O'REILLY®

2nd Edition
Updated for Python 3

Think Python

HOW TO THINK LIKE A COMPUTER SCIENTIST

Allen B. Downey

O'REILLY®

Think Python

If you want to learn how to program, working with Python is an excellent way to start. This hands-on guide takes you through the language a step at a time, beginning with basic programming concepts before moving on to functions, recursion, data structures, and object-oriented design. This second edition and its supporting code have been updated for Python 3.

Through exercises in each chapter, you'll try out programming concepts as you learn them. *Think Python* is ideal for students at the high school or college level, as well as self-learners, home-schooled students, and professionals who need to learn programming basics. Beginners just getting their feet wet will learn how to start with Python in a browser.

- Start with the basics, including language syntax and semantics
- Get a clear definition of each programming concept
- Learn about values, variables, statements, functions, and data structures in a logical progression
- Discover how to work with files and databases
- Understand objects, methods, and object-oriented programming
- Use debugging techniques to fix syntax, runtime, and semantic errors
- Explore functions, data structures and algorithms through a series of case studies

The example code for this book is maintained in a public GitHub repository that users can easily download and modify.

Allen Downey is a Professor of Computer Science at Olin College of Engineering. He has taught at Wellesley College, Colby College, and U.C. Berkeley. He has a PhD in Computer Science from U.C. Berkeley and Master's and Bachelor's degrees from MIT.

"*Think Python* made me smile again and again. Allen Downey's explanations are crystal clear and his inspired exercises will engage learners of many backgrounds."

—**Luciano Ramalho**
Technical Principal at ThoughtWorks
and author of *Fluent Python*

PROGRAMMING/PYTHON

US $44.99 CAN $51.99
ISBN: 978-1-491-93936-9

Twitter: @oreillymedia
facebook.com/oreilly

SECOND EDITION

Think Python

Allen B. Downey

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Think Python
by Allen B. Downey

Printed in the United States of America.

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

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

Editor: Meghan Blanchette
Production Editor: Kristen Brown
Copyeditor: Nan Reinhardt
Proofreader: Amanda Kersey
Indexer: Allen Downey
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

August 2012: First Edition
December 2015: Second Edition

Revision History for the Second Edition
2015-11-20: First Release
2023-11-10: Second Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491939369* for release details.

978-1-491-93936-9

[LSI]

Table of Contents

Preface ..... **xi**

1. The Way of the Program **1**
What Is a Program? 1
Running Python 2
The First Program 3
Arithmetic Operators 3
Values and Types 4
Formal and Natural Languages 5
Debugging 7
Glossary 8
Exercises 9

2. Variables, Expressions and Statements **11**
Assignment Statements 11
Variable Names 12
Expressions and Statements 12
Script Mode 13
Order of Operations 14
String Operations 15
Comments 15
Debugging 16
Glossary 17
Exercises 18

3. Functions **21**
Function Calls 21
Math Functions 22

Composition 23
Adding New Functions 23
Definitions and Uses 25
Flow of Execution 25
Parameters and Arguments 26
Variables and Parameters Are Local 27
Stack Diagrams 28
Fruitful Functions and Void Functions 29
Why Functions? 30
Debugging 30
Glossary 31
Exercises 32

4. Case Study: Interface Design. 35
The turtle Module 35
Simple Repetition 37
Exercises 38
Encapsulation 38
Generalization 39
Interface Design 40
Refactoring 41
A Development Plan 42
docstring 43
Debugging 43
Glossary 44
Exercises 44

5. Conditionals and Recursion. 47
Floor Division and Modulus 47
Boolean Expressions 48
Logical Operators 49
Conditional Execution 49
Alternative Execution 49
Chained Conditionals 50
Nested Conditionals 50
Recursion 51
Stack Diagrams for Recursive Functions 53
Infinite Recursion 53
Keyboard Input 54
Debugging 55
Glossary 56
Exercises 57

6. Fruitful Functions. 61
Return Values 61
Incremental Development 62
Composition 64
Boolean Functions 65
More Recursion 66
Leap of Faith 68
One More Example 68
Checking Types 69
Debugging 70
Glossary 71
Exercises 72

7. Iteration. 75
Reassignment 75
Updating Variables 76
The while Statement 77
break 78
Square Roots 79
Algorithms 81
Debugging 81
Glossary 82
Exercises 82

