HOW TO THINK LIKE A

Computer Scientist

Learning with Python

Allen Downey, Jeffrey Elkner & Chris Meyers

How to Think Like a Computer Scientist

Learning with Python

How to Think Like a Computer Scientist

Learning with Python

Allen Downey
Jeffrey Elkner
Chris Meyers

Green Tea Press

Wellesley, Massachusetts

Copyright © 2002 Allen Downey, Jeffrey Elkner, and Chris Meyers.

Edited by Shannon Turlington and Lisa Cutler. Cover design by Rebecca Gimenez.

Printing history:

April 2002: First edition.

August 2008: Second printing.

Green Tea Press
1 Grove St.
P.O. Box 812901
Wellesley, MA 02482

Permission is granted to copy, distribute, and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being "Foreword," "Preface," and "Contributor List," with no Front-Cover Texts, and with no Back-Cover Texts. A copy of the license is included in the appendix entitled "GNU Free Documentation License."

The GNU Free Documentation License is available from `www.gnu.org` or by writing to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307, USA.

The original form of this book is LaTeX source code. Compiling this LaTeX source has the effect of generating a device-independent representation of a textbook, which can be converted to other formats and printed.

The LaTeX source for this book is available from `http://www.thinkpython.com`

Publisher's Cataloging-in-Publication (provided by Quality Books, Inc.)

Downey, Allen
How to think like a computer scientist : learning with Python / Allen Downey, Jeffrey Elkner, Chris Meyers. – 1st ed.
p. cm.
Includes index.
ISBN 0-9716775-0-6
LCCN 2002100618

1. Python (Computer program language) I. Elkner, Jeffrey. II. Meyers, Chris. III. Title

QA76.73.P98D69 2002 005.13'3
QBI02-200031

Foreword

By David Beazley

As an educator, researcher, and book author, I am delighted to see the completion of this book. Python is a fun and extremely easy-to-use programming language that has steadily gained in popularity over the last few years. Developed over ten years ago by Guido van Rossum, Python's simple syntax and overall feel is largely derived from ABC, a teaching language that was developed in the 1980's. However, Python was also created to solve real problems and it borrows a wide variety of features from programming languages such as C++, Java, Modula-3, and Scheme. Because of this, one of Python's most remarkable features is its broad appeal to professional software developers, scientists, researchers, artists, and educators.

Despite Python's appeal to many different communities, you may still wonder "why Python?" or "why teach programming with Python?" Answering these questions is no simple task—especially when popular opinion is on the side of more masochistic alternatives such as C++ and Java. However, I think the most direct answer is that programming in Python is simply a lot of fun and more productive.

When I teach computer science courses, I want to cover important concepts in addition to making the material interesting and engaging to students. Unfortunately, there is a tendency for introductory programming courses to focus far too much attention on mathematical abstraction and for students to become frustrated with annoying problems related to low-level details of syntax, compilation, and the enforcement of seemingly arcane rules. Although such abstraction and formalism is important to professional software engineers and students who plan to continue their study of computer science, taking such an approach in an introductory course mostly succeeds in making computer science boring. When I teach a course, I don't want to have a room of uninspired students. I would much rather see them trying to solve interesting problems by exploring different ideas, taking unconventional approaches, breaking the rules, and learning from their mistakes.

In doing so, I don't want to waste half of the semester trying to sort out obscure syntax problems, unintelligible compiler error messages, or the several hundred ways that a program might generate a general protection fault.

One of the reasons why I like Python is that it provides a really nice balance between the practical and the conceptual. Since Python is interpreted, beginners can pick up the language and start doing neat things almost immediately without getting lost in the problems of compilation and linking. Furthermore, Python comes with a large library of modules that can be used to do all sorts of tasks ranging from web-programming to graphics. Having such a practical focus is a great way to engage students and it allows them to complete significant projects. However, Python can also serve as an excellent foundation for introducing important computer science concepts. Since Python fully supports procedures and classes, students can be gradually introduced to topics such as procedural abstraction, data structures, and object-oriented programming—all of which are applicable to later courses on Java or C++. Python even borrows a number of features from functional programming languages and can be used to introduce concepts that would be covered in more detail in courses on Scheme and Lisp.

In reading Jeffrey's preface, I am struck by his comments that Python allowed him to see a "higher level of success and a lower level of frustration" and that he was able to "move faster with better results." Although these comments refer to his introductory course, I sometimes use Python for these exact same reasons in advanced graduate level computer science courses at the University of Chicago. In these courses, I am constantly faced with the daunting task of covering a lot of difficult course material in a blistering nine week quarter. Although it is certainly possible for me to inflict a lot of pain and suffering by using a language like C++, I have often found this approach to be counterproductive—especially when the course is about a topic unrelated to just "programming." I find that using Python allows me to better focus on the actual topic at hand while allowing students to complete substantial class projects.

Although Python is still a young and evolving language, I believe that it has a bright future in education. This book is an important step in that direction.

David Beazley
University of Chicago
Author of the *Python Essential Reference*

Preface

By Jeff Elkner

This book owes its existence to the collaboration made possible by the Internet and the free software movement. Its three authors—a college professor, a high school teacher, and a professional programmer—have yet to meet face to face, but we have been able to work closely together and have been aided by many wonderful folks who have donated their time and energy to helping make this book better.

We think this book is a testament to the benefits and future possibilities of this kind of collaboration, the framework for which has been put in place by Richard Stallman and the Free Software Foundation.

How and why I came to use Python

In 1999, the College Board's Advanced Placement (AP) Computer Science exam was given in C++ for the first time. As in many high schools throughout the country, the decision to change languages had a direct impact on the computer science curriculum at Yorktown High School in Arlington, Virginia, where I teach. Up to this point, Pascal was the language of instruction in both our first-year and AP courses. In keeping with past practice of giving students two years of exposure to the same language, we made the decision to switch to C++ in the first-year course for the 1997-98 school year so that we would be in step with the College Board's change for the AP course the following year.

Two years later, I was convinced that C++ was a poor choice to use for introducing students to computer science. While it is certainly a very powerful programming language, it is also an extremely difficult language to learn and teach. I found myself constantly fighting with C++'s difficult syntax and multiple ways of doing things, and I was losing many students unnecessarily as a result. Convinced there

had to be a better language choice for our first-year class, I went looking for an alternative to C++.

I needed a language that would run on the machines in our Linux lab as well as on the Windows and Macintosh platforms most students have at home. I wanted it to be free and available electronically, so that students could use it at home regardless of their income. I wanted a language that was used by professional programmers, and one that had an active developer community around it. It had to support both procedural and object-oriented programming. And most importantly, it had to be easy to learn and teach. When I investigated the choices with these goals in mind, Python stood out as the best candidate for the job.

I asked one of Yorktown's talented students, Matt Ahrens, to give Python a try. In two months he not only learned the language but wrote an application called pyTicket that enabled our staff to report technology problems via the Web. I knew that Matt could not have finished an application of that scale in so short a time in C++, and this accomplishment, combined with Matt's positive assessment of Python, suggested that Python was the solution I was looking for.

Finding a textbook

Having decided to use Python in both of my introductory computer science classes the following year, the most pressing problem was the lack of an available textbook.

Free content came to the rescue. Earlier in the year, Richard Stallman had introduced me to Allen Downey. Both of us had written to Richard expressing an interest in developing free educational content. Allen had already written a first-year computer science textbook, *How to Think Like a Computer Scientist.* When I read this book, I knew immediately that I wanted to use it in my class. It was the clearest and most helpful computer science text I had seen. It emphasized the processes of thought involved in programming rather than the features of a particular language. Reading it immediately made me a better teacher.

How to Think Like a Computer Scientist was not just an excellent book, but it had been released under a GNU public license, which meant it could be used freely and modified to meet the needs of its user. Once I decided to use Python, it occurred to me that I could translate Allen's original Java version of the book into the new language. While I would not have been able to write a textbook on my own, having Allen's book to work from made it possible for me to do so, at the same time demonstrating that the cooperative development model used so well in software could also work for educational content.

Working on this book for the last two years has been rewarding for both my students and me, and my students played a big part in the process. Since I could

make instant changes whenever someone found a spelling error or difficult passage, I encouraged them to look for mistakes in the book by giving them a bonus point each time they made a suggestion that resulted in a change in the text. This had the double benefit of encouraging them to read the text more carefully and of getting the text thoroughly reviewed by its most important critics, students using it to learn computer science.

For the second half of the book on object-oriented programming, I knew that someone with more real programming experience than I had would be needed to do it right. The book sat in an unfinished state for the better part of a year until the free software community once again provided the needed means for its completion.

I received an email from Chris Meyers expressing interest in the book. Chris is a professional programmer who started teaching a programming course last year using Python at Lane Community College in Eugene, Oregon. The prospect of teaching the course had led Chris to the book, and he started helping out with it immediately. By the end of the school year he had created a companion project on our website at `http://www.ibiblio.org/obp` called *Python for Fun* and was working with some of my most advanced students as a master teacher, guiding them beyond where I could take them.

Introducing programming with Python

The process of translating and using *How to Think Like a Computer Scientist* for the past two years has confirmed Python's suitability for teaching beginning students. Python greatly simplifies programming examples and makes important programming ideas easier to teach.

The first example from the text illustrates this point. It is the traditional "hello, world" program, which in the C++ version of the book looks like this:

```
#include <iostream.h>

void main()
{
  cout << "Hello, world." << endl;
}
```

in the Python version it becomes:

```
print "Hello, World!"
```

Even though this is a trivial example, the advantages of Python stand out. Yorktown's Computer Science I course has no prerequisites, so many of the students

seeing this example are looking at their first program. Some of them are undoubtedly a little nervous, having heard that computer programming is difficult to learn. The C++ version has always forced me to choose between two unsatisfying options: either to explain `#include`, `void main()`, {, and }, and risk confusing or intimidating some of the students right at the start, or to tell them, "Just don't worry about all of that stuff now; we will talk about it later," and risk the same thing. The educational objectives at this point in the course are to introduce students to the idea of a programming language and to get them to write their first program, thereby introducing them to the programming environment. The Python program has exactly what is needed to do these things, and nothing more.

Comparing the explanatory text of the program in each version of the book further illustrates what this means to the beginning student. There are thirteen paragraphs of explanation of "Hello, world!" in the C++ version; in the Python version, there are only two. More importantly, the missing eleven paragraphs do not deal with the "big ideas" in computer programming but with the minutia of C++ syntax. I found this same thing happening throughout the book. Whole paragraphs simply disappear from the Python version of the text because Python's much clearer syntax renders them unnecessary.

Using a very high-level language like Python allows a teacher to postpone talking about low-level details of the machine until students have the background that they need to better make sense of the details. It thus creates the ability to put "first things first" pedagogically. One of the best examples of this is the way in which Python handles variables. In C++ a variable is a name for a place that holds a thing. Variables have to be declared with types at least in part because the size of the place to which they refer needs to be predetermined. Thus, the idea of a variable is bound up with the hardware of the machine. The powerful and fundamental concept of a variable is already difficult enough for beginning students (in both computer science and algebra). Bytes and addresses do not help the matter. In Python a variable is a name that refers to a thing. This is a far more intuitive concept for beginning students and is much closer to the meaning of "variable" that they learned in their math courses. I had much less difficulty teaching variables this year than I did in the past, and I spent less time helping students with problems using them.

Another example of how Python aids in the teaching and learning of programming is in its syntax for functions. My students have always had a great deal of difficulty understanding functions. The main problem centers around the difference between a function definition and a function call, and the related distinction between a parameter and an argument. Python comes to the rescue with syntax that is nothing short of beautiful. Function definitions begin with the keyword `def`, so I simply tell my students, "When you define a function, begin with `def`, followed by the name of the function that you are defining; when you call a function, simply

call (type) out its name." Parameters go with definitions; arguments go with calls. There are no return types, parameter types, or reference and value parameters to get in the way, so I am now able to teach functions in less than half the time that it previously took me, with better comprehension.

Using Python has improved the effectiveness of our computer science program for all students. I see a higher general level of success and a lower level of frustration than I experienced during the two years I taught C++. I move faster with better results. More students leave the course with the ability to create meaningful programs and with the positive attitude toward the experience of programming that this engenders.

Building a community

I have received email from all over the globe from people using this book to learn or to teach programming. A user community has begun to emerge, and many people have been contributing to the project by sending in materials for the companion website at `http://www.thinkpython.com`.

With the publication of the book in print form, I expect the growth in the user community to continue and accelerate. The emergence of this user community and the possibility it suggests for similar collaboration among educators have been the most exciting parts of working on this project for me. By working together, we can increase the quality of materials available for our use and save valuable time. I invite you to join our community and look forward to hearing from you. Please write to the authors at `feedback@thinkpython.com`.

Jeffrey Elkner
Yorktown High School
Arlington, Virginia

Contributor List

To paraphrase the philosophy of the Free Software Foundation, this book is free like free speech, but not necessarily free like free pizza. It came about because of a collaboration that would not have been possible without the GNU Free Documentation License. So we thank the Free Software Foundation for developing this license and, of course, making it available to us.

We also thank the more than 100 sharp-eyed and thoughtful readers who have sent us suggestions and corrections over the past few years. In the spirit of free software, we decided to express our gratitude in the form of a contributor list. Unfortunately, this list is not complete, but we are doing our best to keep it up to date.

If you have a chance to look through the list, you should realize that each person here has spared you and all subsequent readers from the confusion of a technical error or a less-than-transparent explanation, just by sending us a note.

Impossible as it may seem after so many corrections, there may still be errors in this book. If you should stumble across one, please check the online version of the book at `http://thinkpython.com`, which is the most up-to-date version. If the error has not been corrected, please take a minute to send us email at `feedback@thinkpython.com`. If we make a change due to your suggestion, you will appear in the next version of the contributor list (unless you ask to be omitted). Thank you!

- 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 is helping us prepare to 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 semi-colons 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."
- Hans Batra found an error in Chapter 16.
- Chris Seberino suggested some improvements in the Preface.
- Yuri Takhteyev pointed out a problem with single and double quotes.

