–327
ASCII code
  converting to, 159–160
  converting to characters, 160–163
  defined, 159
  translating individual characters into, 148
`ask_q` function, global variables, 194, 196
Assignment operator (=)
  creates variable if one does not exist, 189, 191, 204
  passing/modifying lists and direct, 171
  rule for Python variables, 17–18
  shortcut using addition with, 24–25
  strict syntax for, 19
Assignment-concatenation operator (+=), 24–26, 27, 140
Attributes
  in class inheritance, 331–333
  definition of, 397
  displaying name of function, 389, 390
  getter and setter methods, 376

B

Backslash (\)
  continue physical lines, 7–8
  embedded quotation marks, 127
Base class
  creating Polygon "automated" class, 340–342
  definition of, 397
  extending `Fraction` class, 336–337
  realizing inheritance via subclassing, 331, 344, 345
Basic interactive mode, Python in IDLE vs., 4–6
Beatles personality profile, 192–196
Behavior, methods and, 300
Big Bang, 14
Big Blue, IBM super-chess playing program, 293
Big-O notation, 74
Binary files, text files vs., 207–208, 224
Binary numbers, 74–75, 159
Birth events, Game of Life, 349–350, 360–361
Bitwise operations, 391, 397
Blinker pattern, Game of Life, 349–350
Blocking, computer player heuristics, 285, 287
Boolean
  adding machine application, 176–178
  definition of, 397
  list, sieve of Eratosthenes, 93–97
  operators, 55
  testing character case, 149–150
`break` keyword
  breaking from loop, 70–71
  breaking out of infinite loop, 72, 75
  counting trailing zeros, 134
Brute force (look-ahead), heuristics, 293
`b_str` variable, processing lists with `for`, 80–82
Bucket, dictionary key-value access, 232

C

C programming language
  Python characters/strings vs., 135
  Python indentation vs., 33–34
  Python local/global variables syntax vs., 190
C++ programming language
  non-support for infinite integers, 11–12
  Python characters/strings vs., 135
  Python indentation vs., 33–34
  Python local/global variables syntax vs., 190
C++ Without Fear (Overland), 12

`calc_pts` function, difference/sum of two points, 172–173
Calculator, Python as one-line, 10
Callable, 31–32, 397–398
Cartesian plane, Pythagorean distance formula, 48–50
Case
 converting letters, 150–151
 testing upper and lower, 149–150
Case-sensitivity
 keywords, 27
 names, 18–19, 27
Casino odds making, 179–184
Cellular automata, defined, 348, 364
Celsius, 29–33, 82, 379–381
Characters
 accessing individual, 148
 format for padding, 395
 join to form single string, 143–144
 as one-length strings, 148
 Python strings vs., 135
 single. *See* Single-character ops
`chars` string, defined, 136
`chdir` function, `os` module, 209
`chr` function, convert ASCII to character, 160–161, 166
`Circle` subclass, 339–343
Class definition
 creating objects with classes, 297
 methods as functions defined inside, 300
 moving into separate modules, 354–358
 optimizing code with, 317–318
 realizing inheritance via subclassing, 331, 345
 references to instance variables within, 319
 writing for database class, 303, 307
 writing for Point 3D class, 311
Class hierarchy, inheritance, 332–333
`class` keyword, 297
Class variables
 define default values for instance variables via, 339–343
 definition of, 398
 inheritance of, 331
 methods and, 337–338
 summary, 345
Classes
 attaching data to objects, 298–300
 C++ vs. Python, 304
 creating objects with, 297–298
 defining simple, 297
 definition of, 398
 designing database, 303–308
 getting help from your own, 322–323
 overview of, 296
 writing methods at level of, 300–301
Classes and objects. *See also* Classes
 `__init__` method, 301–302
 3D Tic-Tac-Toe, 312–318
 benefits of OOP, 343–344
 class variables and methods, 337–338
 database class design, 303–308
 defining other methods, 309–310
 definition of object, 295–296
 duck typing, 325
 `Fraction` class, 334–337
 function typing/overloading, 323–324
 help from doc strings, 321–323
 inheritance, 331–333
 instance variables as default values, 339–343
 obsession with `self`, 302–303
 overview of, 295
 `Point3D` class/default arguments, 312
 `Point3D` class design, 310–311
 summary, 318–319, 344–345
 variable-length argument lists, 326–327
 writing `PointN` class, 327–331

`close` method
    after opening file, 212
    after reading text file, 217
    loading and saving to file, 240
    writing file with prompt, 215
Collections, 101, 398
Colon (:)
    at end of first line of `if`, 54
    at end of first line of `while`, 61
    function definition syntax, 50
    rules for functions, 31
Colons (::), omitted second arguments, 128
Commas (,)
    joining group of characters into single string, 143–144
    returning multiple function values, 168–173
    separating multiple function arguments, 36–40, 167–168
Comments (#), 61–62, 398
Comparison operators, vs. Boolean, 55
Comparisons
    palindrome test, 154–158
    testing character case, 149–150
    testing for palindromes, 151–154
Composite numbers
    all multiples of prime as, 96
    defined, 93
    optimizing sieve of Eratosthenes code, 97, 115–117
    prime numbers vs., 93
Computer player, Tic-Tac-Toe, 287–292
Concatenation (+) operator, building up strings, 140–141
Condition, defined, 54
Conditional
    list comprehension with, 114–116
    operators, 55
`conditional_expr`, list comprehension, 115
Consciousness, computer, 56–58
Control structure
    automatic indentation inside, 33
    conditional and Boolean operators as, 55
    as decision-making, 53
    decisions inside computer program, 53
    definition of, 398
    `if`, `elif`, and `else` keywords as, 56, 59–60
    processing lists with `for`, 80–82
    programs/robots in *Westworld* and, 56–58
    `while` keyword as, 60–63
    writing exception handling, 218–219
`convert` function
    ASCII code to characters, 30–33
    in print message, 36
    Python indentation and, 33–34
Converting
    to 0-based indexes, 278, 281, 294
    ASCII code to characters, 160–163
    Celsius to Fahrenheit, 29–33, 82
    characters to ASCII code, 159–160
    encoding strings, 162
    factorial to string, 132–133, 137
    numbers into Roman numerals, 196–199
    to numeric format, 43, 131
    Roman numerals into numbers, 201–203
    string input to integer, 324
    strings to all caps, 152–154
    between uppercase/lowercase letters, 150–151
Conway, John Horton, 347–348
Conway's Game of Life
    customizing `Matrix` class, 352–354
    design of program, 352
    full program for, 358–362
    generating neighbor counts, 350–351
    impact of, 347–348