8. Strings. 85
A String Is a Sequence 85
len 86
Traversal with a for Loop 86
String Slices 87
Strings Are Immutable 88
Searching 89
Looping and Counting 89
String Methods 90
The in Operator 91
String Comparison 92
Debugging 92
Glossary 94
Exercises 95

9. Case Study: Word Play. 99
Reading Word Lists 99
Exercises 100

Search 101
Looping with Indices 103
Debugging 104
Glossary 105
Exercises 105

10. Lists. 107
A List Is a Sequence 107
Lists Are Mutable 108
Traversing a List 109
List Operations 110
List Slices 110
List Methods 111
Map, Filter and Reduce 111
Deleting Elements 113
Lists and Strings 113
Objects and Values 114
Aliasing 115
List Arguments 116
Debugging 118
Glossary 119
Exercises 120

11. Dictionaries. 125
A Dictionary Is a Mapping 125
Dictionary as a Collection of Counters 127
Looping and Dictionaries 128
Reverse Lookup 129
Dictionaries and Lists 130
Memos 131
Global Variables 133
Debugging 134
Glossary 135
Exercises 137

12. Tuples. 139
Tuples Are Immutable 139
Tuple Assignment 141
Tuples as Return Values 141
Variable-Length Argument Tuples 142
Lists and Tuples 143
Dictionaries and Tuples 144

Sequences of Sequences 146
Debugging 147
Glossary 148
Exercises 148

13. Case Study: Data Structure Selection. 151
Word Frequency Analysis 151
Random Numbers 152
Word Histogram 153
Most Common Words 155
Optional Parameters 155
Dictionary Subtraction 156
Random Words 157
Markov Analysis 158
Data Structures 159
Debugging 161
Glossary 162
Exercises 163

14. Files. 165
Persistence 165
Reading and Writing 166
Format Operator 166
Filenames and Paths 167
Catching Exceptions 169
Databases 169
Pickling 170
Pipes 171
Writing Modules 172
Debugging 173
Glossary 174
Exercises 175

15. Classes and Objects. 177
Programmer-Defined Types 177
Attributes 178
Rectangles 179
Instances as Return Values 181
Objects Are Mutable 181
Copying 182
Debugging 183
Glossary 184

Exercises 185

16. Classes and Functions. 187
Time 187
Pure Functions 188
Modifiers 189
Prototyping versus Planning 190
Debugging 192
Glossary 192
Exercises 193

17. Classes and Methods. 195
Object-Oriented Features 195
Printing Objects 196
Another Example 198
A More Complicated Example 198
The init Method 199
The __str__ Method 200
Operator Overloading 200
Type-Based Dispatch 201
Polymorphism 202
Interface and Implementation 203
Debugging 204
Glossary 204
Exercises 205

18. Inheritance. 207
Card Objects 207
Class Attributes 208
Comparing Cards 210
Decks 211
Printing the Deck 211
Add, Remove, Shuffle and Sort 212
Inheritance 213
Class Diagrams 214
Data Encapsulation 215
Debugging 217
Glossary 218
Exercises 219

19. The Goodies. 223
Conditional Expressions 223

List Comprehensions 224
Generator Expressions 225
any and all 226
Sets 226
Counters 228
defaultdict 229
Named Tuples 230
Gathering Keyword Args 232
Glossary 233
Exercises 233

20. Debugging. 235
Syntax Errors 235
I keep making changes and it makes no difference. 237
Runtime Errors 237
My program does absolutely nothing. 237
My program hangs. 238
When I run the program I get an exception. 239
I added so many print statements I get inundated with output. 240
Semantic Errors 241
My program doesn't work. 241
I've got a big hairy expression and it doesn't do what I expect. 242
I've got a function that doesn't return what I expect. 243
I'm really, really stuck and I need help. 243
No, I really need help. 243

21. Analysis of Algorithms. 245
Order of Growth 246
Analysis of Basic Python Operations 248
Analysis of Search Algorithms 250
Hashtables 251
Glossary 255

Index. 257

Preface

The Strange History of This Book

In January 1999 I was preparing to teach an introductory programming class in Java. I had taught it three times and I was getting frustrated. The failure rate in the class was too high and, even for students who succeeded, the overall level of achievement was too low.

One of the problems I saw was the books. They were too big, with too much unnecessary detail about Java, and not enough high-level guidance about how to program. And they all suffered from the trapdoor effect: they would start out easy, proceed gradually, and then somewhere around Chapter 5 the bottom would fall out. The students would get too much new material, too fast, and I would spend the rest of the semester picking up the pieces.