Contents

Foreword **v**

Preface **vii**

Contributor List **xiii**

1 The way of the program **1**

1.1 The Python programming language 1

1.2 What is a program? 3

1.3 What is debugging? 4

1.4 Formal and natural languages 6

1.5 The first program 8

1.6 Glossary 8

2 Variables, expressions and statements **11**

2.1 Values and types 11

2.2 Variables 12

2.3 Variable names and keywords 13

2.4 Statements 15

2.5 Evaluating expressions 16

2.6 Operators and operands 17

2.7 Order of operations 17
2.8 Operations on strings 18
2.9 Composition 19
2.10 Comments 19
2.11 Glossary 20

3 Functions 23

3.1 Function calls 23
3.2 Type conversion 24
3.3 Type coercion 24
3.4 Math functions 25
3.5 Composition 26
3.6 Adding new functions 26
3.7 Definitions and use 29
3.8 Flow of execution 29
3.9 Parameters and arguments 30
3.10 Variables and parameters are local 31
3.11 Stack diagrams 32
3.12 Functions with results 33
3.13 Glossary 34

4 Conditionals and recursion 37

4.1 The modulus operator 37
4.2 Boolean expressions 37
4.3 Logical operators 38
4.4 Conditional execution 39
4.5 Alternative execution 39
4.6 Chained conditionals 40

4.7 Nested conditionals 41
4.8 The `return` statement 42
4.9 Recursion 42
4.10 Stack diagrams for recursive functions 44
4.11 Infinite recursion 45
4.12 Keyboard input 45
4.13 Glossary 46

5 Fruitful functions 49

5.1 Return values 49
5.2 Program development 50
5.3 Composition 53
5.4 Boolean functions 54
5.5 More recursion 55
5.6 Leap of faith 57
5.7 One more example 58
5.8 Checking types 58
5.9 Glossary 60

6 Iteration 61

6.1 Multiple assignment 61
6.2 The `while` statement 62
6.3 Tables 64
6.4 Two-dimensional tables 66
6.5 Encapsulation and generalization 67
6.6 More encapsulation 68
6.7 Local variables 69
6.8 More generalization 70
6.9 Functions 71
6.10 Glossary 72

7 Strings **73**

7.1 A compound data type 73
7.2 Length 74
7.3 Traversal and the `for` loop 74
7.4 String slices 76
7.5 String comparison 76
7.6 Strings are immutable 77
7.7 A `find` function 78
7.8 Looping and counting 78
7.9 The `string` module 79
7.10 Character classification 80
7.11 Glossary 81

8 Lists **83**

8.1 List values 83
8.2 Accessing elements 84
8.3 List length 85
8.4 List membership 86
8.5 Lists and `for` loops 86
8.6 List operations 87
8.7 List slices 88
8.8 Lists are mutable 88
8.9 List deletion 89
8.10 Objects and values 91
8.11 Aliasing 92
8.12 Cloning lists 92
8.13 List parameters 93
8.14 Nested lists 94

8.15 Matrices 94
8.16 Strings and lists 95
8.17 Glossary 96

9 Tuples 97

9.1 Mutability and tuples 97
9.2 Tuple assignment 98
9.3 Tuples as return values 99
9.4 Random numbers 99
9.5 List of random numbers 100
9.6 Counting 101
9.7 Many buckets 102
9.8 A single-pass solution 104
9.9 Glossary 105

10 Dictionaries 107

10.1 Dictionary operations 108
10.2 Dictionary methods 109
10.3 Aliasing and copying 110
10.4 Sparse matrices 110
10.5 Hints 111
10.6 Long integers 113
10.7 Counting letters 113
10.8 Glossary 114

11 Files and exceptions 117

11.1 Text files 119
11.2 Writing variables 120
11.3 Directories 123

11.4 Pickling 123
11.5 Exceptions 124
11.6 Glossary 126

12 Classes and objects **129**
12.1 User-defined compound types 129
12.2 Attributes 130
12.3 Instances as arguments 131
12.4 Sameness 131
12.5 Rectangles 133
12.6 Instances as return values 134
12.7 Objects are mutable 134
12.8 Copying 135
12.9 Glossary 137

13 Classes and functions **139**
13.1 Time 139
13.2 Pure functions 140
13.3 Modifiers 141
13.4 Which is better? 142
13.5 Prototype development versus planning 143
13.6 Generalization 144
13.7 Algorithms 144
13.8 Glossary 145

14 Classes and methods **147**
14.1 Object-oriented features 147
14.2 `printTime` 148
14.3 Another example 149

14.4 A more complicated example . . . 150
14.5 Optional arguments . . . 151
14.6 The initialization method . . . 152
14.7 Points revisited . . . 153
14.8 Operator overloading . . . 154
14.9 Polymorphism . . . 155
14.10 Glossary . . . 157

15 Sets of objects **159**
15.1 Composition . . . 159
15.2 `Card` objects . . . 159
15.3 Class attributes and the `__str__` method . . . 161
15.4 Comparing cards . . . 162
15.5 Decks . . . 163
15.6 Printing the deck . . . 163
15.7 Shuffling the deck . . . 165
15.8 Removing and dealing cards . . . 166
15.9 Glossary . . . 167

16 Inheritance **169**
16.1 Inheritance . . . 169
16.2 A hand of cards . . . 170
16.3 Dealing cards . . . 171
16.4 Printing a Hand . . . 171
16.5 The `CardGame` class . . . 172
16.6 `OldMaidHand` class . . . 173
16.7 `OldMaidGame` class . . . 175
16.8 Glossary . . . 179

17 Linked lists **181**

17.1 Embedded references . . . 181
17.2 The `Node` class . . . 181
17.3 Lists as collections . . . 183
17.4 Lists and recursion . . . 184
17.5 Infinite lists . . . 185
17.6 The fundamental ambiguity theorem . . . 186
17.7 Modifying lists . . . 186
17.8 Wrappers and helpers . . . 187
17.9 The `LinkedList` class . . . 188
17.10 Invariants . . . 189
17.11 Glossary . . . 190

18 Stacks **191**

18.1 Abstract data types . . . 191
18.2 The Stack ADT . . . 192
18.3 Implementing stacks with Python lists . . . 192
18.4 Pushing and popping . . . 193
18.5 Using a stack to evaluate postfix . . . 194
18.6 Parsing . . . 194
18.7 Evaluating postfix . . . 195
18.8 Clients and providers . . . 196
18.9 Glossary . . . 197

19 Queues **199**

19.1 The Queue ADT . . . 199
19.2 Linked Queue . . . 200
19.3 Performance characteristics . . . 201

19.4 Improved Linked Queue 201
19.5 Priority queue 203
19.6 The `Golfer` class 205
19.7 Glossary 206

20 Trees 207

20.1 Building trees 208
20.2 Traversing trees 209
20.3 Expression trees 209
20.4 Tree traversal 210
20.5 Building an expression tree 212
20.6 Handling errors 216
20.7 The animal tree 216
20.8 Glossary 219

A Debugging 221

A.1 Syntax errors 221
A.2 Runtime errors 223
A.3 Semantic errors 227

B Creating a new data type 231

B.1 Fraction multiplication 232
B.2 Fraction addition 234
B.3 Euclid's algorithm 234
B.4 Comparing fractions 235
B.5 Taking it further 236
B.6 Glossary 236

C Recommendations for further reading 239

C.1 Python-related web sites and books 240
C.2 Recommended general computer science books 241

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.

1.1 The Python programming language

The programming language you will be learning is Python. Python is an example of a **high-level language**; other high-level languages you might have heard of are C, C++, Perl, and Java.

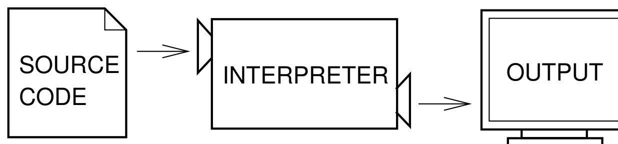
As you might infer from the name "high-level language," there are also **low-level languages**, sometimes referred to as "machine languages" or "assembly

languages." Loosely speaking, computers can only execute programs written in low-level languages. Thus, programs written in a high-level language have to be processed before they can run. This extra processing takes some time, which is a small disadvantage of high-level languages.

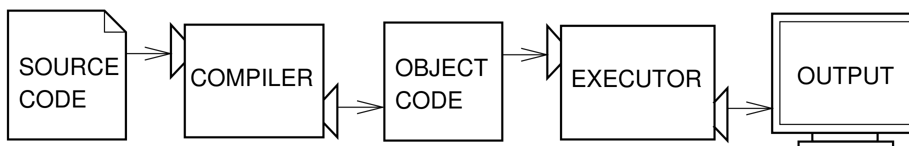
But the advantages are enormous. First, it is much easier to program in a high-level language. Programs written in a high-level language take less time to write, they are shorter and easier to read, and they are more likely to be correct. Second, high-level languages are **portable**, meaning that they can run on different kinds of computers with few or no modifications. Low-level programs can run on only one kind of computer and have to be rewritten to run on another.

Due to these advantages, almost all programs are written in high-level languages. Low-level languages are used only for a few specialized applications.

Two kinds of programs process high-level languages into low-level languages: **interpreters** and **compilers**. An interpreter reads a high-level program and executes it, meaning that it does what the program says. It processes the program a little at a time, alternately reading lines and performing computations.



A compiler reads the program and translates it completely before the program starts running. In this case, the high-level program is called the **source code**, and the translated program is called the **object code** or the **executable**. Once a program is compiled, you can execute it repeatedly without further translation.



Python is considered an interpreted language because Python programs are executed by an interpreter. There are two ways to use the interpreter: command-line mode and script mode. In command-line mode, you type Python programs and the interpreter prints the result:

```
$ python
Python 2.4.1 (#1, Apr 29 2005, 00:28:56)
Type "help", "copyright", "credits" or "license" for more information.
>>> print 1 + 1
2
```

The first line of this example is the command that starts the Python interpreter. The next two lines are messages from the interpreter. The third line starts with `>>>`, which is the prompt the interpreter uses to indicate that it is ready. We typed `print 1 + 1`, and the interpreter replied `2`.

Alternatively, you can write a program in a file and use the interpreter to execute the contents of the file. Such a file is called a **script**. For example, we used a text editor to create a file named `latoya.py` with the following contents:

```
print 1 + 1
```

By convention, files that contain Python programs have names that end with `.py`.

To execute the program, we have to tell the interpreter the name of the script:

```
$ python latoya.py
2
```

In other development environments, the details of executing programs may differ. Also, most programs are more interesting than this one.

Most of the examples in this book are executed on the command line. Working on the command line is convenient for program development and testing, because you can type programs and execute them immediately. Once you have a working program, you should store it in a script so you can execute or modify it in the future.

1.2 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 (strangely enough) compiling a program.

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, or some other device.

output: Display data on the screen or send data to a file or other device.

math: Perform basic mathematical operations like addition and multiplication.

conditional execution: Check for certain conditions and execute the appropriate sequence of statements.

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 more or less like these. Thus, we can describe 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.

That may be a little vague, but we will come back to this topic later when we talk about **algorithms**.

1.3 What is debugging?

Programming is a complex process, and because it is done by human beings, it often leads to errors. For whimsical reasons, programming errors are called **bugs** and the process of tracking them down and correcting them is called **debugging**.

Three kinds of errors can occur in a program: syntax errors, runtime errors, and semantic errors. It is useful to distinguish between them in order to track them down more quickly.

1.3.1 Syntax errors

Python can only execute a program if the program is syntactically correct; otherwise, the process fails and returns an error message. **Syntax** refers to the structure of a program and the rules about that structure. For example, in English, a sentence must begin with a capital letter and end with a period. this sentence contains a **syntax error**. So does this one

For most readers, a few syntax errors are not a significant problem, which is why we can read the poetry of e. e. cummings without spewing error messages. Python is not so forgiving. If there is a single syntax error anywhere in your program, Python will print an error message and quit, and you will not be able to run your program. During the first few weeks of your programming career, you will probably spend a lot of time tracking down syntax errors. As you gain experience, though, you will make fewer errors and find them faster.

1.3.2 Runtime errors

The second type of error is a runtime error, so called because the error does not appear until you run the program. These errors are also called **exceptions** because they usually indicate that something exceptional (and bad) has happened.

Runtime errors are rare in the simple programs you will see in the first few chapters, so it might be a while before you encounter one.

1.3.3 Semantic errors

The third type of error is the **semantic error**. If there is a semantic error in your program, it will run successfully, in the sense that the computer will not generate any error messages, but it will not do the right thing. It will do something else. Specifically, it will do what you told it to do.

The problem is that the program you wrote is not the program you wanted to write. The meaning of the program (its semantics) is wrong. Identifying semantic errors can be tricky because it requires you to work backward by looking at the output of the program and trying to figure out what it is doing.

1.3.4 Experimental debugging

One of the most important skills you will acquire is debugging. Although it can be frustrating, debugging is one of the most intellectually rich, challenging, and interesting parts of programming.

In some ways, debugging is like detective work. You are confronted with clues, and you have to infer the processes and events that led to the results you see.

Debugging is also like an experimental science. Once you have an idea what is going wrong, you modify your program and try again. If your hypothesis was correct, then you can predict the result of the modification, and you take a step closer to a working program. If your hypothesis was wrong, you have to come up with a new one. As Sherlock Holmes pointed out, "When you have eliminated the impossible, whatever remains, however improbable, must be the truth." (A. Conan Doyle, *The Sign of Four*)

For some people, programming and debugging are the same thing. That is, programming is the process of gradually debugging a program until it does what you want. The idea is that you should start with a program that does *something* and make small modifications, debugging them as you go, so that you always have a working program.

For example, Linux is an operating system that contains thousands of lines of code, but it started out as a simple program Linus Torvalds used to explore the Intel 80386 chip. According to Larry Greenfield, "One of Linus's earlier projects was a program that would switch between printing AAAA and BBBB. This later evolved to Linux." (*The Linux Users' Guide* Beta Version 1)

Later chapters will make more suggestions about debugging and other programming practices.

1.4 Formal and natural languages

Natural languages are the languages that 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 rules about syntax. For example, $3 + 3 = 6$ is a syntactically correct mathematical statement, but `3=+6$` is not. H_2O is a syntactically correct chemical name, 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=+6$` is that `$` is not a legal token in mathematics (at least as far as we know). Similarly, ${}_2Zz$ is not legal because there is no element with the abbreviation Zz.

The second type of syntax error pertains to the structure of a statement—that is, the way the tokens are arranged. The statement `3=+6$` is structurally illegal because you can't place a plus sign immediately after an equal sign. Similarly, molecular formulas have to have subscripts after the element name, not before.

> *As an exercise, create what appears to be a well-structured English sentence with unrecognizable tokens in it. Then write another sentence with all valid tokens but with invalid structure.*

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

For example, when you hear the sentence, "The other shoe fell," you understand that "the other shoe" is the subject and "fell" is the predicate. Once you have parsed a sentence, you can figure out what it means, or the semantics of the sentence. Assuming that you know what a shoe is and what it means to fall, you will understand the general implication of this sentence.

Although formal and natural languages have many features in common—tokens, structure, syntax, and semantics—there are many 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 other shoe fell," there is probably no shoe and nothing falling. Formal languages mean exactly what they say.

People who grow up speaking a natural language—everyone—often have a hard time adjusting to formal languages. In some ways, 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.

Here are some suggestions for reading programs (and other formal languages). First, remember that formal languages are much more dense than natural languages, so it takes longer to read them. Also, the structure is very important, so it is usually not a good idea 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. Little things like spelling errors and bad punctuation, which you can get away with in natural languages, can make a big difference in a formal language.

1.5 The first program

Traditionally, the first program written 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**, which doesn't actually print anything on paper. It displays a value 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 value; they don't appear in the result.

Some people judge the quality of a programming language by the simplicity of the "Hello, World!" program. By this standard, Python does about as well as possible.

1.6 Glossary

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

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 execute; also called "machine language" or "assembly language."

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

interpret: To execute a program in a high-level language by translating it one line at a time.

compile: To translate a program written in a high-level language into a low-level language all at once, in preparation for later execution.

source code: A program in a high-level language before being compiled.

object code: The output of the compiler after it translates the program.

executable: Another name for object code that is ready to be executed.

script: A program stored in a file (usually one that will be interpreted).

program: A set of instructions that specifies a computation.

algorithm: A general process for solving a category of problems.

bug: An error in a program.

debugging: The process of finding and removing any of the three kinds of programming errors.

syntax: The structure of a program.

syntax error: An error in a program that makes it impossible to parse (and therefore impossible to interpret).

runtime error: An error that does not occur until the program has started to execute but that prevents the program from continuing.

exception: Another name for a runtime error.

semantic error: An error in a program that makes it do something other than what the programmer intended.

semantics: The meaning of a program.

natural language: Any one of the languages that people speak that evolved naturally.

formal language: Any one of the languages that people have designed for specific purposes, such as representing mathematical ideas or computer programs; all programming languages are formal languages.

token: One of the basic elements of the syntactic structure of a program, analogous to a word in a natural language.

parse: To examine a program and analyze the syntactic structure.

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

Chapter 2

Variables, expressions and statements

2.1 Values and types

A **value** is one of the fundamental things—like a letter or a number—that a program manipulates. The values we have seen so far are `2` (the result when we added `1 + 1`), and `'Hello, World!'`.

These values belong to different **types**: `2` is an integer, and `'Hello, World!'` is a **string**, so-called because it contains a "string" of letters. You (and the interpreter) can identify strings because they are enclosed in quotation marks.

The print statement also works for integers.

```
>>> print 4
4
```

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

```
>>> type('Hello, World!')
<type 'str'>
>>> type(17)
<type 'int'>
```

Not surprisingly, strings belong to the type `str` and integers belong to the type `int`. Less obviously, numbers with a decimal point belong to a type called `float`, because these numbers are represented in a format called **floating-point**.

```
>>> type(3.2)
<type 'float'>
```

What about values like '17' and '3.2'? They look like numbers, but they are in quotation marks like strings.

```
>>> type('17')
<type 'str'>
>>> type('3.2')
<type 'str'>
```

They're strings.