moving `Matrix` class to module, 354–358
overview of, 347
printing Life matrix, 355–358
rules of, 348–350
simulating our reality with, 363–364
slider pattern, 358
summary, 364–365
Copying data to slices, 88–89
`count` method, for lists, 275–278
Count trailing zeros, 131–134, 136–137
Craps (dice game), 179–184
`cry` function, 301
Cryptography
art of, 164
converting ASCII to characters, 160–163
converting characters to ASCII code, 159–160
decoding strings, 164–166
`ctmp` property, multilevel temperature object, 379–381
Curly braces { }
dealing with ends of lines, 7
format string method, 103
print fields, 47, 51
specifying sets, 241–242
Custom random-number generator, 370–373

D

Data
attaching to objects, 298–300
function objects vs., 382
Data types
choosing for dictionary keys, 232
and dictionary magic, 232
as objects in Python, 295–296
testing before using, 323–324
Database class design, 303–308
Death events, Game of Life, 349–350, 360–361
Debugging tools, decorators as, 387–389
Decimal number system, computers, 159
Decimal point (.), 15–16, 27
Decision-making and looping
binary searches and "0" complexity, 74–75
breaking from loop, 70–71
conditional and Boolean operators, 55
decisions inside computer programs, 53–55
factorials, 63–67
`if`, `elif`, and `else`, 56, 59–60
looping with `while`, 60–63
number-guessing game, 71–74
overview of, 53
printing Fibonacci numbers, 67–70
programs and robots in *Westworld*, 56–58
using factorials, 63–65
Decoding
Roman numerals, 201–204
strings, 164–166
Decorators
as debugging tools, 387–389
definition of, 398
exercises, 389
as functions enclosing other functions, 382–385
in Python, 385–387
summary, 390
what they do, 387
Decryption
art of cryptography, 164
converting ASCII to character, 160–163
converting to ASCII code, 159–160
of strings, 164–166
`def` keyword
adding call to `print` function, 35–36
defining functions with, 30–31, 32–33, 400
quadratic equation as function, 38–40
syntax summaries for function definitions, 35–36

Default arguments, 174–178, 185
Delimiters, `split` method for strings, 138–139
Descendant classes, inheritance in, 333
Design
  program, 352
  program for Game of Life, 352–354
  Tic-Tac-Toe board, 271–273
`diagnostics` decorator, as debugging tool, 387–389
Dice game (craps), 179–184
`dict` type, defined, 235–236
Dictionaries
  accessing values, 230
  adding/changing key-value pairs, 229
  converting to lists, 235–236
  creating personal phone book, 232–235
  definition of, 399
  design Employee database as empty, 307
  loading and saving to file, 238–241
  magic of, 232
  overview of, 227
  reading items by prefix, 236–238
  searching for keys, 231–232
  and sets. *See* Sets
  why we need, 227–228
`dict.items`, 235–238
Difference between squares, 109–112, 124
`difference` method, sets, 244
Directories, `os` module, 209–211
Disk files. *See* Files
Distance formula, scripts, 47–48
Division
  operators, 26
  remainder operator. *See* Remainder operator (%)
  storing fractional result as floating-point number, 14–16
Doc strings
  customized `Matrix` class for Game of Life, 352–354
  getting help from, 321–322
  syntactic rules, 322–323
Documentation, drawback of OOP in Python, 344
`do_generation` function, Game of Life, 358–360
Dot notation (.)
  accessing instance variables/methods, 295, 299, 319
  call another class's version of a function, 333
Double backslash (\\), quoted strings, 7–8
Double forward slash (//), integer division, 155
Double threat, computer player heuristics, 286–287
Double underscores (__), 303–304, 377, 379
Downloads menu, installing Python, 4
Duck typing, 325
Duplicate values, lists, 78

E

Edge problem, Game of Life, 360–361
Editing of functions, 31
Einstein, using Pythagorean theory, 122
Elements
  accessing matrix, 250–251
  adding/removing, 241
  creating matrixes, 249–250
  in sets, 241
`elif` keyword
  in control structures, 56
  encoding strings, 161–162
  summary, 75
`else` keyword
  in control structures, 56
  `if/else` operator, 274–278
  simple program using `if`, 59–60
  summary, 75
Embedded quotation marks, text strings, 126–127, 144
Employee class, creating, 307
Employee database, designing, 305–308

Empty sets, creating, 242
Empty strings, 151–153
Encryption
 art of cryptography, 164
 converting ASCII to character, 160–163
 converting to ASCII code, 159–160
 decoding strings, 164–166
`end` argument
 adding to `print` function, 62
 changing list elements from inside loop, 90, 92
 named arguments, 174
 slicing strings, 145
ENIGMA code, 58, 164
`enter_mat_vals` function, rotating matrix, 264–267
`enumerate` function
 indexes, 101–103
 printing tables, 104–106
 reading text file with line numbers, 222
 `for` statement used with, 103, 106, 123
 summary, 123
 two-dimensional list comprehension, 114
 user-initialized matrix, 259–260
Enumeration
 format string method, 103–104
 indexes and, 101–103
 overview of, 101
 printing tables, 104–106
`__eq__` method, `Point3D` class, 310–311, 313, 316
Equivalency, testing in data dictionaries, 228
Eratosthenes
 history of, 98–99
 sieve of. *See* Sieve of Eratosthenes
Error correction, within IDLE, 6–7
Exception handling
 calling function with incompatible argument types, 171
 files and, 217–220
 try/except block advantages, 219–220
Exercises, in this book, 4
Exponent operator (**)
 applied before addition, 11
 exercises, 13
 handling super-large numbers, 10–11
 Pythagorean distance formula, 49
 summary, 26
Expressions
 building complex, 19–20
 how Python resolves, 340