Two weeks before the first day of classes, I decided to write my own book. My goals were:

- Keep it short. It is better for students to read 10 pages than not read 50 pages.
- Be careful with vocabulary. I tried to minimize jargon and define each term at first use.
- Build gradually. To avoid trapdoors, I took the most difficult topics and split them into a series of small steps.
- Focus on programming, not the programming language. I included the minimum useful subset of Java and left out the rest.

I needed a title, so on a whim I chose *How to Think Like a Computer Scientist*.

My first version was rough, but it worked. Students did the reading, and they understood enough that I could spend class time on the hard topics, the interesting topics and (most important) letting the students practice.

I released the book under the GNU Free Documentation License, which allows users to copy, modify, and distribute the book.

What happened next is the cool part. Jeff Elkner, a high school teacher in Virginia, adopted my book and translated it into Python. He sent me a copy of his translation, and I had the unusual experience of learning Python by reading my own book. As Green Tea Press, I published the first Python version in 2001.

In 2003 I started teaching at Olin College and I got to teach Python for the first time. The contrast with Java was striking. Students struggled less, learned more, worked on more interesting projects, and generally had a lot more fun.

Since then I've continued to develop the book, correcting errors, improving some of the examples and adding material, especially exercises.

The result is this book, now with the less grandiose title *Think Python*. Some of the changes are:

- I added a section about debugging at the end of each chapter. These sections present general techniques for finding and avoiding bugs, and warnings about Python pitfalls.
- I added more exercises, ranging from short tests of understanding to a few substantial projects. Most exercises include a link to my solution.
- I added a series of case studies—longer examples with exercises, solutions, and discussion.
- I expanded the discussion of program development plans and basic design patterns.
- I added appendices about debugging and analysis of algorithms.

The second edition of *Think Python* has these new features:

- The book and all supporting code have been updated to Python 3.
- I added a few sections, and more details on the Web, to help beginners get started running Python in a browser, so you don't have to deal with installing Python until you want to.
- For "The turtle Module" on page 35 I switched from my own turtle graphics package, called Swampy, to a more standard Python module, `turtle`, which is easier to install and more powerful.
- I added a new chapter called "The Goodies", which introduces some additional Python features that are not strictly necessary, but sometimes handy.

I hope you enjoy working with this book, and that it helps you learn to program and think like a computer scientist, at least a little bit.

—Allen B. Downey
Olin College

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

Bold
: Indicates terms defined in the Glossary.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *http://www.greenteapress.com/thinkpython2/code*.

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

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Think Python*, 2nd Edition, by Allen B. Downey (O'Reilly). Copyright 2016 Allen Downey, 978-1-4919-3936-9."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Online Learning

For more than 40 years, O'Reilly Media has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *http://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-889-8969 (in the United States or Canada)
707-829-7019 (international or local)
707-829-0104 (fax)
support@oreilly.com
https://www.oreilly.com/about/contact.html

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://bit.ly/think-python_2E*.

For news and information about our books and courses, visit *http://oreilly.com*.

Find us on LinkedIn: *https://linkedin.com/company/oreilly-media*

Follow us on Twitter: *https://twitter.com/oreillymedia*

Watch us on YouTube: *https://youtube.com/oreillymedia*

Acknowledgments

Many thanks to Jeff Elkner, who translated my Java book into Python, which got this project started and introduced me to what has turned out to be my favorite language.

Thanks also to Chris Meyers, who contributed several sections to *How to Think Like a Computer Scientist*.

Thanks to the Free Software Foundation for developing the GNU Free Documentation License, which helped make my collaboration with Jeff and Chris possible, and Creative Commons for the license I am using now.

Thanks to the editors at Lulu who worked on *How to Think Like a Computer Scientist*.

Thanks to the editors at O'Reilly Media who worked on *Think Python*.

Thanks to all the students who worked with earlier versions of this book and all the contributors (listed below) who sent in corrections and suggestions.

Contributor List

More than 100 sharp-eyed and thoughtful readers have sent in suggestions and corrections over the past few years. Their contributions, and enthusiasm for this project, have been a huge help.

If you have a suggestion or correction, please send email to *feedback@thinkpython.com*. If I make a change based on your feedback, I will add you to the contributor list (unless you ask to be omitted).

If you include at least part of the sentence the error appears in, that makes it easy for me to search. Page and section numbers are fine, too, but not quite as easy to work with. Thanks!