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

```
>>> print 1,000,000
1 0 0
```

Well, that's not what we expected at all! Python interprets 1,000,000 as a comma-separated list of three integers, which it prints consecutively. This is the first example we have seen of a semantic error: the code runs without producing an error message, but it doesn't do the "right" thing.

2.2 Variables

One of the most powerful features of a programming language is the ability to manipulate **variables**. A variable is a name that refers to a value.

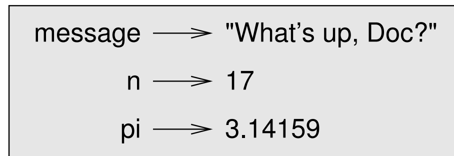
The **assignment statement** creates new variables and gives them values:

```
>>> message = "What's up, Doc?"
>>> n = 17
>>> pi = 3.14159
```

This example makes three assignments. The first assigns the string "What's up, Doc?" to a new variable named message. The second gives the integer 17 to n, and the third gives the floating-point number 3.14159 to pi.

Notice that the first statement uses double quotes to enclose the string. In general, single and double quotes do the same thing, but if the string contains a single quote (or an apostrophe, which is the same character), you have to use double quotes to enclose it.

A common way to represent variables on paper is to write the name with an arrow pointing to the variable's value. This kind of figure is called a **state diagram** because it shows what state each of the variables is in (think of it as the variable's state of mind). This diagram shows the result of the assignment statements:



The print statement also works with variables.

```
>>> print message
What's up, Doc?
>>> print n
17
>>> print pi
3.14159
```

In each case the result is the value of the variable. Variables also have types; again, we can ask the interpreter what they are.

```
>>> type(message)
<type 'str'>
>>> type(n)
<type 'int'>
>>> type(pi)
<type 'float'>
```

The type of a variable is the type of the value it refers to.

2.3 Variable names and keywords

Programmers generally choose names for their variables that are meaningful—they document what the variable is used for.

Variable names can be arbitrarily long. They can contain both letters and numbers, but they have to begin with a letter. Although it is legal to use uppercase letters, by convention we don't. If you do, remember that case matters. `Bruce` and `bruce` are different variables.

The underscore character (_) can appear in a name. It is often used in names with multiple words, such as `my_name` or `price_of_tea_in_china`.

If you give a variable an illegal name, you get a syntax error:

```
>>> 76trombones = 'big parade'
SyntaxError: invalid syntax
>>> more$ = 1000000
SyntaxError: invalid syntax
>>> class = 'Computer Science 101'
SyntaxError: invalid syntax
```

`76trombones` is illegal because it does not begin with a letter. `more$` is illegal because it contains an illegal character, the dollar sign. But what's wrong with `class`?

It turns out that `class` is one of the Python **keywords**. Keywords define the language's rules and structure, and they cannot be used as variable names.

Python has twenty-nine keywords:

```
and       def      exec     if       not      return
assert    del      finally  import   or       try
break     elif     for      in       pass     while
class     else     from     is       print    yield
continue  except   global   lambda   raise
```

You might want to keep this list handy. If the interpreter complains about one of your variable names and you don't know why, see if it is on this list.

2.4 Statements

A statement is an instruction that the Python interpreter can execute. We have seen two kinds of statements: print and assignment.

When you type a statement on the command line, Python executes it and displays the result, if there is one. The result of a print statement is a value. Assignment statements don't produce a result.

A script usually contains a sequence of statements. If there is more than one statement, the results appear one at a time as the statements execute.

For example, the script

```
print 1
x = 2
print x
```

produces the output

```
1
2
```

Again, the assignment statement produces no output.

2.5 Evaluating expressions

An expression is a combination of values, variables, and operators. If you type an expression on the command line, the interpreter **evaluates** it and displays the result:

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

Although expressions contain values, variables, and operators, not every expression contains all of these elements. A value all by itself is considered an expression, and so is a variable.

```
>>> 17
17
>>> x
2
```

Confusingly, evaluating an expression is not quite the same thing as printing a value.

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

When the Python interpreter displays the value of an expression, it uses the same format you would use to enter a value. In the case of strings, that means that it includes the quotation marks. But if you use a print statement, Python displays the contents of the string without the quotation marks.

In a script, an expression all by itself is a legal statement, but it doesn't do anything. The script

```
17
3.2
'Hello, World!'
1 + 1
```

produces no output at all. How would you change the script to display the values of these four expressions?

2.6 Operators and operands

Operators are special symbols that represent computations like addition and multiplication. The values the operator uses are called **operands**.

The following are all legal Python expressions whose meaning is more or less clear:

```
20+32   hour-1   hour*60+minute   minute/60   5**2   (5+9)*(15-7)
```

The symbols +, -, and /, and the use of parenthesis for grouping, mean in Python what they mean in mathematics. The asterisk (*) is the symbol for multiplication, and ** is the symbol for exponentiation.

When a variable name appears in the place of an operand, it is replaced with its value before the operation is performed.

Addition, subtraction, multiplication, and exponentiation all do what you expect, but you might be surprised by division. The following operation has an unexpected result:

```
>>> minute = 59
>>> minute/60
0
```

The value of `minute` is 59, and in conventional arithmetic 59 divided by 60 is 0.98333, not 0. The reason for the discrepancy is that Python is performing **integer division**.

When both of the operands are integers, the result must also be an integer, and by convention, integer division always rounds *down*, even in cases like this where the next integer is very close.

A possible solution to this problem is to calculate a percentage rather than a fraction:

```
>>> minute*100/60
98
```

Again the result is rounded down, but at least now the answer is approximately correct. Another alternative is to use floating-point division, which we get to in Chapter 3.

2.7 Order of operations

When more than one operator appears in an expression, the order of evaluation depends on the **rules of precedence**. Python follows the same precedence rules for its mathematical operators that mathematics does. The acronym **PEMDAS** is a useful way to remember the order of operations:

- **P**arentheses have the highest precedence and can be used to force an expression to evaluate in the order you want. Since expressions in parentheses are evaluated first, `2 * (3-1)` is 4, and `(1+1)**(5-2)` is 8. You can also use parentheses to make an expression easier to read, as in `(minute * 100) / 60`, even though it doesn't change the result.

- **E**xponentiation has the next highest precedence, so `2**1+1` is 3 and not 4, and `3*1**3` is 3 and not 27.

- **M**ultiplication and **D**ivision have the same precedence, which is higher than **A**ddition and **S**ubtraction, which also have the same precedence. So `2*3-1` yields 5 rather than 4, and `2/3-1` is `-1`, not `1` (remember that in integer division, `2/3=0`).

- Operators with the same precedence are evaluated from left to right. So in the expression `minute*100/60`, the multiplication happens first, yielding `5900/60`, which in turn yields `98`. If the operations had been evaluated from right to left, the result would have been `59*1`, which is `59`, which is wrong.

2.8 Operations on strings

In general, you cannot perform mathematical operations on strings, even if the strings look like numbers. The following are illegal (assuming that `message` has type `string`):

```
message-1   'Hello'/123   message*'Hello'   '15'+2
```

Interestingly, the `+` operator does work with strings, although it does not do exactly what you might expect. For strings, the `+` operator represents **concatenation**, which means joining the two operands by linking them end-to-end. For example:

```
fruit = 'banana'
bakedGood = ' nut bread'
print fruit + bakedGood
```

The output of this program is `banana nut bread`. The space before the word `nut` is part of the string, and is necessary to produce the space between the concatenated strings.

The `*` operator also works on strings; it performs repetition. For example, `'Fun'*3` is `'FunFunFun'`. One of the operands has to be a string; the other has to be an integer.

On one hand, this interpretation of + and * makes sense by analogy with addition and multiplication. Just as 4*3 is equivalent to 4+4+4, we expect 'Fun'*3 to be the same as 'Fun'+'Fun'+'Fun', and it is. On the other hand, there is a significant way in which string concatenation and repetition are different from integer addition and multiplication. Can you think of a property that addition and multiplication have that string concatenation and repetition do not?

2.9 Composition

So far, we have looked at the elements of a program—variables, expressions, and statements—in isolation, without talking about how to combine them.

One of the most useful features of programming languages is their ability to take small building blocks and **compose** them. For example, we know how to add numbers and we know how to print; it turns out we can do both at the same time:

```
>>>  print 17 + 3
20
```

In reality, the addition has to happen before the printing, so the actions aren't actually happening at the same time. The point is that any expression involving numbers, strings, and variables can be used inside a print statement. You've already seen an example of this:

```
print 'Number of minutes since midnight: ', hour*60+minute
```

You can also put arbitrary expressions on the right-hand side of an assignment statement:

```
percentage = (minute * 100) / 60
```

This ability may not seem impressive now, but you will see other examples where composition makes it possible to express complex computations neatly and concisely.

Warning: There are limits on where you can use certain expressions. For example, the left-hand side of an assignment statement has to be a *variable* name, not an expression. So, the following is illegal: minute+1 = hour.

2.10 Comments

As programs get bigger and more complicated, they get more difficult to read. Formal languages are dense, and it is often difficult to look at a piece of code and figure out what it is doing, or why.

For this reason, it is a good idea to add notes to your programs to explain in natural language what the program is doing. These notes are called **comments**, and they are marked with the # symbol:

```
# compute the percentage of the hour that has elapsed
percentage = (minute * 100) / 60
```

In this case, the comment appears on a line by itself. You can also put comments at the end of a line:

```
percentage = (minute * 100) / 60     # caution: integer division
```

Everything from the # to the end of the line is ignored—it has no effect on the program. The message is intended for the programmer or for future programmers who might use this code. In this case, it reminds the reader about the ever-surprising behavior of integer division.

This sort of comment is less necessary if you use the integer division operation, //. It has the same effect as the division operator[1], but it signals that the effect is deliberate.

```
percentage = (minute * 100) // 60
```

The integer division operator is like a comment that says, "I know this is integer division, and I like it that way!"

2.11 Glossary

value: A number or string (or other thing to be named later) that can be stored in a variable or computed in an expression.

type: A set of values. The type of a value determines how it can be used in expressions. So far, the types you have seen are integers (type `int`), floating-point numbers (type `float`), and strings (type `string`).

floating-point: A format for representing numbers with fractional parts.

variable: A name that refers to a value.

statement: A section of code that represents a command or action. So far, the statements you have seen are assignments and print statements.

assignment: A statement that assigns a value to a variable.

[1] For now. The behavior of the division operator may change in future versions of Python.

state diagram: A graphical representation of a set of variables and the values to which they refer.

keyword: A reserved word that is used by the compiler to parse a program; you cannot use keywords like `if`, `def`, and `while` as variable names.

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

operand: One of the values on which an operator operates.

expression: A combination of variables, operators, and values that represents a single result value.

evaluate: To simplify an expression by performing the operations in order to yield a single value.

integer division: An operation that divides one integer by another and yields an integer. Integer division yields only the whole number of times that the numerator is divisible by the denominator and discards any remainder.

rules of precedence: The set of rules governing the order in which expressions involving multiple operators and operands are evaluated.

concatenate: To join two operands end-to-end.

composition: The ability to combine simple expressions and statements into compound statements and expressions in order to represent complex computations concisely.

comment: Information in a program that is meant for other programmers (or anyone reading the source code) and has no effect on the execution of the program.

Chapter 3

Functions

3.1 Function calls

You have already seen one example of a **function call**:

```
>>> type("32")
<type 'str'>
```

The name of the function is `type`, and it displays the type of a value or variable. The value or variable, which is called the **argument** of the function, has to be enclosed in parentheses. It is common to say that a function "takes" an argument and "returns" a result. The result is called the **return value**.

Instead of printing the return value, we could assign it to a variable:

```
>>> betty = type("32")
>>> print betty
<type 'str'>
```

As another example, the `id` function takes a value or a variable and returns an integer that acts as a unique identifier for the value:

```
>>> id(3)
134882108
>>> betty = 3
>>> id(betty)
134882108
```

Every value has an `id`, which is a unique number related to where it is stored in the memory of the computer. The `id` of a variable is the `id` of the value to which it refers.

3.2 Type conversion

Python provides a collection of built-in functions that convert values from one type to another. The `int` function takes any value and converts it to an integer, if possible, or complains otherwise:

```
>>> int("32")
32
>>> int("Hello")
ValueError: invalid literal for int(): Hello
```

`int` can also convert floating-point values to integers, but remember that it truncates the fractional part:

```
>>> int(3.99999)
3
>>> int(-2.3)
-2
```

The `float` function converts integers and strings to floating-point numbers:

```
>>> float(32)
32.0
>>> float("3.14159")
3.14159
```

Finally, the `str` function converts to type `string`:

```
>>> str(32)
'32'
>>> str(3.14149)
'3.14149'
```

It may seem odd that Python distinguishes the integer value `1` from the floating-point value `1.0`. They may represent the same number, but they belong to different types. The reason is that they are represented differently inside the computer.

3.3 Type coercion

Now that we can convert between types, we have another way to deal with integer division. Returning to the example from the previous chapter, suppose we want to calculate the fraction of an hour that has elapsed. The most obvious expression, `minute / 60`, does integer arithmetic, so the result is always 0, even at 59 minutes past the hour.

One solution is to convert `minute` to floating-point and do floating-point division:

```
>>> minute = 59
>>> float(minute) / 60
0.983333333333
```

Alternatively, we can take advantage of the rules for automatic type conversion, which is called **type coercion**. For the mathematical operators, if either operand is a `float`, the other is automatically converted to a `float`:

```
>>> minute = 59
>>> minute / 60.0
0.983333333333
```

By making the denominator a `float`, we force Python to do floating-point division.

3.4 Math functions

In mathematics, you have probably seen functions like `sin` and `log`, and you have learned to evaluate expressions like `sin(pi/2)` and `log(1/x)`. First, you evaluate the expression in parentheses (the argument). For example, `pi/2` is approximately 1.571, and `1/x` is 0.1 (if `x` happens to be 10.0).

Then, you evaluate the function itself, either by looking it up in a table or by performing various computations. The `sin` of 1.571 is 1, and the `log` of 0.1 is -1 (assuming that `log` indicates the logarithm base 10).

This process can be applied repeatedly to evaluate more complicated expressions like `log(1/sin(pi/2))`. First, you evaluate the argument of the innermost function, then evaluate the function, and so on.

Python has a math module that provides most of the familiar mathematical functions. A **module** is a file that contains a collection of related functions grouped together.

Before we can use the functions from a module, we have to import them:

```
>>> import math
```

To call one of the functions, we have to specify the name of the module and the name of the function, separated by a dot, also known as a period. This format is called **dot notation**.

```
>>> decibel = math.log10 (17.0)
>>> angle = 1.5
>>> height = math.sin(angle)
```

The first statement sets `decibel` to the logarithm of 17, base `10`. There is also a function called `log` that takes logarithm base `e`.

The third statement finds the sine of the value of the variable `angle`. `sin` and the other trigonometric functions (`cos`, `tan`, etc.) take arguments in radians. To convert from degrees to radians, divide by 360 and multiply by `2*pi`. For example, to find the sine of 45 degrees, first calculate the angle in radians and then take the sine:

```
>>> degrees = 45
>>> angle = degrees * 2 * math.pi / 360.0
>>> math.sin(angle)
0.707106781187
```

The constant `pi` is also part of the math module. If you know your geometry, you can check the previous result by comparing it to the square root of two divided by two:

```
>>> math.sqrt(2) / 2.0
0.707106781187
```

3.5 Composition

Just as with mathematical functions, Python functions can be composed, meaning that you use one expression as part of another. For example, you can use any expression as an argument to a function:

```
>>> x = math.cos(angle + math.pi/2)
```

This statement takes the value of `pi`, divides it by 2, and adds the result to the value of `angle`. The sum is then passed as an argument to the `cos` function.

You can also take the result of one function and pass it as an argument to another:

```
>>> x = math.exp(math.log(10.0))
```

This statement finds the log base `e` of 10 and then raises `e` to that power. The result gets assigned to `x`.

3.6 Adding new functions

So far, we have only been using the functions that come with Python, but it is also possible to add new functions. Creating new functions to solve your particular

problems is one of the most useful things about a general-purpose programming language.

In the context of programming, a **function** is a named sequence of statements that performs a desired operation. This operation is specified in a **function definition**. The functions we have been using so far have been defined for us, and these definitions have been hidden. This is a good thing, because it allows us to use the functions without worrying about the details of their definitions.