F

Factorials
 changing list elements from inside loop, 91–93
 count trailing zeros, 131–134
 optimizing code, 65–67
 writing loops with `while`, 63–65
Fahrenheit, 29–33, 82, 379–381
`False` value
 adding machine, 176–178
 computer program decisions and, 54–55
 dice game (craps), 181
 `in` keyword, 196
 palindrome test, 154–158
 reading text file with line numbers, 222
 searching dictionary keys, 231
 sieve of Eratosthenes, 93–97
 sorting lists, 84–85
 test-for-equality (==) and, 11
 testing character case, 149–150
Fibonacci numbers
 exploiting power of generators, 369–370
 Hindu-Arabic numbering system and, 200
 printing, 67–70
 working with generators, 368
File pointers, resetting, 217, 225
`FileNotFoundError`, 217–220, 225

Files
    exception handling, 217–220
    loading and saving to, 238–241
    opening and closing, 211–212
    `os` module, 208–210
    other modes of, 223
    overview of, 207
    reading text, 216–217
    reading text, with line numbers, 220–223
    running on other systems, 211
    summary, 224
    text vs. binary, 207–208
    writing text, 213–216
First-class objects, 399
`float` numeric type
    convert strings to numeric, 131
    forcing division to be precise, 15
    getting numeric input, 43–44
    as object in Python, 295–296
    in quadratic formula with I/O, 45
    summary, 51
Floating-point numbers
    definition of, 399
    getting numeric input, 43–44
    Python and, 10, 14–16
    rule for length/precision, 395
    variable naming conventions, 23
`for` loop
    casino odds making, 184
    combine generator with, 369
    definition of, 399
    `load_file` function, 240
    Pythagorean triples, 119–120
`for` statement
    accessing dictionary values, 230
    end specification in, 92
    `enumerate` function with, 103, 106, 123
    list comprehension, 106–111, 112–116, 123–124
    printing out entire dictionary contents, 230
    processing lists, 81–84
    reading text file, 216–217
    replace calls to `make_roman` function with, 199
    sieve of Eratosthenes example, 94–96
`format` string method
    creating matrix for multiplication table, 256–257
    difference between squares, 110
    format fields of output, 47, 51
    printing table, 104–106
    revisited, 103–104
    summary, 123
`format_spec_str.`**`format(`**`args`**`)`**, 103–104
Formatting rules, important
    arguments, 393
    left justification for non-numeric values, 394
    ordinary text, 393–395
    right justification for numeric values, 394
    specify order of arguments, 393
    truncation (limit size of print field), 394
`Fraction` class, 333–337
`ftmp` property, multilevel temperature object, 379–381
`func` function, decorators, 383–387
Functions
    adding call to built-in, 35–36
    creating with `def`, 30–33
    decorator. *See* Decorators
    definition of, 400
    editing, 31
    executing new, 31
    generator. *See* Generators
    getting string input, 41–43
    global vs. local variable rules for, 189–190
    list, 97–98
    methods vs., 295
    naming conventions, 23
    order of arguments, 39–40
    passing back multiple values, 176–177

passing back None, 177–178
pseudocode descriptions of, 40
quadratic equations as, 38–40
summary, 50–51
syntax summaries for, 36–37
typing and overloading, 323–324
wrapper, 406
writing first Python, 30–31
Functions, advanced
arguments by name, 173–174
casino odds making, 182–184
default arguments, 174–178
importing functions from modules, 178–182
multiple arguments, 167–168
overview of, 167
passing and modifying lists, 170–171
returning more than one value, 168–173
summary, 185–186

G

Game of Life. *See* Conway's Game of Life
Garbage collection, 262–264, 269
Gardner, Martin, 347–348
Generators
about randomness, 372–375
custom random-number, 370–372
definition of, 400
exercises, 372
exploiting power of, 369–370
overview of, 367–368
summary, 389–390
`gen_fibo` function, 368, 369–370
`gen_odd_num` iterable, 369
`get` method
converting dictionaries to lists, 236
personal phone book, 234–235
searching for dictionary keys, 231
`get_area()` method, instance variables, 339–343
`get_cell` method, Game of Life, 352–353
`get_comp_move` function, 288, 290–291
`getcwd` function, `os` module, 210
`get_mat_str` function, Game of Life, 355–356
`get_move` function, Tic-Tac-Toe, 275–277
`get_next_prime` function, random-number generator, 372
`get_num` function, adding machine, 176–178
`get_point` function
3D Tic-Tac-Toe, 314
difference/sum of two points, 172–173
getter methods
defined, 376
multilevel temperature object, 379–381
using with setter methods, 378–379
writing for property, 376–377
`global` keyword, 190
Global variables
Beatles personality profile, 192–196
C++ syntax for, 191–192
decoding Roman numerals, 201–204
definition of, 400
`global` keyword, 190
local variables vs., 188–190
preventing local variable trap, 190–191
printing Life matrix for Game of Life, 355–356
reading text file with line numbers, 221–222
Roman numerals, 196–201
summary, 204–205
using global statements, 266–267
Glossary, 397–406
Google, as unimaginably large number, 11–14
Google-plex, 12, 14
Grounded division. *See* Integer division (//)

Group theory, eclipsing set theory, 244
GUI (graphical user interface) package. *See* Tkinter (GUI package)

H

Hash table, dictionaries, 232
h_dist (horizontal distance), Pythagorean distance formula, 48–49
Help
- begin using Python with IDLE, 6
- dictionary, 235
- from doc strings, 321–323
- online resources for, 8
- os module, 209
- sets, 243
- string methods, 148–149