- Lloyd Hugh Allen sent in a correction to Section 8.4.
- Yvon Boulianne sent in a correction of a semantic error in Chapter 5.
- Fred Bremmer submitted a correction in Section 2.1.
- Jonah Cohen wrote the Perl scripts to convert the LaTeX source for this book into beautiful HTML.
- Michael Conlon sent in a grammar correction in Chapter 2 and an improvement in style in Chapter 1, and he initiated discussion on the technical aspects of interpreters.
- Benoit Girard sent in a correction to a humorous mistake in Section 5.6.
- Courtney Gleason and Katherine Smith wrote `horsebet.py`, which was used as a case study in an earlier version of the book. Their program can now be found on the website.
- Lee Harr submitted more corrections than we have room to list here, and indeed he should be listed as one of the principal editors of the text.
- James Kaylin is a student using the text. He has submitted numerous corrections.
- David Kershaw fixed the broken `catTwice` function in Section 3.10.
- Eddie Lam has sent in numerous corrections to Chapters 1, 2, and 3. He also fixed the Makefile so that it creates an index the first time it is run and helped us set up a versioning scheme.

- Man-Yong Lee sent in a correction to the example code in Section 2.4.
- David Mayo pointed out that the word "unconsciously" in Chapter 1 needed to be changed to "subconsciously".
- Chris McAloon sent in several corrections to Sections 3.9 and 3.10.
- Matthew J. Moelter has been a long-time contributor who sent in numerous corrections and suggestions to the book.
- Simon Dicon Montford reported a missing function definition and several typos in Chapter 3. He also found errors in the `increment` function in Chapter 13.
- John Ouzts corrected the definition of "return value" in Chapter 3.
- Kevin Parks sent in valuable comments and suggestions as to how to improve the distribution of the book.
- David Pool sent in a typo in the glossary of Chapter 1, as well as kind words of encouragement.
- Michael Schmitt sent in a correction to the chapter on files and exceptions.
- Robin Shaw pointed out an error in Section 13.1, where the `printTime` function was used in an example without being defined.
- Paul Sleigh found an error in Chapter 7 and a bug in Jonah Cohen's Perl script that generates HTML from LaTeX.
- Craig T. Snydal is testing the text in a course at Drew University. He has contributed several valuable suggestions and corrections.
- Ian Thomas and his students are using the text in a programming course. They are the first ones to test the chapters in the latter half of the book, and they have made numerous corrections and suggestions.
- Keith Verheyden sent in a correction in Chapter 3.
- Peter Winstanley let us know about a longstanding error in our Latin in Chapter 3.
- Chris Wrobel made corrections to the code in the chapter on file I/O and exceptions.
- Moshe Zadka has made invaluable contributions to this project. In addition to writing the first draft of the chapter on Dictionaries, he provided continual guidance in the early stages of the book.
- Christoph Zwerschke sent several corrections and pedagogic suggestions, and explained the difference between *gleich* and *selbe*.
- James Mayer sent us a whole slew of spelling and typographical errors, including two in the contributor list.
- Hayden McAfee caught a potentially confusing inconsistency between two examples.
- Angel Arnal is part of an international team of translators working on the Spanish version of the text. He has also found several errors in the English version.
- Tauhidul Hoque and Lex Berezhny created the illustrations in Chapter 1 and improved many of the other illustrations.
- Dr. Michele Alzetta caught an error in Chapter 8 and sent some interesting pedagogic comments and suggestions about Fibonacci and Old Maid.
- Andy Mitchell caught a typo in Chapter 1 and a broken example in Chapter 2.