The syntax for a function definition is:

```
def NAME( LIST OF PARAMETERS ):
  STATEMENTS
```

You can make up any names you want for the functions you create, except that you can't use a name that is a Python keyword. The list of parameters specifies what information, if any, you have to provide in order to use the new function.

There can be any number of statements inside the function, but they have to be indented from the left margin. In the examples in this book, we will use an indentation of two spaces.

The first couple of functions we are going to write have no parameters, so the syntax looks like this:

```
def newLine():
  print
```

This function is named `newLine`. The empty parentheses indicate that it has no parameters. It contains only a single statement, which outputs a newline character. (That's what happens when you use a `print` command without any arguments.)

The syntax for calling the new function is the same as the syntax for built-in functions:

```
print "First Line."
newLine()
print "Second Line."
```

The output of this program is:

```
First line.

Second line.
```

Notice the extra space between the two lines. What if we wanted more space between the lines? We could call the same function repeatedly:

```
print "First Line."
newLine()
newLine()
newLine()
print "Second Line."
```

Or we could write a new function named `threeLines` that prints three new lines:

```
def threeLines():
  newLine()
  newLine()
  newLine()

print "First Line."
threeLines()
print "Second Line."
```

This function contains three statements, all of which are indented by two spaces. Since the next statement is not indented, Python knows that it is not part of the function.

You should notice a few things about this program:

1. You can call the same procedure repeatedly. In fact, it is quite common and useful to do so.

2. You can have one function call another function; in this case `threeLines` calls `newLine`.

So far, it may not be clear why it is worth the trouble to create all of these new functions. Actually, there are a lot of reasons, but this example demonstrates two:

- Creating a new function gives you an opportunity to name a group of statements. Functions can simplify a program by hiding a complex computation behind a single command and by using English words in place of arcane code.

- Creating a new function can make a program smaller by eliminating repetitive code. For example, a short way to print nine consecutive new lines is to call `threeLines` three times.

  As an exercise, write a function called `nineLines` *that uses* `threeLines` *to print nine blank lines. How would you print twenty-seven new lines?*

3.7 Definitions and use

Pulling together the code fragments from Section 3.6, the whole program looks like this:

```
def newLine():
  print

def threeLines():
  newLine()
  newLine()
  newLine()

print "First Line."
threeLines()
print "Second Line."
```

This program contains two function definitions: `newLine` and `threeLines`. Function definitions get executed just like other statements, but the effect is to create the new function. The statements inside the function do not get executed until the function is called, and the function definition generates no output.

As you might expect, you have to create a function before you can execute it. In other words, the function definition has to be executed before the first time it is called.

> *As an exercise, move the last three lines of this program to the top, so the function calls appear before the definitions. Run the program and see what error message you get.*
>
> *As another exercise, start with the working version of the program and move the definition of* `newLine` *after the definition of* `threeLines`*. What happens when you run this program?*

3.8 Flow of execution

In order to ensure that a function is defined before its first use, you have to know the order in which statements are executed, which is called the **flow of execution**.

Execution always begins at the first statement of the program. Statements are executed one at a time, in order from top to bottom.

Function definitions do not alter the flow of execution of the program, but remember that statements inside the function are not executed until the function

is called. Although it is not common, you can define one function inside another. In this case, the inner definition isn't executed until the outer function is called.

Function calls are like a detour in the flow of execution. Instead of going to the next statement, the flow jumps to the first line of the called function, executes all the statements there, and then comes back to pick up where it left off.

That sounds simple enough, until you remember that one function can call another. While in the middle of one function, the program might have to execute the statements in another function. But while executing that new function, the program might have to execute yet another function!

Fortunately, Python is adept at keeping track of where it is, so each time a function completes, the program picks up where it left off in the function that called it. When it gets to the end of the program, it terminates.

What's the moral of this sordid tale? When you read a program, don't read from top to bottom. Instead, follow the flow of execution.

3.9 Parameters and arguments

Some of the built-in functions you have used require arguments, the values that control how the function does its job. For example, if you want to find the sine of a number, you have to indicate what the number is. Thus, `sin` takes a numeric value as an argument.

Some functions take more than one argument. For example, `pow` takes two arguments, the base and the exponent. Inside the function, the values that are passed get assigned to variables called **parameters**.

Here is an example of a user-defined function that has a parameter:

```
def printTwice(bruce):
  print bruce, bruce
```

This function takes a single argument and assigns it to a parameter named `bruce`. The value of the parameter (at this point we have no idea what it will be) is printed twice, followed by a newline. The name `bruce` was chosen to suggest that the name you give a parameter is up to you, but in general, you want to choose something more illustrative than `bruce`.

The function `printTwice` works for any type that can be printed:

```
>>> printTwice('Spam')
Spam Spam
```

```
>>> printTwice(5)
5 5
>>> printTwice(3.14159)
3.14159 3.14159
```

In the first function call, the argument is a string. In the second, it's an integer. In the third, it's a `float`.

The same rules of composition that apply to built-in functions also apply to user-defined functions, so we can use any kind of expression as an argument for `printTwice`:

```
>>> printTwice('Spam'*4)
SpamSpamSpamSpam SpamSpamSpamSpam
>>> printTwice(math.cos(math.pi))
-1.0 -1.0
```

As usual, the expression is evaluated before the function is run, so `printTwice` prints `SpamSpamSpamSpam SpamSpamSpamSpam` instead of `'Spam'*4 'Spam'*4`.

> *As an exercise, write a call to* `printTwice` *that does print* `'Spam'*4 'Spam'*4`*. Hint: strings can be enclosed in either single or double quotes, and the type of quote not used to enclose the string can be used inside it as part of the string.*

We can also use a variable as an argument:

```
>>> michael = 'Eric, the half a bee.'
>>> printTwice(michael)
Eric, the half a bee. Eric, the half a bee.
```

Notice something very important here. The name of the variable we pass as an argument (`michael`) has nothing to do with the name of the parameter (`bruce`). It doesn't matter what the value was called back home (in the caller); here in `printTwice`, we call everybody `bruce`.

3.10 Variables and parameters are local

When you create a **local variable** inside a function, it only exists inside the function, and you cannot use it outside. For example:

```
def catTwice(part1, part2):
  cat = part1 + part2
  printTwice(cat)
```

This function takes two arguments, concatenates them, and then prints the result twice. We can call the function with two strings:

```
>>> chant1 = "Pie Jesu domine, "
>>> chant2 = "Dona eis requiem."
>>> catTwice(chant1, chant2)
Pie Jesu domine, Dona eis requiem. Pie Jesu domine, Dona eis requiem.
```

When `catTwice` terminates, the variable `cat` is destroyed. If we try to print it, we get an error:

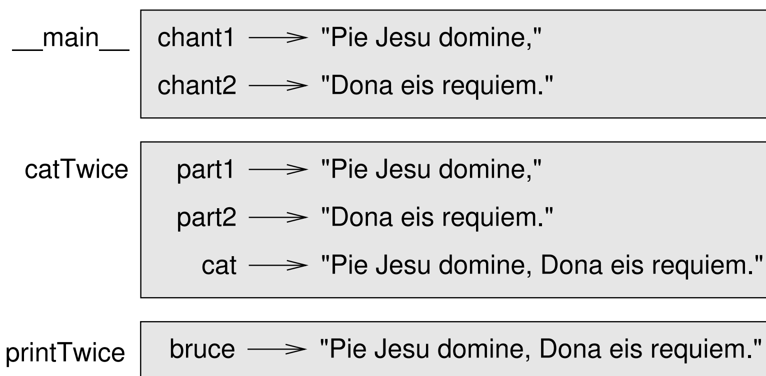
```
>>> print cat
NameError: cat
```

Parameters are also local. For example, outside the function `printTwice`, there is no such thing as `bruce`. If you try to use it, Python will complain.

3.11 Stack diagrams

To keep track of which variables can be used where, it is sometimes useful to draw a **stack diagram**. Like state diagrams, stack diagrams show the value of each variable, but they also show the function to which each variable belongs.

Each function is represented by a **frame**. A frame is a box with the name of a function beside it and the parameters and variables of the function inside it. The stack diagram for the previous example looks like this:



The order of the stack shows the flow of execution. `printTwice` was called by `catTwice`, and `catTwice` was called by `__main__`, which is a special name for the topmost function. When you create a variable outside of any function, it belongs to `__main__`.

Each parameter refers to the same value as its corresponding argument. So, `part1` has the same value as `chant1`, `part2` has the same value as `chant2`, and `bruce` has the same value as `cat`.

If an error occurs during a function call, Python prints the name of the function, and the name of the function that called it, and the name of the function that called *that*, all the way back to `__main__`.

For example, if we try to access `cat` from within `printTwice`, we get a `NameError`:

```
Traceback (innermost last):
  File "test.py", line 13, in __main__
    catTwice(chant1, chant2)
  File "test.py", line 5, in catTwice
    printTwice(cat)
  File "test.py", line 9, in printTwice
    print cat
NameError: cat
```

This list of functions is called a **traceback**. It tells you what program file the error occurred in, and what line, and what functions were executing at the time. It also shows the line of code that caused the error.

Notice the similarity between the traceback and the stack diagram. It's not a coincidence.

3.12 Functions with results

You might have noticed by now that some of the functions we are using, such as the math functions, yield results. Other functions, like `newLine`, perform an action but don't return a value. That raises some questions:

1. What happens if you call a function and you don't do anything with the result (i.e., you don't assign it to a variable or use it as part of a larger expression)?

2. What happens if you use a function without a result as part of an expression, such as `newLine() + 7`?

3. Can you write functions that yield results, or are you stuck with simple function like `newLine` and `printTwice`?

The answer to the last question is that you can write functions that yield results, and we'll do it in Chapter 5.

> *As an exercise, answer the other two questions by trying them out. When you have a question about what is legal or illegal in Python, a good way to find out is to ask the interpreter.*

3.13 Glossary

function call: A statement that executes a function. It consists of the name of the function followed by a list of arguments enclosed in parentheses.

argument: A value provided to a function when the function is called. This value is assigned to the corresponding parameter in the function.

return value: The result of a function. If a function call is used as an expression, the return value is the value of the expression.

type conversion: An explicit statement that takes a value of one type and computes a corresponding value of another type.

type coercion: A type conversion that happens automatically according to Python's coercion rules.

module: A file that contains a collection of related functions and classes.

dot notation: The syntax for calling a function in another module, specifying the module name followed by a dot (period) and the function name.

function: A named sequence of statements that performs some useful operation. Functions may or may not take arguments and may or may not produce a result.

function definition: A statement that creates a new function, specifying its name, parameters, and the statements it executes.

flow of execution: The order in which statements are executed during a program run.

parameter: A name used inside a function to refer to the value passed as an argument.

local variable: A variable defined inside a function. A local variable can only be used inside its function.

stack diagram: A graphical representation of a stack of functions, their variables, and the values to which they refer.

frame: A box in a stack diagram that represents a function call. It contains the local variables and parameters of the function.

traceback: A list of the functions that are executing, printed when a runtime error occurs.

Chapter 4

Conditionals and recursion

4.1 The modulus operator

The **modulus operator** works on integers (and integer expressions) and yields the remainder when the first operand is divided by the second. In Python, the modulus operator is a percent sign (%). The syntax is the same as for other operators:

```
>>> quotient = 7 / 3
>>> print quotient
2
>>> remainder = 7 % 3
>>> print remainder
1
```

So 7 divided by 3 is 2 with 1 left over.

The modulus operator turns out to be surprisingly useful. For example, you can check whether one number is divisible by another—if `x % y` is zero, then `x` is divisible by `y`.

Also, you can extract the right-most digit or digits from a number. For example, `x % 10` yields the right-most digit of `x` (in base 10). Similarly `x % 100` yields the last two digits.

4.2 Boolean expressions

A **boolean expression** is an expression that is either true or false. One way to write a boolean expression is to use the operator `==`, which compares two values and produces a boolean value:

```
>>> 5 == 5
True
>>> 5 == 6
False
```

In the first statement, the two operands are equal, so the value of the expression is `True`; in the second statement, 5 is not equal to 6, so we get `False`. `True` and `False` are special values that are built into Python.

The `==` operator is one of the **comparison operators**; the others are:

```
x != y               # x is not equal to y
x > y                # x is greater than y
x < y                # x is less than y
x >= y               # x is greater than or equal to y
x <= y               # x is less than or equal to y
```

Although these operations are probably familiar to you, the Python symbols are different from the mathematical symbols. A common error is to use a single equal sign (`=`) instead of a double equal sign (`==`). Remember that `=` is an assignment operator and `==` is a comparison operator. Also, there is no such thing as `=<` or `=>`.

4.3 Logical operators

There are three **logical operators**: `and`, `or`, and `not`. The semantics (meaning) of these operators is similar to their meaning in English. For example, `x > 0 and x < 10` is true only if `x` is greater than 0 *and* less than 10.

`n%2 == 0 or n%3 == 0` is true if *either* of the conditions is true, that is, if the number is divisible by 2 *or* 3.

Finally, the `not` operator negates a boolean expression, so `not(x > y)` is true if `(x > y)` is false, that is, if `x` is less than or equal to `y`.

Strictly speaking, the operands of the logical operators should be boolean expressions, but Python is not very strict. Any nonzero number is interpreted as "true."

```
>>>  x = 5
>>>  x and 1
1
>>>  y = 0
>>>  y and 1
0
```

In general, this sort of thing is not considered good style. If you want to compare a value to zero, you should do it explicitly.

4.4 Conditional execution

In order to write useful programs, we almost always need the ability to check conditions and change the behavior of the program accordingly. **Conditional statements** give us this ability. The simplest form is the `if` statement:

```
if x > 0:
  print "x is positive"
```

The boolean expression after the `if` statement is called the **condition**. If it is true, then the indented statement gets executed. If not, nothing happens.

Like other compound statements, the `if` statement is made up of a header and a block of statements:

```
HEADER:
  FIRST STATEMENT
  ...
  LAST STATEMENT
```

The header begins on a new line and ends with a colon (:). The indented statements that follow are called a **block**. The first unindented statement marks the end of the block. A statement block inside a compound statement is called the **body** of the statement.

There is no limit on the number of statements that can appear in the body of an if statement, but there has to be at least one. Occasionally, it is useful to have a body with no statements (usually as a place keeper for code you haven't written yet). In that case, you can use the `pass` statement, which does nothing.

4.5 Alternative execution

A second form of the `if` statement is alternative execution, in which there are two possibilities and the condition determines which one gets executed. The syntax looks like this:

```
if x%2 == 0:
  print x, "is even"
else:
  print x, "is odd"
```

If the remainder when x is divided by 2 is 0, then we know that x is even, and the program displays a message to that effect. If the condition is false, the second set of statements is executed. Since the condition must be true or false, exactly one of the alternatives will be executed. The alternatives are called **branches**, because they are branches in the flow of execution.

As an aside, if you need to check the parity (evenness or oddness) of numbers often, you might "wrap" this code in a function:

```
def printParity(x):
  if x%2 == 0:
    print x, "is even"
  else:
    print x, "is odd"
```

For any value of x, printParity displays an appropriate message. When you call it, you can provide any integer expression as an argument.

```
>>> printParity(17)
17 is odd
>>> y = 17
>>> printParity(y+1)
18 is even
```

4.6 Chained conditionals

Sometimes there are more than two possibilities and we need more than two branches. One way to express a computation like that is a **chained conditional**:

```
if x < y:
  print x, "is less than", y
elif x > y:
  print x, "is greater than", y
else:
  print x, "and", y, "are equal"
```

elif is an abbreviation of "else if." Again, exactly one branch will be executed. There is no limit of the number of elif statements, but the last branch has to be an else statement:

```
if choice == 'A':
  functionA()
elif choice == 'B':
  functionB()
elif choice == 'C':
  functionC()
else:
  print "Invalid choice."
```

Each condition is checked in order. If the first is false, the next is checked, and so on. If one of them is true, the corresponding branch executes, and the statement ends. Even if more than one condition is true, only the first true branch executes.