Heuristics
- art of, 292–293
- computer player strategy, 285–287
- of get_comp_move function, 288, 290–291

History, of Python, 1–2
Horizontal distance (h_dist), Pythagorean distance formula, 48–49
Hyphens (-), joining group of characters, 143–144

I

i expression, list comprehension, 106–109, 116
IBM's Big Blue, super-chess playing program, 293
IDLE (interactive development environment)
- basic interactive mode vs., 4–6
- begin Python with, 6
- correcting mistakes from within, 6–7
- definition of, 400
- how this books works, 3
- printing values for variable names, 19

if/else operator, 274–278
if keyword
- computer program decisions, 53–55
- control structures, 56
- encoding strings, 161–162
- list comprehension, 114–115, 124
- simple program using, 59–60
- summary, 75

Immutable
- definition of, 400–401
- Python strings as, 139–141

Importing
- in Fraction class inheritance, 334
- functions from modules, 178–182
- in number-guessing game, 72
- os module, 209–210

inc_cells method, Game of Life, 352–353
Indentation
- computer program decisions, 54
- of doc strings, 323
- processing lists with for, 80–82
- in Python, 2, 33–34
- rules for functions, 31–32

Index numbers. *See* Zero-based indexing
Indexes
- definition of, 401
- enumerate function and, 101–103
- for lists, 89–93
- one-based, 294, 403
- out-of-range, 171
- and slicing, 85–88, 127–130
- for strings, 145
- zero-based, 406

Infinite integers
- computer hardware limits on, 12
- definition of, 401
- in Python, 11
- why C++ does not support, 11–12

Inheritance
- definition of, 331, 401
- Fraction class and, 333–336
- OOP and, 344
- overview of, 330–333

__init__ function
- 3D Tic-Tac-Toe, 313, 316
- database class design, 303–307

Fraction class and, 335–336
function typing/overloading, 324
Game of Life, 352–353
importance of, 301–302
as magic method, 402
Point3D class design, 310–312
PointN class and, 328–329
summary, 345
Initialization
creating N*M matrixes, 254–255
creating user-initialized matrix, 258–261
database class design, 303–307
elements in small matrixes, 249–250
__init__. *See* __init__ function
Point3D class design, 310–312
in operator
generator used with, 369
global variables, 194, 196
lists, 98
In/out parameters, lists, 170–171
I/O
opening text file, 211–212
quadratic equation with, 44–46
input function
getting numeric, 43–44
getting string input, 41–43
quadratic formula with I/O, 45–46
returns string, 131
Installation, Python, 4–6
Instance, definition of, 401
Instance variables
as default values, 339–343
definition of, 337, 401
in getter methods, 376
not necessarily inherited, 331
in setter methods, 377
int numeric type
convert strings to numeric, 131
getting numeric input, 43–44
in number-guessing game, 72
as object in Python, 295–296
summary, 51
Integer division (//)
automatically rounds down, 155
definition of, 401
extending Fraction class, 335
floating point and, 15
Integers
changing list elements, 89–93
definition of, 401
getting numeric input, 43–44
variable naming conventions, 23
intersection method, sets, 243
Irregular-shaped matrixes, length of row, 251–252
isalpha() method, 149–150, 166
is_even function, computer program decisions, 54
isinstance function, 323–324
islower() method, character case, 149–150, 166
is prime function, random-number generator, 372
istitle() method, character case, 149–150
isupper() method, character case, 149–150, 166
is_win function, 3D Tic-Tac-Toe, 314–315
Iterable
creating own, 368
definition of, 101, 401–402
exploiting power of generators, 369–370
reading text file as, 216–217, 225
understanding, 367–368
Iteration, in Python, 2

J

join method
Game of Life, 357
for one-character strings, 364
optimizing code, 97–98
returns one long string, 136, 141, 143–145
for series of strings, 154, 166

Justification
    left, for non-numeric values, 394
    modifying fields in format string method, 103–104
    right, for numeric values, 394
    rule for truncation and, 395

K

KeyError exception, dictionaries, 230, 231
Keys, dictionary
    accessing existing values, 230
    definition of, 402
    explaining dictionary magic, 232
    personal phone book, 233–235
    restrictions, 232
    searching, 231–232
Key-value pairs
    creating dictionary as group of, 228
    explaining dictionary magic, 232
    storing in file, 240
    syntax for adding/changing, 229
Keywords
    defined, 11
    definition of, 402
    how this book works, 3

L

Left justification, format strings, 103–104
`len` function
    converting dictionaries to lists, 236
    counting trailing zeros, 137
    decoding Roman numerals, 202–203
    for dictionaries/sets/lists, 247
    `range` combined with, 90–91
    returns length of collection, 100
    returns length of string, 131, 136
    returns number of files in current directory, 209
Letters, variable naming conventions, 19, 27
Life matrix
    design of program, 352
    Game of Life rules, 350
    generating neighbor counts, 350–351
    printing, 355–358
`life_mat`, Game of Life, 355–356, 358–360
Linear vs. binary searches, 74–75
Lines
    dealing with ends of, 7–8
    reading text file with numbers of, 220–223
    sort words on, 141–142
List assignment, 23, 27
list-by-prefix command, saving to file, 239
List comprehension
    with conditional, 114–115
    creating large matrixes, 254–255, 256
    definition of, 402
    in difference between squares, 109–112
    optimizing rotating matrix code, 267–268
    printing Life matrix for Game of Life, 356–357
    Pythagorean triples, 118–120
    Sieve of Eratosthenes 2, 115–118
    simple, 106–109
    summary, 123–124
    two-dimensional, 112–114
`listdir` function, `os` module, 210
Lists
    converting dictionaries to, 235–236
    converting numbers to Roman numerals, 199–201
    copying data to slices, 88–89
    definition of, 402
    generating with `split` method, 138–139
    indexing and slicing, 85–88
    as iterables in Python, 101
    `in` keyword and, 98
    matrixes as list of. *See* Matrixes (2D lists)
    modifying elements with `for`, 82–85
    naming conventions, 148
    overview of, 77
    passing and modifying, 170–171

processing with for, 80–82
ranges and, 89–93
sets vs., 242
sieve of Eratosthenes example, 93–97
sorting information, 77–80
summary, 99–100
Tic-Tac-Toe, count method, 279–285
tuples vs., 120
variable-length arguments, 326–327
variable naming conventions, 23