- Kalin Harvey suggested a clarification in Chapter 7 and caught some typos.
- Christopher P. Smith caught several typos and helped us update the book for Python 2.2.
- David Hutchins caught a typo in the Foreword.
- Gregor Lingl is teaching Python at a high school in Vienna, Austria. He is working on a German translation of the book, and he caught a couple of bad errors in Chapter 5.
- Julie Peters caught a typo in the Preface.
- Florin Oprina sent in an improvement in `makeTime`, a correction in `printTime`, and a nice typo.
- D. J. Webre suggested a clarification in Chapter 3.
- Ken found a fistful of errors in Chapters 8, 9 and 11.
- Ivo Wever caught a typo in Chapter 5 and suggested a clarification in Chapter 3.
- Curtis Yanko suggested a clarification in Chapter 2.
- Ben Logan sent in a number of typos and problems with translating the book into HTML.
- Jason Armstrong saw the missing word in Chapter 2.
- Louis Cordier noticed a spot in Chapter 16 where the code didn't match the text.
- Brian Cain suggested several clarifications in Chapters 2 and 3.
- Rob Black sent in a passel of corrections, including some changes for Python 2.2.
- Jean-Philippe Rey at Ecole Centrale Paris sent a number of patches, including some updates for Python 2.2 and other thoughtful improvements.
- Jason Mader at George Washington University made a number of useful suggestions and corrections.
- Jan Gundtofte-Bruun reminded us that "a error" is an error.
- Abel David and Alexis Dinno reminded us that the plural of "matrix" is "matrices", not "matrixes". This error was in the book for years, but two readers with the same initials reported it on the same day. Weird.
- Charles Thayer encouraged us to get rid of the semicolons we had put at the ends of some statements and to clean up our use of "argument" and "parameter".
- Roger Sperberg pointed out a twisted piece of logic in Chapter 3.
- Sam Bull pointed out a confusing paragraph in Chapter 2.
- Andrew Cheung pointed out two instances of "use before def".
- C. Corey Capel spotted a missing word and a typo in Chapter 4.
- Alessandra helped clear up some Turtle confusion.
- Wim Champagne found a braino in a dictionary example.
- Douglas Wright pointed out a problem with floor division in `arc`.
- Jared Spindor found some jetsam at the end of a sentence.
- Lin Peiheng sent a number of very helpful suggestions.
- Ray Hagtvedt sent in two errors and a not-quite-error.
- Torsten Hübsch pointed out an inconsistency in Swampy.
- Inga Petuhhov corrected an example in Chapter 14.

- Arne Babenhauserheide sent several helpful corrections.
- Mark E. Casida is is good at spotting repeated words.
- Scott Tyler filled in a that was missing. And then sent in a heap of corrections.
- Gordon Shephard sent in several corrections, all in separate emails.
- Andrew Turner spotted an error in Chapter 8.
- Adam Hobart fixed a problem with floor division in `arc`.
- Daryl Hammond and Sarah Zimmerman pointed out that I served up `math.pi` too early. And Zim spotted a typo.
- George Sass found a bug in a Debugging section.
- Brian Bingham suggested Exercise 11-5.
- Leah Engelbert-Fenton pointed out that I used `tuple` as a variable name, contrary to my own advice. And then found a bunch of typos and a "use before def".
- Joe Funke spotted a typo.
- Chao-chao Chen found an inconsistency in the Fibonacci example.
- Jeff Paine knows the difference between space and spam.
- Lubos Pintes sent in a typo.
- Gregg Lind and Abigail Heithoff suggested Exercise 14-3.
- Max Hailperin has sent in a number of corrections and suggestions. Max is one of the authors of the extraordinary *Concrete Abstractions* (Course Technology, 1998), which you might want to read when you are done with this book.
- Chotipat Pornavalai found an error in an error message.
- Stanislaw Antol sent a list of very helpful suggestions.
- Eric Pashman sent a number of corrections for Chapters 4–11.
- Miguel Azevedo found some typos.
- Jianhua Liu sent in a long list of corrections.
- Nick King found a missing word.
- Martin Zuther sent a long list of suggestions.
- Adam Zimmerman found an inconsistency in my instance of an "instance" and several other errors.
- Ratnakar Tiwari suggested a footnote explaining degenerate triangles.
- Anurag Goel suggested another solution for `is_abecedarian` and sent some additional corrections. And he knows how to spell Jane Austen.
- Kelli Kratzer spotted one of the typos.
- Mark Griffiths pointed out a confusing example in Chapter 3.
- Roydan Ongie found an error in my Newton's method.
- Patryk Wolowiec helped me with a problem in the HTML version.
- Mark Chonofsky told me about a new keyword in Python 3.
- Russell Coleman helped me with my geometry.