> *As an exercise, wrap these examples in functions called* `compare(x, y)` *and* `dispatch(choice)`*.*

4.7 Nested conditionals

One conditional can also be nested within another. We could have written the trichotomy example as follows:

```
if x == y:
  print x, "and", y, "are equal"
else:
  if x < y:
    print x, "is less than", y
  else:
    print x, "is greater than", y
```

The outer conditional contains two branches. The first branch contains a simple output statement. The second branch contains another `if` statement, which has two branches of its own. Those two branches are both output statements, although they could have been conditional statements as well.

Although the indentation of the statements makes the structure apparent, nested conditionals become difficult to read very quickly. In general, it is a good idea to avoid them when you can.

Logical operators often provide a way to simplify nested conditional statements. For example, we can rewrite the following code using a single conditional:

```
if 0 < x:
  if x < 10:
    print "x is a positive single digit."
```

The `print` statement is executed only if we make it past both the conditionals, so we can use the `and` operator:

```
if 0 < x and x < 10:
  print "x is a positive single digit."
```

These kinds of conditions are common, so Python provides an alternative syntax that is similar to mathematical notation:

```
if 0 < x < 10:
  print "x is a positive single digit."
```

This condition is semantically the same as the compound boolean expression and the nested conditional.

4.8 The `return` statement

The `return` statement allows you to terminate the execution of a function before you reach the end. One reason to use it is if you detect an error condition:

```
import math

def printLogarithm(x):
  if x <= 0:
    print "Positive numbers only, please."
    return

  result = math.log(x)
  print "The log of x is", result
```

The function `printLogarithm` has a parameter named `x`. The first thing it does is check whether `x` is less than or equal to 0, in which case it displays an error message and then uses `return` to exit the function. The flow of execution immediately returns to the caller, and the remaining lines of the function are not executed.

Remember that to use a function from the math module, you have to import it.

4.9 Recursion

We mentioned that it is legal for one function to call another, and you have seen several examples of that. We neglected to mention that it is also legal for a function to call itself. It may not be obvious why that is a good thing, but it turns out to be one of the most magical and interesting things a program can do. For example, look at the following function:

```
def countdown(n):
  if n == 0:
    print "Blastoff!"
  else:
    print n
    countdown(n-1)
```

`countdown` expects the parameter, `n`, to be a positive integer. If `n` is 0, it outputs the word, "Blastoff!" Otherwise, it outputs `n` and then calls a function named `countdown`—itself—passing `n-1` as an argument.

What happens if we call this function like this:

```
>>> countdown(3)
```

The execution of `countdown` begins with `n=3`, and since `n` is not 0, it outputs the value 3, and then calls itself...

> The execution of `countdown` begins with `n=2`, and since `n` is not 0, it outputs the value 2, and then calls itself...
>
> > The execution of `countdown` begins with `n=1`, and since `n` is not 0, it outputs the value 1, and then calls itself...
> >
> > > The execution of `countdown` begins with `n=0`, and since `n` is 0, it outputs the word, "Blastoff!" and then returns.
> >
> > The `countdown` that got `n=1` returns.
>
> The `countdown` that got `n=2` returns.

The `countdown` that got `n=3` returns.

And then you're back in `__main__` (what a trip). So, the total output looks like this:

```
3
2
1
Blastoff!
```

As a second example, look again at the functions `newLine` and `threeLines`:

```
def newline():
  print

def threeLines():
  newLine()
  newLine()
  newLine()
```

Although these work, they would not be much help if we wanted to output 2 newlines, or 106. A better alternative would be this:

```
def nLines(n):
  if n > 0:
    print
    nLines(n-1)
```

This program is similar to `countdown`; as long as `n` is greater than 0, it outputs one newline and then calls itself to output `n-1` additional newlines. Thus, the total number of newlines is `1 + (n - 1)` which, if you do your algebra right, comes out to `n`.

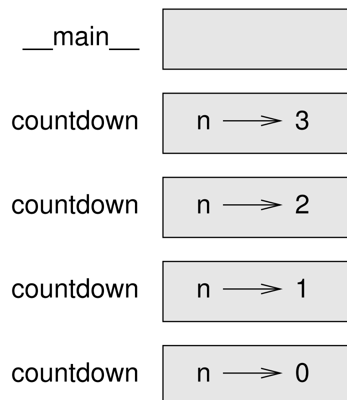
The process of a function calling itself is **recursion**, and such functions are said to be recursive.

4.10 Stack diagrams for recursive functions

In Section 3.11, we used a stack diagram to represent the state of a program during a function call. The same kind of diagram can help interpret a recursive function.

Every time a function gets called, Python creates a new function frame, which contains the function's local variables and parameters. For a recursive function, there might be more than one frame on the stack at the same time.

This figure shows a stack diagram for `countdown` called with `n = 3`:



As usual, the top of the stack is the frame for `__main__`. It is empty because we did not create any variables in `__main__` or pass any arguments to it.

The four `countdown` frames have different values for the parameter `n`. The bottom of the stack, where `n=0`, is called the **base case**. It does not make a recursive call, so there are no more frames.

As an exercise, draw a stack diagram for `nLines` *called with* `n=4`*.*

4.11 Infinite recursion

If a recursion never reaches a base case, it goes on making recursive calls forever, and the program never terminates. This is known as **infinite recursion**, and it is generally not considered a good idea. Here is a minimal program with an infinite recursion:

```
def recurse():
  recurse()
```

In most programming environments, a program with infinite recursion does not really run forever. Python reports an error message when the maximum recursion depth is reached:

```
  File "<stdin>", line 2, in recurse
  (98 repetitions omitted)
  File "<stdin>", line 2, in recurse
RuntimeError: Maximum recursion depth exceeded
```

This traceback is a little bigger than the one we saw in the previous chapter. When the error occurs, there are 100 `recurse` frames on the stack!

> *As an exercise, write a function with infinite recursion and run it in the Python interpreter.*

4.12 Keyboard input

The programs we have written so far are a bit rude in the sense that they accept no input from the user. They just do the same thing every time.

Python provides built-in functions that get input from the keyboard. The simplest is called `raw_input`. When this function is called, the program stops and waits for the user to type something. When the user presses Return or the Enter key, the program resumes and `raw_input` returns what the user typed as a **string**:

```
>>> input = raw_input ()
What are you waiting for?
>>> print input
What are you waiting for?
```

Before calling `raw_input`, it is a good idea to print a message telling the user what to input. This message is called a **prompt**. We can supply a prompt as an argument to `raw_input`:

```
>>> name = raw_input ("What...is your name? ")
What...is your name? Arthur, King of the Britons!
>>> print name
Arthur, King of the Britons!
```

If we expect the response to be an integer, we can use the `input` function:

```
prompt = "What...is the airspeed velocity of an unladen swallow?\n"
speed = input(prompt)
```

The sequence `\n` at the end of the string represents a newline, so the user's input appears below the prompt.

If the user types a string of digits, it is converted to an integer and assigned to `speed`. Unfortunately, if the user types a character that is not a digit, the program crashes:

```
>>> speed = input (prompt)
What...is the airspeed velocity of an unladen swallow?
What do you mean, an African or a European swallow?
SyntaxError: invalid syntax
```

To avoid this kind of error, it is generally a good idea to use `raw_input` to get a string and then use conversion functions to convert to other types.

4.13 Glossary

modulus operator: An operator, denoted with a percent sign (`%`), that works on integers and yields the remainder when one number is divided by another.

boolean expression: An expression that is either true or false.

comparison operator: One of the operators that compares two values: `==`, `!=`, `>`, `<`, `>=`, and `<=`.

logical operator: One of the operators that combines boolean expressions: `and`, `or`, and `not`.

conditional statement: A statement that controls the flow of execution depending on some condition.

condition: The boolean expression in a conditional statement that determines which branch is executed.

compound statement: A statement that consists of a header and a body. The header ends with a colon (:). The body is indented relative to the header.

block: A group of consecutive statements with the same indentation.

body: The block in a compound statement that follows the header.

nesting: One program structure within another, such as a conditional statement inside a branch of another conditional statement.

recursion: The process of calling the function that is currently executing.

base case: A branch of the conditional statement in a recursive function that does not result in a recursive call.

infinite recursion: A function that calls itself recursively without ever reaching the base case. Eventually, an infinite recursion causes a runtime error.

prompt: A visual cue that tells the user to input data.

Chapter 5

Fruitful functions

5.1 Return values

Some of the built-in functions we have used, such as the math functions, have produced results. Calling the function generates a new value, which we usually assign to a variable or use as part of an expression.

```
e = math.exp(1.0)
height = radius * math.sin(angle)
```

But so far, none of the functions we have written has returned a value.

In this chapter, we are going to write functions that return values, which we will call **fruitful functions**, for want of a better name. The first example is `area`, which returns the area of a circle with the given radius:

```
import math

def area(radius):
  temp = math.pi * radius**2
  return temp
```

We have seen the `return` statement before, but in a fruitful function the `return` statement includes a **return value**. This statement means: "Return immediately from this function and use the following expression as a return value." The expression provided can be arbitrarily complicated, so we could have written this function more concisely:

```
def area(radius):
  return math.pi * radius**2
```

On the other hand, **temporary variables** like `temp` often make debugging easier.

Sometimes it is useful to have multiple return statements, one in each branch of a conditional:

```
def absoluteValue(x):
  if x < 0:
    return -x
  else:
    return x
```

Since these `return` statements are in an alternative conditional, only one will be executed. As soon as one is executed, the function terminates without executing any subsequent statements.

Code that appears after a `return` statement, or any other place the flow of execution can never reach, is called **dead code**.

In a fruitful function, it is a good idea to ensure that every possible path through the program hits a `return` statement. For example:

```
def absoluteValue(x):
  if x < 0:
    return -x
  elif x > 0:
    return x
```

This program is not correct because if `x` happens to be 0, neither condition is true, and the function ends without hitting a `return` statement. In this case, the return value is a special value called `None`:

```
>>> print absoluteValue(0)
None
```

> *As an exercise, write a* `compare` *function that returns* `1` *if* `x > y`*,* `0` *if* `x == y`*, and* `-1` *if* `x < y`*.*

5.2 Program development

At this point, you should be able to look at complete functions and tell what they do. Also, if you have been doing the exercises, you have written some small functions. As you write larger functions, you might start to have more difficulty, especially with runtime and semantic errors.

To deal with increasingly complex programs, we are going to suggest a technique called **incremental development**. The goal of incremental development is to

avoid long debugging sessions by adding and testing only a small amount of code at a time.

As an example, suppose you want to find the distance between two points, given by the coordinates (x_1, y_1) and (x_2, y_2). By the Pythagorean theorem, the distance is:

$$\text{distance} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

The first step is to consider what a `distance` function should look like in Python. In other words, what are the inputs (parameters) and what is the output (return value)?

In this case, the two points are the inputs, which we can represent using four parameters. The return value is the distance, which is a floating-point value.

Already we can write an outline of the function:

```
def distance(x1, y1, x2, y2):
  return 0.0
```

Obviously, this version of the function doesn't compute distances; it always returns zero. But it is syntactically correct, and it will run, which means that we can test it before we make it more complicated.

To test the new function, we call it with sample values:

```
>>> distance(1, 2, 4, 6)
0.0
```

We chose these values so that the horizontal distance equals 3 and the vertical distance equals 4; that way, the result is 5 (the hypotenuse of a 3-4-5 triangle). When testing a function, it is useful to know the right answer.

At this point we have confirmed that the function is syntactically correct, and we can start adding lines of code. After each incremental change, we test the function again. If an error occurs at any point, we know where it must be—in the last line we added.

A logical first step in the computation is to find the differences $x_2 - x_1$ and $y_2 - y_1$. We will store those values in temporary variables named `dx` and `dy` and print them.

```
def distance(x1, y1, x2, y2):
  dx = x2 - x1
  dy = y2 - y1
  print "dx is", dx
  print "dy is", dy
  return 0.0
```

If the function is working, the outputs should be 3 and 4. If so, we know that the function is getting the right arguments and performing the first computation correctly. If not, there are only a few lines to check.

Next we compute the sum of squares of `dx` and `dy`:

```
def distance(x1, y1, x2, y2):
  dx = x2 - x1
  dy = y2 - y1
  dsquared = dx**2 + dy**2
  print "dsquared is: ", dsquared
  return 0.0
```

Notice that we removed the `print` statements we wrote in the previous step. Code like that is called **scaffolding** because it is helpful for building the program but is not part of the final product.

Again, we would run the program at this stage and check the output (which should be 25).

Finally, if we have imported the math module, we can use the `sqrt` function to compute and return the result:

```
def distance(x1, y1, x2, y2):
  dx = x2 - x1
  dy = y2 - y1
  dsquared = dx**2 + dy**2
  result = math.sqrt(dsquared)
  return result
```

If that works correctly, you are done. Otherwise, you might want to print the value of `result` before the return statement.

When you start out, you should add only a line or two of code at a time. As you gain more experience, you might find yourself writing and debugging bigger chunks. Either way, the incremental development process can save you a lot of debugging time.

The key aspects of the process are:

1. Start with a working program and make small incremental changes. At any point, if there is an error, you will know exactly where it is.

2. Use temporary variables to hold intermediate values so you can output and check them.

3. Once the program is working, you might want to remove some of the scaffolding or consolidate multiple statements into compound expressions, but only if it does not make the program difficult to read.

   As an exercise, use incremental development to write a function called `hypotenuse` *that returns the length of the hypotenuse of a right triangle given the lengths of the two legs as arguments. Record each stage of the incremental development process as you go.*

5.3 Composition

As you should expect by now, you can call one function from within another. This ability is called **composition**.

As an example, we'll write a function that takes two points, the center of the circle and a point on the perimeter, and computes the area of the circle.

Assume that the center point is stored in the variables `xc` and `yc`, and the perimeter point is in `xp` and `yp`. The first step is to find the radius of the circle, which is the distance between the two points. Fortunately, there is a function, `distance`, that does that:

```
radius = distance(xc, yc, xp, yp)
```

The second step is to find the area of a circle with that radius and return it:

```
result = area(radius)
return result
```

Wrapping that up in a function, we get:

```
def area2(xc, yc, xp, yp):
  radius = distance(xc, yc, xp, yp)
  result = area(radius)
  return result
```

We called this function `area2` to distinguish it from the `area` function defined earlier. There can only be one function with a given name within a given module.

The temporary variables `radius` and `result` are useful for development and debugging, but once the program is working, we can make it more concise by composing the function calls:

```
def area2(xc, yc, xp, yp):
  return area(distance(xc, yc, xp, yp))
```

As an exercise, write a function `slope(x1, y1, x2, y2)` *that returns the slope of the line through the points* $(x1, y1)$ *and* $(x2, y2)$*. Then use this function in a function called* `intercept(x1, y1, x2, y2)` *that returns the y-intercept of the line through the points* `(x1, y1)` *and* `(x2, y2)`*.*

5.4 Boolean functions

Functions can return boolean values, which is often convenient for hiding complicated tests inside functions. For example:

```
def isDivisible(x, y):
  if x % y == 0:
    return True
  else:
    return False
```

The name of this function is `isDivisible`. It is common to give boolean functions names that sound like yes/no questions. `isDivisible` returns either `True` or `False` to indicate whether the `x` is or is not divisible by `y`.

We can make the function more concise by taking advantage of the fact that the condition of the `if` statement is itself a boolean expression. We can return it directly, avoiding the `if` statement altogether:

```
def isDivisible(x, y):
  return x % y == 0
```

This session shows the new function in action:

```
>>>   isDivisible(6, 4)
False
>>>   isDivisible(6, 3)
True
```

Boolean functions are often used in conditional statements:

```
if isDivisible(x, y):
  print "x is divisible by y"
else:
  print "x is not divisible by y"
```

It might be tempting to write something like:

```
if isDivisible(x, y) == True:
```

But the extra comparison is unnecessary.

> *As an exercise, write a function* `isBetween(x, y, z)` *that returns* `True` *if* $y \leq x \leq z$ *or* `False` *otherwise.*

5.5 More recursion

So far, you have only learned a small subset of Python, but you might be interested to know that this subset is a *complete* programming language, which means that anything that can be computed can be expressed in this language. Any program ever written could be rewritten using only the language features you have learned so far (actually, you would need a few commands to control devices like the keyboard, mouse, disks, etc., but that's all).