Literal quotations, creating, 126–127
Living cells, 348–350, 351
load_file function, 238–241
Local variables
C++ syntax, 191–192
definition of, 402
favoring, 187–188
global variables vs., 188–190
preventing trap of, 190–191
summary, 204–205

log n (0) notation, binary searches, 74–75
Look-ahead (brute force), heuristics, 293
Loops
breaking from, 70–71
changing list elements from inside, 89–93
creating with while, 61–63
defined, 60
definition of, 402
number-guessing game, 71–74
optimizing code, 65–67
printing Fibonacci numbers, 67–70
using factorials, 63–65

Lowercase letters
character code for, 160
converting case of, 150–151
testing character case, 149–150

M

Mac systems, opening IDLE in, 6
Magic method
definition of, 402
explaining dictionary, 232
garbage collection, 262
list comprehension, 107
revising blocks of code, 45
special meaning in Python, 402

main function
dice game (craps), 181
difference/sum of two points, 172–173
factorials, 64
reading text file with line numbers, 220–223
rotating matrix, 264–266
Tic-Tac-Toe 3D, 313–314
Tic-Tac-Toe with one-line if/else, 275–277

Maintenance Rule, Conway's Game of Life, 349
make_roman function, converting numbers, 197–201
mat variable
creates unusable lists of lists, 253
irregular-shaped matrixes, 251–252

Matrix class, 352–358
Matrix, definition of, 402
Matrixes (2D lists)
accessing elements, 250–251
creating multiplication table, 256–258
creating N*M matrixes, 254–255
creating user-initialized, 258–261
customized for Game of Life, 352–354
irregular-shaped, and length of row, 251–252
multiplication (*) and lists, 252–253
overview of, 249
Python problem with, 253
rotating, 261–268
simple, 249–250
summary, 268–269
Tic-Tac-Toe. *See* Tic-Tac-Toe game
why it is not easier to create large, 255–256

max function, lists, 98

Methods
 `__init__`. *See* `__init__` function
 all are inherited, 331
 class variables and, 337–338
 for customized `Matrix` class in Game of Life, 352–354
 defining other, 309–310
 definition of, 403
 getting from doc strings, 322–323
 getting help for string, 148–149
 properties driven by, 376
 reserved names for, 301–302
 strings as objects that support, 295
 write, 300–301
`min` function
 lists, 98
 `PointN` class, 328, 330
Modula-3 language, and Python, 1
Modular division
 definition of, 403
 extending `Fraction` class, 334–336
 in randomization, 374–375
Modules
 moving `Matrix` class to, 354–358
 syntax for importing, 178–182, 186
Modulus operator (%). *See* Remainder operator (%)
Monty Python's Flying Circus, Python named after, 2
Multidimensional lists, 254–255
Multiple arguments, passing, 36–40, 167–168
Multiple-assignments, as shortcut, 23–24
Multiplication assignment (`*=`) operator
 factorial code, 64–65
 factorial code optimization, 92
 not used in larger expressions, 26
Multiplication operator (`*`)
 affect on lists, 252–253
 generating characters repeatedly, 365
 printing Life matrix for Game of Life, 355–356
 in sieve of Eratosthenes, 93
 string/number conversions, 130–131
Multiplication table, matrix for, 256–258
Mutable
 copying data to slices, 88–89
 definition of, 78, 403

N

Named arguments, passing, 174–175, 185
Naming conventions
 case-sensitivity, 18–19
 lists, 148
 modules, 354
 for special methods, 300–301
 strings, 147–148
 variables, 19, 23, 27
`nc_mat` cells, Game of Life, 358–361
Negative indexes, 86
Negative numbers
 counting trailing zeros, 134
 indexing lists, 86
 indexing strings, 145
 strings, 129
Neighbor Count, Conway's Game of Life, 348–351, 358, 360–361
Nested functions, writing decorators, 382–385
Nested loops
 building two-dimensional arrays, 364
 list comprehension for, 113
 optimizing code for factorial program, 66
 print/reset all values of matrix, 257, 269
New File command, writing programs, 47
Newline character (`\n`)
 matrix for multiplication tables, 257–258
 `save_file` function, 240
 writing separate lines, 216, 224
`next` function, generators, 369–370
N*M matrixes, 254–255

None value
  of empty lists, 278
  functions, 177–178
  personal phone book application, 235
  reading text file with line numbers, 221
  searching for dictionary keys, 231
`not in` operator
  generators, 369
  lists, 98
`not` operator, Boolean, 55
Number-guessing game, 71–75
Numbers
  adding machine, 176–178
  assigning to variables, 17–22
  built-in support for, 2
  changing list elements from inside loop, 89–93
  as common data in lists, 78
  converting ASCII to characters from, 160–161
  converting into Roman numerals, 196–201
  converting strings to, 130–134
  converting to ASCII code, 159–160
  difference between square, 109–112
  floating-point. *See* Floating-point numbers
  getting numeric input, 43–44
  indexing and slicing lists, 85–88
  personal phone book, 232–235
  printing Fibonacci, 67–70
  Python and, 9–14
  shortcuts for, 23–26
  sieve of Eratosthenes generating prime, 93–97
  summary, 26–27
  writing factorials, 63–67