- Wei Huang spotted several typographical errors.
- Karen Barber spotted the the oldest typo in the book.
- Nam Nguyen found a typo and pointed out that I used the Decorator pattern but didn't mention it by name.
- Stéphane Morin sent in several corrections and suggestions.
- Paul Stoop corrected a typo in `uses_only`.
- Eric Bronner pointed out a confusion in the discussion of the order of operations.
- Alexandros Gezerlis set a new standard for the number and quality of suggestions he submitted. We are deeply grateful!
- Gray Thomas knows his right from his left.
- Giovanni Escobar Sosa sent a long list of corrections and suggestions.
- Alix Etienne fixed one of the URLs.
- Kuang He found a typo.
- Daniel Neilson corrected an error about the order of operations.
- Will McGinnis pointed out that `polyline` was defined differently in two places.
- Swarup Sahoo spotted a missing semicolon.
- Frank Hecker pointed out an exercise that was under-specified, and some broken links.
- Animesh B helped me clean up a confusing example.
- Martin Caspersen found two round-off errors.
- Gregor Ulm sent several corrections and suggestions.
- Dimitrios Tsirigkas suggested I clarify an exercise.
- Carlos Tafur sent a page of corrections and suggestions.
- Martin Nordsletten found a bug in an exercise solution.
- Lars O.D. Christensen found a broken reference.
- Victor Simeone found a typo.
- Sven Hoexter pointed out that a variable named `input` shadows a build-in function.
- Viet Le found a typo.
- Stephen Gregory pointed out the problem with `cmp` in Python 3.
- Matthew Shultz let me know about a broken link.
- Lokesh Kumar Makani let me know about some broken links and some changes in error messages.
- Ishwar Bhat corrected my statement of Fermat's last theorem.
- Brian McGhie suggested a clarification.
- Andrea Zanella translated the book into Italian, and sent a number of corrections along the way.
- Many, many thanks to Melissa Lewis and Luciano Ramalho for excellent comments and suggestions on the second edition.

- Thanks to Harry Percival from PythonAnywhere for his help getting people started running Python in a browser.
- Xavier Van Aubel made several useful corrections in the second edition.
- Laurent Rosenfeld and Mihaela Rotaru translated this book into French. Along the way, they sent many corrections and suggestions.

CHAPTER 1

The Way of the Program

The goal of this book is to teach you to think like a computer scientist. This way of thinking combines some of the best features of mathematics, engineering, and natural science. Like mathematicians, computer scientists use formal languages to denote ideas (specifically computations). Like engineers, they design things, assembling components into systems and evaluating tradeoffs among alternatives. Like scientists, they observe the behavior of complex systems, form hypotheses, and test predictions.

The single most important skill for a computer scientist is **problem solving**. Problem solving means the ability to formulate problems, think creatively about solutions, and express a solution clearly and accurately. As it turns out, the process of learning to program is an excellent opportunity to practice problem-solving skills. That's why this chapter is called "The Way of the Program".

On one level, you will be learning to program, a useful skill by itself. On another level, you will use programming as a means to an end. As we go along, that end will become clearer.

What Is a Program?

A **program** is a sequence of instructions that specifies how to perform a computation. The computation might be something mathematical, such as solving a system of equations or finding the roots of a polynomial, but it can also be a symbolic computation, such as searching and replacing text in a document or something graphical, like processing an image or playing a video.

The details look different in different languages, but a few basic instructions appear in just about every language:

input:
: Get data from the keyboard, a file, the network, or some other device.

output:
: Display data on the screen, save it in a file, send it over the network, etc.

math:
: Perform basic mathematical operations like addition and multiplication.

conditional execution:
: Check for certain conditions and run the appropriate code.

repetition:
: Perform some action repeatedly, usually with some variation.

Believe it or not, that's pretty much all there is to it. Every program you've ever used, no matter how complicated, is made up of instructions that look pretty much like these. So you can think of programming as the process of breaking a large, complex task into smaller and smaller subtasks until the subtasks are simple enough to be performed with one of these basic instructions.

Running Python

One of the challenges of getting started with Python is that you might have to install Python and related software on your computer. If you are familiar with your operating system, and especially if you are comfortable with the command-line interface, you will have no trouble installing Python. But for beginners, it can be painful to learn about system administration and programming at the same time.

To avoid that problem, I recommend that you start out running Python in a browser. Later, when you are comfortable with Python, I'll make suggestions for installing Python on your computer.

There are a number of web pages you can use to run Python. If you already have a favorite, go ahead and use it. Otherwise I recommend PythonAnywhere. I provide detailed instructions for getting started at *http://tinyurl.com/thinkpython2e*.

There are two versions of Python, called Python 2 and Python 3. They are very similar, so if you learn one, it is easy to switch to the other. In fact, there are only a few differences you will encounter as a beginner. This book is written for Python 3, but I include some notes about Python 2.