Proving that claim is a nontrivial exercise first accomplished by Alan Turing, one of the first computer scientists (some would argue that he was a mathematician, but a lot of early computer scientists started as mathematicians). Accordingly, it is known as the Turing Thesis. If you take a course on the Theory of Computation, you will have a chance to see the proof.

To give you an idea of what you can do with the tools you have learned so far, we'll evaluate a few recursively defined mathematical functions. A recursive definition is similar to a circular definition, in the sense that the definition contains a reference to the thing being defined. A truly circular definition is not very useful:

frabjuous: An adjective used to describe something that is frabjuous.

If you saw that definition in the dictionary, you might be annoyed. On the other hand, if you looked up the definition of the mathematical function factorial, you might get something like this:

$$0! = 1$$
$$n! = n(n-1)!$$

This definition says that the factorial of 0 is 1, and the factorial of any other value, n, is n multiplied by the factorial of $n - 1$.

So 3! is 3 times 2!, which is 2 times 1!, which is 1 times 0!. Putting it all together, 3! equals 3 times 2 times 1 times 1, which is 6.

If you can write a recursive definition of something, you can usually write a Python program to evaluate it. The first step is to decide what the parameters are for this function. With little effort, you should conclude that `factorial` has a single parameter:

```
def factorial(n):
```

If the argument happens to be 0, all we have to do is return 1:

```
def factorial(n):
  if n == 0:
    return 1
```

Otherwise, and this is the interesting part, we have to make a recursive call to find the factorial of $n-1$ and then multiply it by n:

```
def factorial(n):
  if n == 0:
    return 1
  else:
    recurse = factorial(n-1)
    result = n * recurse
    return result
```

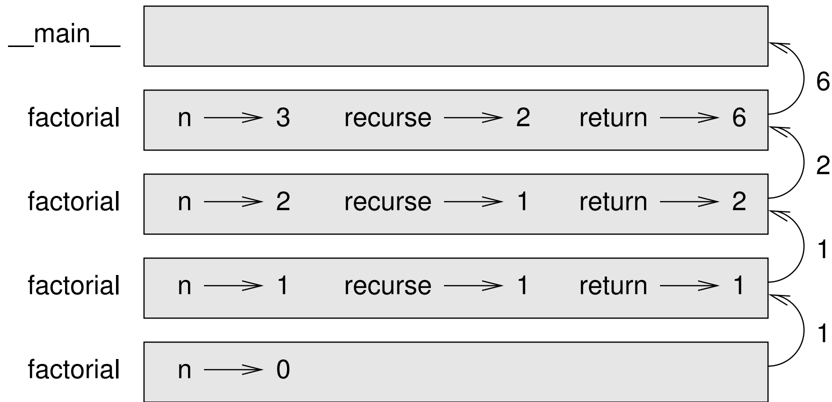
The flow of execution for this program is similar to the flow of `countdown` in Section 4.9. If we call `factorial` with the value 3:

Since 3 is not 0, we take the second branch and calculate the factorial of `n-1`...

> Since 2 is not 0, we take the second branch and calculate the factorial of `n-1`...
>
> > Since 1 is not 0, we take the second branch and calculate the factorial of `n-1`...
> >
> > > Since 0 *is* 0, we take the first branch and return 1 without making any more recursive calls.
> >
> > The return value (1) is multiplied by n, which is 1, and the result is returned.
>
> The return value (1) is multiplied by n, which is 2, and the result is returned.

The return value (2) is multiplied by n, which is 3, and the result, 6, becomes the return value of the function call that started the whole process.

Here is what the stack diagram looks like for this sequence of function calls:



The return values are shown being passed back up the stack. In each frame, the return value is the value of `result`, which is the product of `n` and `recurse`.

Notice that in the last frame, the local variables `recurse` and `result` do not exist, because the branch that creates them did not execute.

5.6 Leap of faith

Following the flow of execution is one way to read programs, but it can quickly become labyrinthine. An alternative is what we call the "leap of faith." When you come to a function call, instead of following the flow of execution, you *assume* that the function works correctly and returns the appropriate value.

In fact, you are already practicing this leap of faith when you use built-in functions. When you call `math.cos` or `math.exp`, you don't examine the implementations of those functions. You just assume that they work because the people who wrote the built-in functions were good programmers.

The same is true when you call one of your own functions. For example, in Section 5.4, we wrote a function called `isDivisible` that determines whether one number is divisible by another. Once we have convinced ourselves that this function is correct—by testing and examining the code—we can use the function without looking at the code again.

The same is true of recursive programs. When you get to the recursive call, instead of following the flow of execution, you should assume that the recursive call works (yields the correct result) and then ask yourself, "Assuming that I can find the factorial of $n - 1$, can I compute the factorial of n?" In this case, it is clear that you can, by multiplying by n.

Of course, it's a bit strange to assume that the function works correctly when you haven't finished writing it, but that's why it's called a leap of faith!

5.7 One more example

In the previous example, we used temporary variables to spell out the steps and to make the code easier to debug, but we could have saved a few lines:

```
def factorial(n):
  if n == 0:
    return 1
  else:
    return n * factorial(n-1)
```

From now on, we will tend to use the more concise form, but we recommend that you use the more explicit version while you are developing code. When you have it working, you can tighten it up if you are feeling inspired.

After `factorial`, the most common example of a recursively defined mathematical function is `fibonacci`, which has the following definition:

$$\begin{aligned}
\text{fibonacci}(0) &= 1 \\
\text{fibonacci}(1) &= 1 \\
\text{fibonacci}(n) &= \text{fibonacci}(n-1) + \text{fibonacci}(n-2);
\end{aligned}$$

Translated into Python, it looks like this:

```
def fibonacci (n):
  if n == 0 or n == 1:
    return 1
  else:
    return fibonacci(n-1) + fibonacci(n-2)
```

If you try to follow the flow of execution here, even for fairly small values of n, your head explodes. But according to the leap of faith, if you assume that the two recursive calls work correctly, then it is clear that you get the right result by adding them together.

5.8 Checking types

What happens if we call `factorial` and give it 1.5 as an argument?

```
>>> factorial (1.5)
RuntimeError: Maximum recursion depth exceeded
```

It looks like an infinite recursion. But how can that be? There is a base case—when `n == 0`. The problem is that the values of `n` *miss* the base case.

In the first recursive call, the value of `n` is 0.5. In the next, it is -0.5. From there, it gets smaller and smaller, but it will never be 0.

We have two choices. We can try to generalize the `factorial` function to work with floating-point numbers, or we can make `factorial` check the type of its argument. The first option is called the gamma function and it's a little beyond the scope of this book. So we'll go for the second.

We can use the built-in function `isinstance` to verify the type of the argument. While we're at it, we also make sure the argument is positive:

```
def factorial (n):
  if not isinstance(n, int):
    print "Factorial is only defined for integers."
    return -1
  elif n < 0:
    print "Factorial is only defined for positive integers."
    return -1
  elif n == 0:
    return 1
  else:
    return n * factorial(n-1)
```

Now we have three base cases. The first catches nonintegers. The second catches negative integers. In both cases, the program prints an error message and returns a special value, -1, to indicate that something went wrong:

```
>>> factorial ("fred")
Factorial is only defined for integers.
-1
>>> factorial (-2)
Factorial is only defined for positive integers.
-1
```

If we get past both checks, then we know that n is a positive integer, and we can prove that the recursion terminates.

This program demonstrates a pattern sometimes called a **guardian**. The first two conditionals act as guardians, protecting the code that follows from values that might cause an error. The guardians make it possible to prove the correctness of the code.

5.9 Glossary

fruitful function: A function that yields a return value.

return value: The value provided as the result of a function call.

temporary variable: A variable used to store an intermediate value in a complex calculation.

dead code: Part of a program that can never be executed, often because it appears after a `return` statement.

`None`**:** A special Python value returned by functions that have no return statement, or a return statement without an argument.

incremental development: A program development plan intended to avoid debugging by adding and testing only a small amount of code at a time.

scaffolding: Code that is used during program development but is not part of the final version.

guardian: A condition that checks for and handles circumstances that might cause an error.

Chapter 6

Iteration

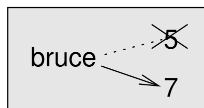
6.1 Multiple assignment

As you may have discovered, it is legal to make more than one assignment to the same variable. A new assignment makes an existing variable refer to a new value (and stop referring to the old value).

```
bruce = 5
print bruce,
bruce = 7
print bruce
```

The output of this program is `5 7`, because the first time `bruce` is printed, his value is 5, and the second time, his value is 7. The comma at the end of the first `print` statement suppresses the newline after the output, which is why both outputs appear on the same line.

Here is what **multiple assignment** looks like in a state diagram:



With multiple assignment it is especially important to distinguish between an assignment operation and a statement of equality. Because Python uses the equal sign (`=`) for assignment, it is tempting to interpret a statement like `a = b` as a statement of equality. It is not!

First, equality is commutative and assignment is not. For example, in mathematics, if $a = 7$ then $7 = a$. But in Python, the statement `a = 7` is legal and `7 = a` is not.

Furthermore, in mathematics, a statement of equality is always true. If $a = b$ now, then a will always equal b. In Python, an assignment statement can make two variables equal, but they don't have to stay that way:

```
a = 5
b = a    # a and b are now equal
a = 3    # a and b are no longer equal
```

The third line changes the value of `a` but does not change the value of `b`, so they are no longer equal. (In some programming languages, a different symbol is used for assignment, such as `<-` or `:=`, to avoid confusion.)

Although multiple assignment is frequently helpful, you should use it with caution. If the values of variables change frequently, it can make the code difficult to read and debug.

6.2 The `while` statement

Computers are often used to automate repetitive tasks. Repeating identical or similar tasks without making errors is something that computers do well and people do poorly.

We have seen two programs, `nLines` and `countdown`, that use recursion to perform repetition, which is also called **iteration**. Because iteration is so common, Python provides several language features to make it easier. The first feature we are going to look at is the `while` statement.

Here is what `countdown` looks like with a `while` statement:

```
def countdown(n):
  while n > 0:
    print n
    n = n-1
  print "Blastoff!"
```

Since we removed the recursive call, this function is not recursive.

You can almost read the `while` statement as if it were English. It means, "While `n` is greater than 0, continue displaying the value of `n` and then reducing the value of `n` by 1. When you get to 0, display the word `Blastoff!`"

More formally, here is the flow of execution for a `while` statement:

1. Evaluate the condition, yielding `0` or `1`.
2. If the condition is false (0), exit the `while` statement and continue execution at the next statement.
3. If the condition is true (1), execute each of the statements in the body and then go back to step 1.

The body consists of all of the statements below the header with the same indentation.

This type of flow is called a **loop** because the third step loops back around to the top. Notice that if the condition is false the first time through the loop, the statements inside the loop are never executed.

The body of the loop should change the value of one or more variables so that eventually the condition becomes false and the loop terminates. Otherwise the loop will repeat forever, which is called an **infinite loop**. An endless source of amusement for computer scientists is the observation that the directions on shampoo, "Lather, rinse, repeat," are an infinite loop.

In the case of `countdown`, we can prove that the loop terminates because we know that the value of `n` is finite, and we can see that the value of `n` gets smaller each time through the loop, so eventually we have to get to 0. In other cases, it is not so easy to tell:

```
def sequence(n):
  while n != 1:
    print n,
    if n%2 == 0:        # n is even
      n = n/2
    else:               # n is odd
      n = n*3+1
```

The condition for this loop is `n != 1`, so the loop will continue until `n` is `1`, which will make the condition false.

Each time through the loop, the program outputs the value of `n` and then checks whether it is even or odd. If it is even, the value of `n` is divided by 2. If it is odd, the value is replaced by `n*3+1`. For example, if the starting value (the argument passed to `sequence`) is 3, the resulting sequence is 3, 10, 5, 16, 8, 4, 2, 1.

Since `n` sometimes increases and sometimes decreases, there is no obvious proof that `n` will ever reach 1, or that the program terminates. For some particular values of `n`, we can prove termination. For example, if the starting value is a power of two, then the value of `n` will be even each time through the loop until it reaches 1. The previous example ends with such a sequence, starting with 16.

Particular values aside, the interesting question is whether we can prove that this program terminates for *all positive values* of `n`. So far, no one has been able to prove it *or* disprove it!

> *As an exercise, rewrite the function* `nLines` *from Section 4.9 using iteration instead of recursion.*

6.3 Tables

One of the things loops are good for is generating tabular data. Before computers were readily available, people had to calculate logarithms, sines and cosines, and other mathematical functions by hand. To make that easier, mathematics books contained long tables listing the values of these functions. Creating the tables was slow and boring, and they tended to be full of errors.

When computers appeared on the scene, one of the initial reactions was, "This is great! We can use the computers to generate the tables, so there will be no errors." That turned out to be true (mostly) but shortsighted. Soon thereafter, computers and calculators were so pervasive that the tables became obsolete.

Well, almost. For some operations, computers use tables of values to get an approximate answer and then perform computations to improve the approximation. In some cases, there have been errors in the underlying tables, most famously in the table the Intel Pentium used to perform floating-point division.

Although a log table is not as useful as it once was, it still makes a good example of iteration. The following program outputs a sequence of values in the left column and their logarithms in the right column:

```
x = 1.0
while x < 10.0:
  print x, '\t', math.log(x)
  x = x + 1.0
```

The string `'\t'` represents a **tab** character.

As characters and strings are displayed on the screen, an invisible marker called the **cursor** keeps track of where the next character will go. After a `print` statement, the cursor normally goes to the beginning of the next line.

The tab character shifts the cursor to the right until it reaches one of the tab stops. Tabs are useful for making columns of text line up, as in the output of the previous program:

```
1.0     0.0
2.0     0.69314718056
3.0     1.09861228867
4.0     1.38629436112
5.0     1.60943791243
6.0     1.79175946923
7.0     1.94591014906
8.0     2.07944154168
9.0     2.19722457734
```

If these values seem odd, remember that the `log` function uses base `e`. Since powers of two are so important in computer science, we often want to find logarithms with respect to base 2. To do that, we can use the following formula:

$$\log_2 x = \frac{\log_e x}{\log_e 2}$$

Changing the output statement to:

```
print x, '\t',  math.log(x)/math.log(2.0)
```

yields:

```
1.0     0.0
2.0     1.0
3.0     1.58496250072
4.0     2.0
5.0     2.32192809489
6.0     2.58496250072
7.0     2.80735492206
8.0     3.0
9.0     3.16992500144
```

We can see that 1, 2, 4, and 8 are powers of two because their logarithms base 2 are round numbers. If we wanted to find the logarithms of other powers of two, we could modify the program like this:

```
x = 1.0
while x < 100.0:
  print x, '\t', math.log(x)/math.log(2.0)
  x = x * 2.0
```

Now instead of adding something to `x` each time through the loop, which yields an arithmetic sequence, we multiply `x` by something, yielding a geometric sequence. The result is:

```
1.0     0.0
2.0     1.0
4.0     2.0
8.0     3.0
16.0    4.0
32.0    5.0
64.0    6.0
```

Because of the tab characters between the columns, the position of the second column does not depend on the number of digits in the first column.

Logarithm tables may not be useful any more, but for computer scientists, knowing the powers of two is!

> *As an exercise, modify this program so that it outputs the powers of two up to 65,536 (that's 2^{16}). Print it out and memorize it.*

The backslash character in `'\t'` indicates the beginning of an **escape sequence**. Escape sequences are used to represent invisible characters like tabs and newlines. The sequence `\n` represents a newline.

An escape sequence can appear anywhere in a string; in the example, the tab escape sequence is the only thing in the string.

How do you think you represent a backslash in a string?

> *As an exercise, write a single string that*

```
produces
        this
                output.
```

6.4 Two-dimensional tables

A two-dimensional table is a table where you read the value at the intersection of a row and a column. A multiplication table is a good example. Let's say you want to print a multiplication table for the values from 1 to 6.