O

O player. *See* Tic-Tac-Toe game
Object orientation
  advantages of, 343–344
  definition of, 403
  overview of, 295
  polymorphic methods in Python and, 301
  write methods at class level, 300–301
Object-oriented programming (OOP), 343–344
Objects
  attaching data to, 298–300
  definition of, 295–296, 403
  as instances of classes, 296–301
  using classes to create, 297–298
One-based indexing, 294, 403
`one_char_string` argument, `ord` function, 159
One-line version of if/else, Tic-Tac-Toe, 274–278
One-to-many relationship, classes, 296–301
Online resources, help, 8
OOP (object-oriented programming), 343–344
Op System (`os`) module, 208–211, 224
`open` method, text files, 211–212, 224
Open parenthesis, ends of lines, 7
Operations, on sets, 242–244
Operators
  conditional and Boolean, 55
  exponent (**), 10
  precedence, 11
  precedence table for Python, 391–392
  remainder (%), 12–13
  test-for-equality (==), 10–11
Optimized approach, in this book, 4
`or` operator, Boolean, 55
`ord` function, converts one-length string to ASCII, 159, 166
Order
  meaningful in lists, 78–80
  no significance in dictionaries, 228
  no significance in sets, 117–118, 242–243
`os` (Op System) module, 208–211, 224

Output string, formatted, 46–47
Overloading, function typing and, 323–324

P

Packages, Python, 2
Padding characters, 395
Parentheses ()
  completing test for, 154–158
  enclosing arguments in functions, 31
  executing functions, 31
  famous, 158
  testing for, 151–154
  using classes to create objects, 297–298
Parenthesis, open, 7
Pascal, Blaise, 183
`pass` keyword, as placeholder, 297
Path name, reading/writing text, 210
Patterns, Game of Life, 358
PEP-8 standard, 32–34
phi number, as golden ratio, 39, 40, 44–46
Phone book, personal
  converting dictionaries to lists, 235–236
  exercises, 241
  loading and saving to file, 238–241
  personal phone book, 232–235
  reading items by prefix, 236–238
`phone_dict`. *See* Phone book, personal
Playing for position, heuristics, 293
`play_the_game` function, dice (craps), 179–182
`Point3D` class, 310–312
`PointN` class, 327–331
Polygon "automated" class, 340–343
Polymorphism, 301, 403–404
Positional arguments, advanced functions, 174
Precedence
  Boolean vs. comparison operators, 55
  Python operator, 11, 13
  table for Python operator, 391–392
Precision, limited in floating-point numbers, 16
Prefix, reading phone book items by, 236–238
Prime numbers
  creating random-number generator, 372
  defined, 93
  in randomization, 375
  sieve of Eratosthenes generating, 93–97, 115–118
Print field, formatting rules, 394
`print` function
  adding call to, 35–36
  adding `end` keyword on, 62
  arguments, 175
  computer program decisions for, 53
  Fibonacci numbers, 69–70
  formatted output strings, 46–47
  full program for Game of Life, 358–360
  indexing and slicing strings, 127–130
  Life matrix for Game of Life, 355–356
  making into generator, 368
  multiplication table matrix, 256–258
  optimizing code, 96–97
  performance time and number of calls to, 258, 269
  reading text file, 216
  summary, 51
  syntax, 38–40
  using quotation marks, 126–127
`print_mat` function, 265–266, 275–278
`print_nums` function, decorators, 387–389
Printouts, string method, 148–149
Privacy
  data access in C++ vs. Python, 304
  getter methods, 377
  setter methods, 378
Probability calculation, casino odds, 183–184
Program, in *Westworld*, 56–58
Programs, your first
  adding print message, 35–36
  distance formula in scripts, 47–48
  formatted output string, 46–50

- getting numeric input, 43
- getting string input, 41–43
- indentation, 33–34
- quadratic equation as a function, 38–40
- quadratic formula with I/O, 44–46
- summary, 50–51
- syntax summaries, 36–37
- temperatures rising, 29–33
- writing Python functions, 29

`promote` method, 309
Prompt, writing text file with, 214–216
Properties
- definition of, 404
- exercises, 381
- getter methods, 376–377
- multilevel temperature object, 379–381
- overview of, 375–376
- putting getters and setters together, 378–379
- setter methods, 377–378
- summary, 390

Prototyping language, Python as, 3
Pseudocode
- changing list elements from inside loop, 92
- computer play, 290, 292
- `decode_roman` function, 202–203
- dice game (craps), 180–181
- of programs or functions, 40
- in this book, 3
- user-initialized matrix, 259–260
- writing loops, 62–65
- writing Sieve of Eratosthenes, 95, 116
- writing text file with prompt, 215

Pseudorandom numbers, 372–373, 374
Punctuation, in this book, 3
Pythagorean distance formula, 48–50
Pythagorean theorem, 120–122
Pythagorean triples, 118–120
Python, introduction
- begin with IDLE, 6
- brief history of, 1–2
- correct mistakes from within IDLE, 6–7
- dealing with ends of lines, 7–8
- how it is different, 2–3
- how this book works, 3–4
- installing, 4–6
- online sources of help, 8
- overview of, 1

Python lists. *See* Lists
Python Package Index, 2
`Python txt` subdirectory, 209

Q

`quad` function, 38–40
Quadratic equation
- assigning numbers to variables, 19–21
- as function, 38–40
- with I/O, 44–46

Quotation marks
- adding call to `print` function, 35–36
- delineating text string with matching, 125–127, 144–145
- getting from doc strings, 322–323
- typing literal backslash (\\) in strings, 8

R

`randint` function, dice game (craps), 181
`random` module, 370–373
Randomness, how to introduce, 373–375
Random-number generator, customizing, 370–373
`range` function
- combining with `for loop`, 399
- combining with `len` function, 90–91
- counting trailing zeros, 132–133
- difference between squares, 110
- generating list of integers from 0 to N-1, 89–90
- iterating through lists without, 101–102
- revised sieve of Eratosthenes, 244
- sieve of Eratosthenes example, 96
- simple list comprehension, 106–109