The Python **interpreter** is a program that reads and executes Python code. Depending on your environment, you might start the interpreter by clicking on an icon, or by typing `python` on a command line. When it starts, you should see output like this:

```
Python 3.4.0 (default, Jun 19 2015, 14:20:21)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The first three lines contain information about the interpreter and the operating system it's running on, so it might be different for you. But you should check that the version number, which is `3.4.0` in this example, begins with 3, which indicates that you are running Python 3. If it begins with 2, you are running (you guessed it) Python 2.

The last line is a **prompt** that indicates that the interpreter is ready for you to enter code. If you type a line of code and hit Enter, the interpreter displays the result:

```
>>> 1 + 1
2
```

Now you're ready to get started. From here on, I assume that you know how to start the Python interpreter and run code.

The First Program

Traditionally, the first program you write in a new language is called "Hello, World!" because all it does is display the words "Hello, World!" In Python, it looks like this:

```
>>> print('Hello, World!')
```

This is an example of a **print statement**, although it doesn't actually print anything on paper. It displays a result on the screen. In this case, the result is the words

```
Hello, World!
```

The quotation marks in the program mark the beginning and end of the text to be displayed; they don't appear in the result.

The parentheses indicate that `print` is a function. We'll get to functions in Chapter 3.

In Python 2, the print statement is slightly different; it is not a function, so it doesn't use parentheses.

```
>>> print 'Hello, World!'
```

This distinction will make more sense soon, but that's enough to get started.

Arithmetic Operators

After "Hello, World", the next step is arithmetic. Python provides **operators**, which are special symbols that represent computations like addition and multiplication.

The operators `+`, `-`, and `*` perform addition, subtraction, and multiplication, as in the following examples:

```
>>> 40 + 2
42
>>> 43 - 1
42
>>> 6 * 7
42
```

The operator `/` performs division:

```
>>> 84 / 2
42.0
```

You might wonder why the result is `42.0` instead of `42`. I'll explain in the next section.

Finally, the operator `**` performs exponentiation; that is, it raises a number to a power:

```
>>> 6**2 + 6
42
```

In some other languages, `^` is used for exponentiation, but in Python it is a bitwise operator called XOR. If you are not familiar with bitwise operators, the result will surprise you:

```
>>> 6 ^ 2
4
```

I won't cover bitwise operators in this book, but you can read about them at *http://wiki.python.org/moin/BitwiseOperators*.

Values and Types

A **value** is one of the basic things a program works with, like a letter or a number. Some values we have seen so far are `2`, `42.0`, and `'Hello, World!'`

These values belong to different **types**: `2` is an **integer**, `42.0` is a **floating-point number**, and `'Hello, World!'` is a **string**, so-called because the letters it contains are strung together.

If you are not sure what type a value has, the interpreter can tell you:

```
>>> type(2)
<class 'int'>
>>> type(42.0)
<class 'float'>
>>> type('Hello, World!')
<class 'str'>
```

In these results, the word "class" is used in the sense of a category; a type is a category of values.

Not surprisingly, integers belong to the type `int`, strings belong to `str`, and floating-point numbers belong to `float`.

What about values like `'2'` and `'42.0'`? They look like numbers, but they are in quotation marks like strings:

```
>>> type('2')
<class 'str'>
>>> type('42.0')
<class 'str'>
```

They're strings.

When you type a large integer, you might be tempted to use commas between groups of digits, as in `1,000,000`. This is not a legal *integer* in Python, but it is legal:

```
>>> 1,000,000
(1, 0, 0)
```

That's not what we expected at all! Python interprets `1,000,000` as a comma-separated sequence of integers. We'll learn more about this kind of sequence later.

Formal and Natural Languages

Natural languages are the languages people speak, such as English, Spanish, and French. They were not designed by people (although people try to impose some order on them); they evolved naturally.

Formal languages are languages that are designed by people for specific applications. For example, the notation that mathematicians use is a formal language that is particularly good at denoting relationships among numbers and symbols. Chemists use a formal language to represent the chemical structure of molecules. And most importantly:

> **Programming languages are formal languages that have been designed to express computations.**

Formal languages tend to have strict **syntax** rules that govern the structure of statements. For example, in mathematics the statement 3 + 3 = 6 has correct syntax, but 3 + = 3$6 does not. In chemistry H_2O is a syntactically correct formula, but ${}_2Zz$ is not.

Syntax rules come in two flavors, pertaining to **tokens** and structure. Tokens are the basic elements of the language, such as words, numbers, and chemical elements. One of the problems with 3 + = 3$6 is that $ is not a legal token in mathematics (at least as far as I know). Similarly, ${}_2Zz$ is not legal because there is no element with the abbreviation *Zz*.

The second type of syntax rule pertains to the way tokens are combined. The equation 3 + /3 is illegal because even though + and / are legal tokens, you can't have one right after the other. Similarly, in a chemical formula the subscript comes after the element name, not before.

This is @ well-structured Engli$h sentence with invalid t*kens in it. This sentence all valid tokens has, but invalid structure with.

When you read a sentence in English or a statement in a formal language, you have to figure out the structure (although in a natural language you do this subconsciously). This process is called **parsing**.

Although formal and natural languages have many features in common—tokens, structure, and syntax—there are some differences:

ambiguity:
: Natural languages are full of ambiguity, which people deal with by using contextual clues and other information. Formal languages are designed to be nearly or completely unambiguous, which means that any statement has exactly one meaning, regardless of context.

redundancy:
: In order to make up for ambiguity and reduce misunderstandings, natural languages employ lots of redundancy. As a result, they are often verbose. Formal languages are less redundant and more concise.

literalness:
: Natural languages are full of idiom and metaphor. If I say, "The penny dropped", there is probably no penny and nothing dropping (this idiom means that someone understood something after a period of confusion). Formal languages mean exactly what they say.

Because we all grow up speaking natural languages, it is sometimes hard to adjust to formal languages. The difference between formal and natural language is like the difference between poetry and prose, but more so:

Poetry:
: Words are used for their sounds as well as for their meaning, and the whole poem together creates an effect or emotional response. Ambiguity is not only common but often deliberate.

Prose:
: The literal meaning of words is more important, and the structure contributes more meaning. Prose is more amenable to analysis than poetry but still often ambiguous.

Programs:

> The meaning of a computer program is unambiguous and literal, and can be understood entirely by analysis of the tokens and structure.

Formal languages are more dense than natural languages, so it takes longer to read them. Also, the structure is important, so it is not always best to read from top to bottom, left to right. Instead, learn to parse the program in your head, identifying the tokens and interpreting the structure. Finally, the details matter. Small errors in spelling and punctuation, which you can get away with in natural languages, can make a big difference in a formal language.

Debugging

Programmers make mistakes. For whimsical reasons, programming errors are called **bugs** and the process of tracking them down is called **debugging**.

Programming, and especially debugging, sometimes brings out strong emotions. If you are struggling with a difficult bug, you might feel angry, despondent, or embarrassed.

There is evidence that people naturally respond to computers as if they were people. When they work well, we think of them as teammates, and when they are obstinate or rude, we respond to them the same way we respond to rude, obstinate people (Reeves and Nass, *The Media Equation: How People Treat Computers, Television, and New Media Like Real People and Places*).

Preparing for these reactions might help you deal with them. One approach is to think of the computer as an employee with certain strengths, like speed and precision, and particular weaknesses, like lack of empathy and inability to grasp the big picture.

Your job is to be a good manager: find ways to take advantage of the strengths and mitigate the weaknesses. And find ways to use your emotions to engage with the problem, without letting your reactions interfere with your ability to work effectively.

Learning to debug can be frustrating, but it is a valuable skill that is useful for many activities beyond programming. At the end of each chapter there is a section, like this one, with my suggestions for debugging. I hope they help!

Glossary

problem solving:
: The process of formulating a problem, finding a solution, and expressing it.

high-level language:
: A programming language like Python that is designed to be easy for humans to read and write.

low-level language:
: A programming language that is designed to be easy for a computer to run; also called "machine language" or "assembly language".

portability:
: A property of a program that can run on more than one kind of computer.

interpreter:
: A program that reads another program and executes it.

prompt:
: Characters displayed by the interpreter to indicate that it is ready to take input from the user.

program:
: A set of instructions that specifies a computation.

print statement:
: An instruction that causes the Python interpreter to display a value on the screen.

operator:
: A special symbol that represents a simple computation like addition, multiplication, or string concatenation.

value:
: One of the basic units of data, like a number or string, that a program manipulates.

type:
: A category of values. The types we have seen so far are integers (type `int`), floating-point numbers (type `float`), and strings (type `str`).

integer:
: A type that represents whole numbers.

floating-point:
: A type that represents numbers with fractional parts.