A good way to start is to write a loop that prints the multiples of 2, all on one line:

```
i = 1
while i <= 6:
  print 2*i, '   ',
  i = i + 1
print
```

The first line initializes a variable named `i`, which acts as a counter or **loop variable**. As the loop executes, the value of `i` increases from 1 to 6. When `i` is 7, the loop terminates. Each time through the loop, it displays the value of `2*i`, followed by three spaces.

Again, the comma in the `print` statement suppresses the newline. After the loop completes, the second `print` statement starts a new line.

The output of the program is:

```
2       4       6       8       10      12
```

So far, so good. The next step is to **encapsulate** and **generalize**.

6.5 Encapsulation and generalization

Encapsulation is the process of wrapping a piece of code in a function, allowing you to take advantage of all the things functions are good for. You have seen two examples of encapsulation: `printParity` in Section 4.5; and `isDivisible` in Section 5.4.

Generalization means taking something specific, such as printing the multiples of 2, and making it more general, such as printing the multiples of any integer.

This function encapsulates the previous loop and generalizes it to print multiples of `n`:

```
def printMultiples(n):
  i = 1
  while i <= 6:
    print n*i, '\t',
    i = i + 1
  print
```

To encapsulate, all we had to do was add the first line, which declares the name of the function and the parameter list. To generalize, all we had to do was replace the value 2 with the parameter `n`.

If we call this function with the argument 2, we get the same output as before. With the argument 3, the output is:

```
3       6       9       12      15      18
```

With the argument 4, the output is:

```
4       8       12      16      20      24
```

By now you can probably guess how to print a multiplication table—by calling `printMultiples` repeatedly with different arguments. In fact, we can use another loop:

```
i = 1
while i <= 6:
  printMultiples(i)
  i = i + 1
```

Notice how similar this loop is to the one inside `printMultiples`. All we did was replace the `print` statement with a function call.

The output of this program is a multiplication table:

```
1       2       3       4       5       6
2       4       6       8       10      12
3       6       9       12      15      18
4       8       12      16      20      24
5       10      15      20      25      30
6       12      18      24      30      36
```

6.6 More encapsulation

To demonstrate encapsulation again, let's take the code from the end of Section 6.5 and wrap it up in a function:

```
def printMultTable():
  i = 1
  while i <= 6:
    printMultiples(i)
    i = i + 1
```

This process is a common **development plan**. We develop code by writing lines of code outside any function, or typing them in to the interpreter. When we get the code working, we extract it and wrap it up in a function.

This development plan is particularly useful if you don't know, when you start writing, how to divide the program into functions. This approach lets you design as you go along.

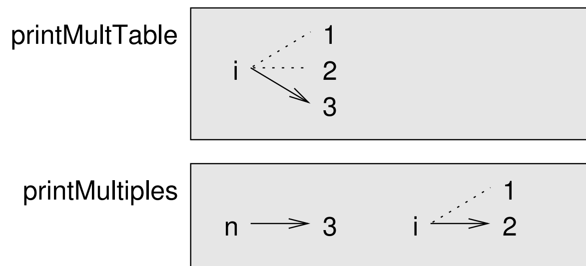
6.7 Local variables

You might be wondering how we can use the same variable, `i`, in both `printMultiples` and `printMultTable`. Doesn't it cause problems when one of the functions changes the value of the variable?

The answer is no, because the `i` in `printMultiples` and the `i` in `printMultTable` are *not* the same variable.

Variables created inside a function definition are local; you can't access a local variable from outside its "home" function. That means you are free to have multiple variables with the same name as long as they are not in the same function.

The stack diagram for this program shows that the two variables named `i` are not the same variable. They can refer to different values, and changing one does not affect the other.



The value of `i` in `printMultTable` goes from 1 to 6. In the diagram it happens to be 3. The next time through the loop it will be 4. Each time through the loop, `printMultTable` calls `printMultiples` with the current value of `i` as an argument. That value gets assigned to the parameter `n`.

Inside `printMultiples`, the value of `i` goes from 1 to 6. In the diagram, it happens to be 2. Changing this variable has no effect on the value of `i` in `printMultTable`.

It is common and perfectly legal to have different local variables with the same name. In particular, names like `i` and `j` are used frequently as loop variables. If you avoid using them in one function just because you used them somewhere else, you will probably make the program harder to read.

6.8 More generalization

As another example of generalization, imagine you wanted a program that would print a multiplication table of any size, not just the six-by-six table. You could add a parameter to `printMultTable`:

```
def printMultTable(high):
  i = 1
  while i <= high:
    printMultiples(i)
    i = i + 1
```

We replaced the value 6 with the parameter `high`. If we call `printMultTable` with the argument 7, it displays:

```
1       2       3       4       5       6
2       4       6       8       10      12
3       6       9       12      15      18
4       8       12      16      20      24
5       10      15      20      25      30
6       12      18      24      30      36
7       14      21      28      35      42
```

This is fine, except that we probably want the table to be square—with the same number of rows and columns. To do that, we add another parameter to `printMultiples` to specify how many columns the table should have.

Just to be annoying, we call this parameter `high`, demonstrating that different functions can have parameters with the same name (just like local variables). Here's the whole program:

```
def printMultiples(n, high):
  i = 1
  while i <= high:
    print n*i, '\t',
    i = i + 1
  print

def printMultTable(high):
  i = 1
  while i <= high:
    printMultiples(i, high)
    i = i + 1
```

Notice that when we added a new parameter, we had to change the first line of the function (the function heading), and we also had to change the place where

the function is called in `printMultTable`.

As expected, this program generates a square seven-by-seven table:

```
1       2       3       4       5       6       7
2       4       6       8       10      12      14
3       6       9       12      15      18      21
4       8       12      16      20      24      28
5       10      15      20      25      30      35
6       12      18      24      30      36      42
7       14      21      28      35      42      49
```

When you generalize a function appropriately, you often get a program with capabilities you didn't plan. For example, you might notice that, because $ab = ba$, all the entries in the table appear twice. You could save ink by printing only half the table. To do that, you only have to change one line of `printMultTable`. Change

```
printMultiples(i, high)
```

to

```
printMultiples(i, i)
```

and you get

```
1
2       4
3       6       9
4       8       12      16
5       10      15      20      25
6       12      18      24      30      36
7       14      21      28      35      42      49
```

As an exercise, trace the execution of this version of `printMultTable` *and figure out how it works.*

6.9 Functions

A few times now, we have mentioned "all the things functions are good for." By now, you might be wondering what exactly those things are. Here are some of them:

- Giving a name to a sequence of statements makes your program easier to read and debug.

- Dividing a long program into functions allows you to separate parts of the program, debug them in isolation, and then compose them into a whole.
- Functions facilitate both recursion and iteration.
- Well-designed functions are often useful for many programs. Once you write and debug one, you can reuse it.

6.10 Glossary

multiple assignment: Making more than one assignment to the same variable during the execution of a program.

iteration: Repeated execution of a set of statements using either a recursive function call or a loop.

loop: A statement or group of statements that execute repeatedly until a terminating condition is satisfied.

infinite loop: A loop in which the terminating condition is never satisfied.

body: The statements inside a loop.

loop variable: A variable used as part of the terminating condition of a loop.

tab: A special character that causes the cursor to move to the next tab stop on the current line.

newline: A special character that causes the cursor to move to the beginning of the next line.

cursor: An invisible marker that keeps track of where the next character will be printed.

escape sequence: An escape character (\) followed by one or more printable characters used to designate a nonprintable character.

encapsulate: To divide a large complex program into components (like functions) and isolate the components from each other (by using local variables, for example).

generalize: To replace something unnecessarily specific (like a constant value) with something appropriately general (like a variable or parameter). Generalization makes code more versatile, more likely to be reused, and sometimes even easier to write.

development plan: A process for developing a program. In this chapter, we demonstrated a style of development based on developing code to do simple, specific things and then encapsulating and generalizing.

Chapter 7

Strings

7.1 A compound data type

So far we have seen three types: `int`, `float`, and `string`. Strings are qualitatively different from the other two because they are made up of smaller pieces—characters.

Types that comprise smaller pieces are called **compound data types**. Depending on what we are doing, we may want to treat a compound data type as a single thing, or we may want to access its parts. This ambiguity is useful.

The bracket operator selects a single character from a string.

```
>>> fruit = "banana"
>>> letter = fruit[1]
>>> print letter
```

The expression `fruit[1]` selects character number 1 from `fruit`. The variable `letter` refers to the result. When we display `letter`, we get a surprise:

```
a
```

The first letter of `"banana"` is not `a`. Unless you are a computer scientist. In that case you should think of the expression in brackets as an offset from the beginning of the string, and the offset of the first letter is zero. So `b` is the 0th letter ("zero-eth") of `"banana"`, `a` is the 1th letter ("one-eth"), and `n` is the 2th ("two-eth") letter.

To get the first letter of a string, you just put 0, or any expression with the value 0, in the brackets:

```
>>> letter = fruit[0]
>>> print letter
b
```

The expression in brackets is called an **index**. An index specifies a member of an ordered set, in this case the set of characters in the string. The index *indicates* which one you want, hence the name. It can be any integer expression.

7.2 Length

The `len` function returns the number of characters in a string:

```
>>> fruit = "banana"
>>> len(fruit)
6
```

To get the last letter of a string, you might be tempted to try something like this:

```
length = len(fruit)
last = fruit[length]       # ERROR!
```

That won't work. It causes the runtime error `IndexError: string index out of range`. The reason is that there is no 6th letter in `"banana"`. Since we started counting at zero, the six letters are numbered 0 to 5. To get the last character, we have to subtract 1 from `length`:

```
length = len(fruit)
last = fruit[length-1]
```

Alternatively, we can use negative indices, which count backward from the end of the string. The expression `fruit[-1]` yields the last letter, `fruit[-2]` yields the second to last, and so on.

7.3 Traversal and the `for` loop

A lot of computations involve processing a string one character at a time. Often they start at the beginning, select each character in turn, do something to it, and continue until the end. This pattern of processing is called a **traversal**. One way to encode a traversal is with a `while` statement:

```
index = 0
while index < len(fruit):
  letter = fruit[index]
  print letter
  index = index + 1
```

This loop traverses the string and displays each letter on a line by itself. The loop condition is `index < len(fruit)`, so when `index` is equal to the length of the string, the condition is false, and the body of the loop is not executed. The last character accessed is the one with the index `len(fruit)-1`, which is the last character in the string.

> *As an exercise, write a function that takes a string as an argument and outputs the letters backward, one per line.*

Using an index to traverse a set of values is so common that Python provides an alternative, simpler syntax—the `for` loop:

```
for char in fruit:
  print char
```

Each time through the loop, the next character in the string is assigned to the variable `char`. The loop continues until no characters are left.

The following example shows how to use concatenation and a `for` loop to generate an abecedarian series. "Abecedarian" refers to a series or list in which the elements appear in alphabetical order. For example, in Robert McCloskey's book *Make Way for Ducklings*, the names of the ducklings are Jack, Kack, Lack, Mack, Nack, Ouack, Pack, and Quack. This loop outputs these names in order:

```
prefixes = "JKLMNOPQ"
suffix = "ack"

for letter in prefixes:
  print letter + suffix
```

The output of this program is:

```
Jack
Kack
Lack
Mack
Nack
Oack
Pack
Qack
```

Of course, that's not quite right because "Ouack" and "Quack" are misspelled.

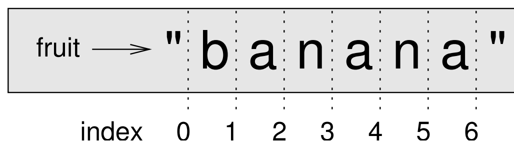
> *As an exercise, modify the program to fix this error.*

7.4 String slices

A segment of a string is called a **slice**. Selecting a slice is similar to selecting a character:

```
>>> s = "Peter, Paul, and Mary"
>>> print s[0:5]
Peter
>>> print s[7:11]
Paul
>>> print s[17:21]
Mary
```

The operator `[n:m]` returns the part of the string from the "n-eth" character to the "m-eth" character, including the first but excluding the last. This behavior is counterintuitive; it makes more sense if you imagine the indices pointing *between* the characters, as in the following diagram:



If you omit the first index (before the colon), the slice starts at the beginning of the string. If you omit the second index, the slice goes to the end of the string. Thus:

```
>>> fruit = "banana"
>>> fruit[:3]
'ban'
>>> fruit[3:]
'ana'
```

What do you think `s[:]` means?

7.5 String comparison

The comparison operators work on strings. To see if two strings are equal:

```
if word == "banana":
  print  "Yes, we have no bananas!"
```

Other comparison operations are useful for putting words in alphabetical order:

```
if word < "banana":
  print "Your word," + word + ", comes before banana."
elif word > "banana":
  print "Your word," + word + ", comes after banana."
else:
  print "Yes, we have no bananas!"
```

You should be aware, though, that Python does not handle upper- and lowercase letters the same way that people do. All the uppercase letters come before all the lowercase letters. As a result:

```
Your word, Zebra, comes before banana.
```

A common way to address this problem is to convert strings to a standard format, such as all lowercase, before performing the comparison. A more difficult problem is making the program realize that zebras are not fruit.

7.6 Strings are immutable

It is tempting to use the `[]` operator on the left side of an assignment, with the intention of changing a character in a string. For example:

```
greeting = "Hello, world!"
greeting[0] = 'J'            # ERROR!
print greeting
```

Instead of producing the output `Jello, world!`, this code produces the runtime error `TypeError: object doesn't support item assignment`.

Strings are **immutable**, which means you can't change an existing string. The best you can do is create a new string that is a variation on the original:

```
greeting = "Hello, world!"
newGreeting = 'J' + greeting[1:]
print newGreeting
```

The solution here is to concatenate a new first letter onto a slice of `greeting`. This operation has no effect on the original string.

7.7 A find function

What does the following function do?

```
def find(str, ch):
  index = 0
  while index < len(str):
    if str[index] == ch:
      return index
    index = index + 1
  return -1
```

In a sense, find is the opposite of the [] operator. Instead of taking an index and extracting the corresponding character, it takes a character and finds the index where that character appears. If the character is not found, the function returns -1.

This is the first example we have seen of a return statement inside a loop. If str[index] == ch, the function returns immediately, breaking out of the loop prematurely.

If the character doesn't appear in the string, then the program exits the loop normally and returns -1.

This pattern of computation is sometimes called a "eureka" traversal because as soon as we find what we are looking for, we can cry "Eureka!" and stop looking.

> *As an exercise, modify the* find *function so that it has a third parameter, the index in the string where it should start looking.*

7.8 Looping and counting

The following program counts the number of times the letter a appears in a string:

```
fruit = "banana"
count = 0
for char in fruit:
  if char == 'a':
    count = count + 1
print count
```

This program demonstrates another pattern of computation called a **counter**. The variable count is initialized to 0 and then incremented each time an a is found. (To **increment** is to increase by one; it is the opposite of **decrement**, and unrelated to "excrement," which is a noun.) When the loop exits, count contains the result—the total number of a's.

As an exercise, encapsulate this code in a function named `countLetters`*, and generalize it so that it accepts the string and the letter as arguments.*

As a second exercise, rewrite this function so that instead of traversing the string, it uses the three-parameter version of `find` *from the previous.*

7.9 The `string` module

The `string` module contains useful functions that manipulate strings. As usual, we have to import the module before we can use it:

```
>>> import string
```

The `string` module includes a function named `find` that does the same thing as the function we wrote. To call it we have to specify the name of the module and the name of the function using dot notation.

```
>>> fruit = "banana"
>>> index = string.find(fruit, "a")
>>> print index
1
```

This example demonstrates one of the benefits of modules—they help avoid collisions between the names of built-in functions and user-defined functions. By using dot notation we can specify which version of `find` we want.

Actually, `string.find` is more general than our version. First, it can find substrings, not just characters:

```
>>> string.find("banana", "na")
2
```

Also, it takes an additional argument that specifies the index it should start at:

```
>>> string.find("banana", "na", 3)
4
```

Or it can take two additional arguments that specify a range of indices:

```
>>> string.find("bob", "b", 1, 2)
-1
```

In this example, the search fails because the letter *b* does not appear in the index range from `1` to `2` (not including `2`).

7.10 Character classification

It is often helpful to examine a character and test whether it is upper- or lowercase, or whether it is a character or a digit. The `string` module provides several constants that are useful for these purposes.