Rapid application development language, Python, 3
raw_input function, Python 2.0, 41–43
Reading
 items by prefix, 236–238
 text files, 216–217
 text files vs. binary, 207–208
 text files with line numbers, 220–223, 225
readlines method
 load_file function, 240
 reading text files, 217
 reading text files with line numbers, 222
References, variables in Python as, 21–22
Remainder division operator (%). *See* Remainder operator (%)
Remainder operator (%)
 computer program decisions, 54
 creating random-number generator, 373
 defined, 12–13
 floating-point numbers and, 15
 full program for Game of Life, 361
remove method, set operations, 243, 247
__repr__ method, 310
return statement
 creating own generator, 368
 returning multiple values to functions, 169–170, 176–177, 185
 rules for functions, 31
Revised sieve of Eratosthenes, 401
Right justification, format string method, 103–104
Right triangles, Pythagorean triples, 118–120
Robots, in *Westworld*, 56–58
r mode, opening file for reading, 224
roll function, dice game (craps), 181, 182
Roman numerals
 decoding, 201–204
 global variables, 196–199
 marking highly important figures in, 201
 superior Hindu-Arabic system vs., 200
rom_list, Roman numerals, 199–203
rotate_mat function, rotating matrix, 264–268
Rotating matrix
 example, 264–268
 and garbage collection, 263–264
 overview of, 261–263
Rows. *See* Matrixes (2D lists)
Rule(s)
 assigning numbers to variables, 17–20
 computer player heuristics, 285–287
 Conway's Game of Life, 348–350
 functions, 31
 global variables, 191
 important formatting, 393–395
 local vs. global variables, 189–190
 Python syntax, 3–4
 variables, 27
Run Module (F5), Python programs, 48

S

Save File button, installing Python, 4
Save (or Save As) command, Python programs, 48
save_file function, 238–241
Scripts
 distance formula, 47–48
 string input, 41–43
seek function, file pointer, 217, 225
self argument
 __init__ method, 301–302
 defined, 318
 defining other methods, 309–310
 design database class, 306–307, 309–310
 design Point3D class, 310–311, 312
 in Python, 302–303
sep'' (separator) argument, 47, 51
sep (separator) argument
 named arguments, 174
 print function, 47, 51
Separator characters, 143–144

Set braces notation, mathematics, 241
`set_cells` method, Game of Life, 352–353
Set comprehension, 244–245, 404
`set` keyword, sieve of Eratosthenes, 244
Sets
  definition of, 404
  importance of, 244
  mathematical theory, 244
  operations on, 242–244
  overview of, 241–242
  sieve of Eratosthenes, 117–118, 244–246
  summary, 246–247
setter methods
  defined, 376
  multilevel temperature object example, 379–381
  using with getter methods, 378–379
  writing for property, 377–378
Set-theory (Venn) diagram, 243
Shortcuts, Python, 23–27
Side space, computer player heuristics, 287
Sieve of Eratosthenes
  list functions and `in` keyword, 97–98
  overview of, 93–97
  revised, 244–246
  using list comprehension to write, 115–118
  who Eratosthenes was, 98–99
Simulation, Conway's Game of Life, 360–361
Single-character ops
  accessing individual characters, 148
  art of cryptography, 164
  completing palindrome test, 154–158
  converting ASCII to character, 160–163
  converting case of letters, 150–151
  converting to ASCII code, 159–160
  decoding strings, 164–166
  encoding strings, 161–164
  famous palindromes, 158
  help with string methods, 148–149
  overview of, 147
  Python string/list naming conventions, 147–148
  summary, 166
  testing for palindromes, 151–154
  testing uppercase vs. lowercase, 149–150
Slicing
  changing any/all values in lists, 171
  definition of, 404
  producing parts of strings, 128–130, 145
  strings, 166
  testing for palindromes, 157–158
  user-initialized matrix, 260
Slicing lists
  copying data to slices, 88–89
  creating separate lists, 79
  and indexing, 85–87
  indexing vs., 88
Slider pattern, Conway's Game of Life, 358
`sort` method
  sieve of Eratosthenes, 117–118
  sorting lists, 83–85, 98
  sorting words on line, 141–142
Spaces
  in default `print` function, 35–36
  formatted output strings, 46–47
  Python indentation and, 33–34
  replacing tab with multiple blank, 34
  `strip` method used on, 136
Special method names
  `__init__`. *See* `__init__` function
  `__repr__`, 310
  `__str__`, 309
  defined, 301–302
  `Point3D` class, 310–311
`Sphere` subclass, Polygon "automated" class, 341–342
`split` method
  difference/sum of two points, 172–173
  returns list of smaller strings, 136, 138–139, 145
  sorting words on line, 141–142
  Tic-Tac-Toe, 275, 277

Square root
    assigning numbers to variables, 19–21
    Pythagorean distance formula, 48–50
Square subclass, Polygon "automated" class, 341–342
Squares, difference between, 109–112, 124
start argument
    changing list elements from inside loop, 90
    counting trailing zeros, 132–134
    slicing strings, 145
startswith method, read phone book by prefix, 237–238
Starvation Rule, Conway's Game of Life, 349
step argument
    indexing and slicing lists, 87–88
    slicing strings, 145, 157, 166
StopIteration exception, generators, 369
Strings
    alphabetical order for lists of, 79
    building with concatenation (+), 139–141
    characters as one-length, 148
    as common data in lists, 78
    creating with quote marks, 125–127
    definition of, 405
    encrypting/decoding. *See* Cryptography
    formatted output, 46–47
    getting help from doc, 321–323
    getting help with methods for, 148–149
    indexing and slicing, 127–130
    input for, 41–43
    as iterables in Python, 101
    join method, 143–144
    naming conventions, 147–148
    as objects in Python, 295–296
    as objects that support methods, 295
    overview of, 125
    Python characters vs., 135
    single-character. *See* Single-character ops
    sorting words on line, 141–142
    splitting, 138–139
    string/number conversions, 130–134
    stripping, 135–138
    summary, 144–146
strip method
    counting trailing zeros, 131–132
    difference and sum of two points, 173
    producing stripped strings, 135–136, 145
    of string object, 295
__str__ method
    converting object to string format, 309
    extending Fraction class, 334–336
    writing PointN class, 328–329
str type, 131, 324
Subclasses
    advantage of, 332
    inheriting methods from superclasses, 333
    in Polygon "automated" class, 342
    summary, 345
    syntax, 331
__sub__ method
    3D Tic-Tac-Toe, 313, 316
    Point3D class, 310–311
Super-chess playing programs, 293
Superclasses
    definition of, 405
    subclasses inherit methods of their, 333
Symbol, definition of, 405
Syntax errors
    functions, 31
    indentation, 33–34
System time
    creating random-number generator, 370–372
    randomization via, 374

T

Tab characters, indentation and, 33–34
Temperature class, multilevel temperature object, 379–381
temp_list, sorting information, 77–80

Test-for-equality (==) operator
  in case-insensitive comparisons, 150–151
  as comparison operator, 55
  dice game (craps), 182
  `if` statement, 54
  sorting information with lists, 79
  vs. assignment (=), 19
  working with, 10–11
Testing
  for character case, 149–150
  complete palindrome, 154–158
  palindrome, 151–154
`test_way` function, Tic-Tac-Toe, 282, 284
`test_win` function, Tic-Tac-Toe, 280–283
Text-editor, writing/executing long programs, 48
Text files
  vs. binary files, 207–208, 224
  closing, 212
  opening, 211–212
  reading, 216–217
  treating as iterables, 216
  writing, 213–216
Text, rules for formatting ordinary, 393
Text strings
  getting input, 41–43
  variable naming conventions, 23
This book
  how it works, 3–4
  Python 3.0 used in, 2
Three-dimensional (3D) Tic-Tac-Toe. *See* 3D Tic-Tac-Toe game
Tic-Tac-Toe game
  3D. *See* 3D Tic-Tac-Toe game
  art of heuristics, 292–293
  computer play, computer plays first, 287–291
  computer play, computer plays second, 291–292
  computer player, 285–287
  `count` method for lists, 279–285
  create variations on, 279
  designing board, 271–273
  implementing strategy, 271
  phases of play, 273
  Python one-line if/else, 274–278
  summary, 294
`time` module
  creating random-number generator, 371–372
  decorators as debugging tool, 387–389
  randomization via system time, 374
Tkinter (GUI package)
  definition of, 405
  Python installed with, 4
  shifting focus to window generated by, 48
Tracking employees, 305–308
Transformations, generators and, 374–375
Triple quotation marks, 126–127
`True` value
  adding machine application, 176–178
  completing palindrome test, 154–158
  decisions inside computer program, 54–55
  dice game (craps), 181–182
  `get_move` function, 277
  `in` keyword producing, 196
  reading text file with line numbers, 222
  searching for dictionary keys, 231
  sieve of Eratosthenes, 93–97
  sorting lists, 85
  test-for-equality (==) operator, 11
  testing character case, 149–150
Truncation (limit size of print field) rules, 394, 395
`Try/except` block
  advantages of, 219–220
  duck typing, 325
  `FileNotFoundError`, 218–219, 225
  searching for dictionary keys, 231
  `StopIteration` exception, 370
`ttt_list`, two-player Tic-Tac-Toe, 280–283

Tuples
converting numbers to Roman numerals, 199–201
customizing `Matrix` class for Game of Life, 354
definition of, 405
`enumerate` function generating, 103
lists vs., 120
loading and saving to file, 239
printing table, 106
Pythagorean triples and, 119
as Python shortcut, 23–24
Turing, Alan, 58, 164
Two-dimensional (2D) list comprehension, 112–114
Two-dimensional (2D) lists. *See* Matrixes (2D lists)
Types, definition of, 405

U

Underscore (_), variable names, 19, 27
Unicode, numeric display on computer, 159
`union` method, sets, 243
`upper()` method, converts string to uppercase, 151, 166
Uppercase letters
character code for, 160
converting case of, 150–151
converting strings to, 151–153
testing characters for case, 149–150
User-initialized matrix
creating, 258–261
rotating. *See* Rotating matrix

V

Values
accessing existing dictionary, 230
creating default argument, 174–175, 185
defining default instance variables, 339–343
difference and sum of two points, 173
duplicating/ordering in Python lists, 78
explaining dictionary magic, 232
passed to function arguments based on position, 185
personal phone book, 233–235
properties can encapsulate virtual, 379
Python elements have no associated, 241
returning multiple function, 168–173
sets maintaining unique, 242
Van Rossum, Guido, 1–2
Variable-length argument lists, 326–327
Variables
as aliases, 79
assigning numbers to, 17–22
definition of, 405
general rule of Python, 17, 27
getting numeric input, 41–43
getting string input, 41–43
global. *See* Global variables
how Python is different, 2
local. *See* Local variables
naming conventions, 19, 27
as references in Python, 21–22
`v_dist` (vertical distance), Pythagorean distance formula, 48–49
Venn (set-theory) diagram, 243
Versions, integer division differences, 15
Vertical distance (`v_dist`), Pythagorean distance formula, 48–49
Virtual sequences, generators, 368
Virtual values, properties, 379–380
Von Newman, John, 347

W

Westworld, programs and robots in, 56–58
`while` loop
breaking from, 70–71
counting trailing zeros, 133
creating, 61–63

files and exception handling, 218–219
optimizing code, 65–67
printing Fibonacci numbers, 67–70
summary, 74–75
using factorials, 63–65
`while True`, infinite loops, 72, 75
Widget, definition of, 405
Win detection, two-player Tic-Tac-Toe, 279–285
Words, sorting on line, 141–142
Wrapper
decoration in Python, 385–387
definition of, 406
implementing in Game of Life, 351
writing decorators, 383–384
`write` method
`save_file` function, 240
text file, 213–216

X

X player. *See* Tic-Tac-Toe game

Y

`yield` statement, generators, 368

Z

Zero-based indexing
accessing matrix elements, 250, 269
definition of, 406
vs. one-based, 294
Zeros, count trailing, 131–134, 136–137
`zip` function, 330