The string `string.lowercase` contains all of the letters that the system considers to be lowercase. Similarly, `string.uppercase` contains all of the uppercase letters. Try the following and see what you get:

```
>>> print string.lowercase
>>> print string.uppercase
>>> print string.digits
```

We can use these constants and `find` to classify characters. For example, if `find(lowercase, ch)` returns a value other than `-1`, then `ch` must be lowercase:

```
def isLower(ch):
  return string.find(string.lowercase, ch) != -1
```

Alternatively, we can take advantage of the `in` operator, which determines whether a character appears in a string:

```
def isLower(ch):
  return ch in string.lowercase
```

As yet another alternative, we can use the comparison operator:

```
def isLower(ch):
  return 'a' <= ch <= 'z'
```

If `ch` is between *a* and *z*, it must be a lowercase letter.

> *As an exercise, discuss which version of* `isLower` *you think will be fastest. Can you think of other reasons besides speed to prefer one or the other?*

Another constant defined in the `string` module may surprise you when you print it:

```
>>> print string.whitespace
```

Whitespace characters move the cursor without printing anything. They create the white space between visible characters (at least on white paper). The constant `string.whitespace` contains all the whitespace characters, including space, tab (`\t`), and newline (`\n`).

There are other useful functions in the `string` module, but this book isn't intended to be a reference manual. On the other hand, the *Python Library Reference* is. Along with a wealth of other documentation, it's available from the Python website, `www.python.org`.

7.11 Glossary

compound data type: A data type in which the values are made up of components, or elements, that are themselves values.

traverse: To iterate through the elements of a set, performing a similar operation on each.

index: A variable or value used to select a member of an ordered set, such as a character from a string.

slice: A part of a string specified by a range of indices.

mutable: A compound data types whose elements can be assigned new values.

counter: A variable used to count something, usually initialized to zero and then incremented.

increment: To increase the value of a variable by one.

decrement: To decrease the value of a variable by one.

whitespace: Any of the characters that move the cursor without printing visible characters. The constant `string.whitespace` contains all the whitespace characters.

Chapter 8

Lists

A **list** is an ordered set of values, where each value is identified by an index. The values that make up a list are called its **elements**. Lists are similar to strings, which are ordered sets of characters, except that the elements of a list can have any type. Lists and strings—and other things that behave like ordered sets—are called **sequences**.

8.1 List values

There are several ways to create a new list; the simplest is to enclose the elements in square brackets ([and]):

```
[10, 20, 30, 40]
["spam", "bungee", "swallow"]
```

The first example is a list of four integers. The second is a list of three strings. The elements of a list don't have to be the same type. The following list contains a string, a float, an integer, and (mirabile dictu) another list:

```
["hello", 2.0, 5, [10, 20]]
```

A list within another list is said to be **nested**.

Lists that contain consecutive integers are common, so Python provides a simple way to create them:

```
>>> range(1,5)
[1, 2, 3, 4]
```

The range function takes two arguments and returns a list that contains all the integers from the first to the second, including the first but not including the second!

There are two other forms of range. With a single argument, it creates a list that starts at 0:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

If there is a third argument, it specifies the space between successive values, which is called the **step size**. This example counts from 1 to 10 by steps of 2:

```
>>> range(1, 10, 2)
[1, 3, 5, 7, 9]
```

Finally, there is a special list that contains no elements. It is called the empty list, and it is denoted [].

With all these ways to create lists, it would be disappointing if we couldn't assign list values to variables or pass lists as arguments to functions. We can.

```
vocabulary = ["ameliorate", "castigate", "defenestrate"]
numbers = [17, 123]
empty = []
print vocabulary, numbers, empty
['ameliorate', 'castigate', 'defenestrate'] [17, 123] []
```

8.2 Accessing elements

The syntax for accessing the elements of a list is the same as the syntax for accessing the characters of a string—the bracket operator ([]). The expression inside the brackets specifies the index. Remember that the indices start at 0:

```
print numbers[0]
numbers[1] = 5
```

The bracket operator can appear anywhere in an expression. When it appears on the left side of an assignment, it changes one of the elements in the list, so the one-eth element of numbers, which used to be 123, is now 5.

Any integer expression can be used as an index:

```
>>> numbers[3-2]
5
>>> numbers[1.0]
TypeError: sequence index must be integer
```

If you try to read or write an element that does not exist, you get a runtime error:

```
>>> numbers[2] = 5
IndexError: list assignment index out of range
```

If an index has a negative value, it counts backward from the end of the list:

```
>>> numbers[-1]
5
>>> numbers[-2]
17
>>> numbers[-3]
IndexError: list index out of range
```

`numbers[-1]` is the last element of the list, `numbers[-2]` is the second to last, and `numbers[-3]` doesn't exist.

It is common to use a loop variable as a list index.

```
horsemen = ["war", "famine", "pestilence", "death"]

i = 0
while i < 4:
  print horsemen[i]
  i = i + 1
```

This `while` loop counts from 0 to 4. When the loop variable `i` is 4, the condition fails and the loop terminates. So the body of the loop is only executed when `i` is 0, 1, 2, and 3.

Each time through the loop, the variable `i` is used as an index into the list, printing the `i`-eth element. This pattern of computation is called a **list traversal**.

8.3 List length

The function `len` returns the length of a list. It is a good idea to use this value as the upper bound of a loop instead of a constant. That way, if the size of the list changes, you won't have to go through the program changing all the loops; they will work correctly for any size list:

```
horsemen = ["war", "famine", "pestilence", "death"]

i = 0
while i < len(horsemen):
  print horsemen[i]
  i = i + 1
```

The last time the body of the loop is executed, `i` is `len(horsemen) - 1`, which is the index of the last element. When `i` is equal to `len(horsemen)`, the condition fails and the body is not executed, which is a good thing, because `len(horsemen)` is not a legal index.

Although a list can contain another list, the nested list still counts as a single element. The length of this list is four:

```
['spam!', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]
```

> *As an exercise, write a loop that traverses the previous list and prints the length of each element. What happens if you send an integer to* `len`*?*

8.4 List membership

`in` is a boolean operator that tests membership in a sequence. We used it in Section 7.10 with strings, but it also works with lists and other sequences:

```
>>> horsemen = ['war', 'famine', 'pestilence', 'death']
>>> 'pestilence' in horsemen
True
>>> 'debauchery' in horsemen
False
```

Since "pestilence" is a member of the `horsemen` list, the `in` operator returns true. Since "debauchery" is not in the list, `in` returns false.

We can use the `not` in combination with `in` to test whether an element is not a member of a list:

```
>>> 'debauchery' not in horsemen
True
```

8.5 Lists and `for` loops

The `for` loop we saw in Section 7.3 also works with lists. The generalized syntax of a `for` loop is:

```
for VARIABLE in LIST:
  BODY
```

This statement is equivalent to:

```
i = 0
while i < len(LIST):
  VARIABLE = LIST[i]
  BODY
  i = i + 1
```

The `for` loop is more concise because we can eliminate the loop variable, `i`. Here is the previous loop written with a `for` loop.

```
for horseman in horsemen:
  print horseman
```

It almost reads like English: "For (every) horseman in (the list of) horsemen, print (the name of the) horseman."

Any list expression can be used in a `for` loop:

```
for number in range(20):
  if number % 2 == 0:
    print  number

for fruit in ["banana", "apple", "quince"]:
  print "I like to eat " + fruit + "s!"
```

The first example prints all the even numbers between zero and nineteen. The second example expresses enthusiasm for various fruits.

8.6 List operations

The `+` operator concatenates lists:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> print c
[1, 2, 3, 4, 5, 6]
```

Similarly, the `*` operator repeats a list a given number of times:

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

The first example repeats [0] four times. The second example repeats the list [1, 2, 3] three times.

8.7 List slices

The slice operations we saw in Section 7.4 also work on lists:

```
>>> list = ['a', 'b', 'c', 'd', 'e', 'f']
>>> list[1:3]
['b', 'c']
>>> list[:4]
['a', 'b', 'c', 'd']
>>> list[3:]
['d', 'e', 'f']
```

If you omit the first index, the slice starts at the beginning. If you omit the second, the slice goes to the end. So if you omit both, the slice is really a copy of the whole list.

```
>>> list[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

8.8 Lists are mutable

Unlike strings, lists are mutable, which means we can change their elements. Using the bracket operator on the left side of an assignment, we can update one of the elements:

```
>>> fruit = ["banana", "apple", "quince"]
>>> fruit[0] = "pear"
>>> fruit[-1] = "orange"
>>> print fruit
['pear', 'apple', 'orange']
```

With the slice operator we can update several elements at once:

```
>>> list = ['a', 'b', 'c', 'd', 'e', 'f']
>>> list[1:3] = ['x', 'y']
>>> print list
['a', 'x', 'y', 'd', 'e', 'f']
```

We can also remove elements from a list by assigning the empty list to them:

```
>>> list = ['a', 'b', 'c', 'd', 'e', 'f']
>>> list[1:3] = []
>>> print list
['a', 'd', 'e', 'f']
```

And we can add elements to a list by squeezing them into an empty slice at the desired location:

```
>>> list = ['a', 'd', 'f']
>>> list[1:1] = ['b', 'c']
>>> print list
['a', 'b', 'c', 'd', 'f']
>>> list[4:4] = ['e']
>>> print list
['a', 'b', 'c', 'd', 'e', 'f']
```

8.9 List deletion

Using slices to delete list elements can be awkward, and therefore error-prone. Python provides an alternative that is more readable.

`del` removes an element from a list:

```
>>> a = ['one', 'two', 'three']
>>> del a[1]
>>> a
['one', 'three']
```

As you might expect, `del` handles negative indices and causes a runtime error if the index is out of range.

You can use a slice as an index for `del`:

```
>>> list = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del list[1:5]
>>> print list
['a', 'f']
```

As usual, slices select all the elements up to, but not including, the second index.

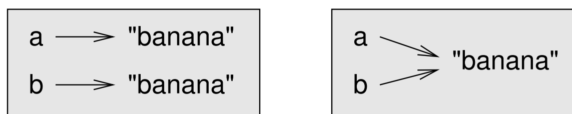
8.10 Objects and values

If we execute these assignment statements,

```
a = "banana"
b = "banana"
```

we know that `a` and `b` will refer to a string with the letters `"banana"`. But we can't tell whether they point to the *same* string.

There are two possible states:



In one case, `a` and `b` refer to two different things that have the same value. In the second case, they refer to the same thing. These "things" have names—they are called **objects**. An object is something a variable can refer to.

Every object has a unique **identifier**, which we can obtain with the `id` function. By printing the identifier of `a` and `b`, we can tell whether they refer to the same object.

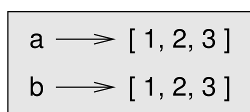
```
>>> id(a)
135044008
>>> id(b)
135044008
```

In fact, we get the same identifier twice, which means that Python only created one string, and both `a` and `b` refer to it.

Interestingly, lists behave differently. When we create two lists, we get two objects:

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> id(a)
135045528
>>> id(b)
135041704
```

So the state diagram looks like this:



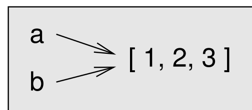
`a` and `b` have the same value but do not refer to the same object.

8.11 Aliasing

Since variables refer to objects, if we assign one variable to another, both variables refer to the same object:

```
>>> a = [1, 2, 3]
>>> b = a
```

In this case, the state diagram looks like this:



Because the same list has two different names, `a` and `b`, we say that it is **aliased**. Changes made with one alias affect the other:

```
>>> b[0] = 5
>>> print a
[5, 2, 3]
```

Although this behavior can be useful, it is sometimes unexpected or undesirable. In general, it is safer to avoid aliasing when you are working with mutable objects. Of course, for immutable objects, there's no problem. That's why Python is free to alias strings when it sees an opportunity to economize.

8.12 Cloning lists

If we want to modify a list and also keep a copy of the original, we need to be able to make a copy of the list itself, not just the reference. This process is sometimes called **cloning**, to avoid the ambiguity of the word "copy."

The easiest way to clone a list is to use the slice operator:

```
>>> a = [1, 2, 3]
>>> b = a[:]
>>> print b
[1, 2, 3]
```

Taking any slice of `a` creates a new list. In this case the slice happens to consist of the whole list.

Now we are free to make changes to `b` without worrying about `a`:

```
>>> b[0] = 5
>>> print a
[1, 2, 3]
```

> *As an exercise, draw a state diagram for* `a` *and* `b` *before and after this change.*

8.13 List parameters

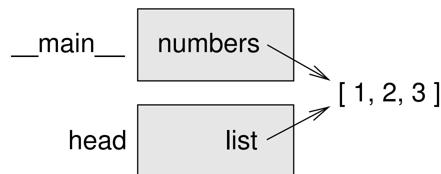
Passing a list as an argument actually passes a reference to the list, not a copy of the list. For example, the function `head` takes a list as an argument and returns the first element:

```
def head(list):
  return list[0]
```

Here's how it is used:

```
>>> numbers = [1, 2, 3]
>>> head(numbers)
1
```

The parameter `list` and the variable `numbers` are aliases for the same object. The state diagram looks like this:



Since the list object is shared by two frames, we drew it between them.

If a function modifies a list parameter, the caller sees the change. For example, `deleteHead` removes the first element from a list:

```
def deleteHead(list):
  del list[0]
```

Here's how `deleteHead` is used:

```
>>> numbers = [1, 2, 3]
>>> deleteHead(numbers)
>>> print numbers
[2, 3]
```

If a function returns a list, it returns a reference to the list. For example, `tail` returns a list that contains all but the first element of the given list:

```
def tail(list):
  return list[1:]
```

Here's how `tail` is used:

```
>>> numbers = [1, 2, 3]
>>> rest = tail(numbers)
>>> print rest
[2, 3]
```

Because the return value was created with the slice operator, it is a new list. Creating `rest`, and any subsequent changes to `rest`, have no effect on `numbers`.

8.14 Nested lists

A nested list is a list that appears as an element in another list. In this list, the three-eth element is a nested list:

```
>>> list = ["hello", 2.0, 5, [10, 20]]
```

If we print `list[3]`, we get `[10, 20]`. To extract an element from the nested list, we can proceed in two steps:

```
>>> elt = list[3]
>>> elt[0]
10
```

Or we can combine them:

```
>>> list[3][1]
20
```

Bracket operators evaluate from left to right, so this expression gets the three-eth element of `list` and extracts the one-eth element from it.

8.15 Matrices

Nested lists are often used to represent matrices. For example, the matrix:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

might be represented as:

```
>>> matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

`matrix` is a list with three elements, where each element is a row of the matrix. We can select an entire row from the matrix in the usual way:

```
>>> matrix[1]
[4, 5, 6]
```

Or we can extract a single element from the matrix using the double-index form:

```
>>> matrix[1][1]
5
```

The first index selects the row, and the second index selects the column. Although this way of representing matrices is common, it is not the only possibility. A small variation is to use a list of columns instead of a list of rows. Later we will see a more radical alternative using a dictionary.

8.16 Strings and lists

Two of the most useful functions in the `string` module involve lists of strings. The `split` function breaks a string into a list of words. By default, any number of whitespace characters is considered a word boundary:

```
>>> import string
>>> song = "The rain in Spain..."
>>> string.split(song)
['The', 'rain', 'in', 'Spain...']
```

An optional argument called a **delimiter** can be used to specify which characters to use as word boundaries. The following example uses the string `ai` as the delimiter:

```
>>> string.split(song, 'ai')
['The r', 'n in Sp', 'n...']
```

Notice that the delimiter doesn't appear in the list.

The `join` function is the inverse of `split`. It takes a list of strings and concatenates the elements with a space between each pair:

```
>>> list = ['The', 'rain', 'in', 'Spain...']
>>> string.join(list)
'The rain in Spain...'
```

Like `split`, `join` takes an optional delimiter that is inserted between elements:

```
>>> string.join(list, '_')
'The_rain_in_Spain...'
```

> *As an exercise, describe the relationship between* `string.join(string.split(song))` *and* `song`*. Are they the same for all strings? When would they be different?*

8.17 Glossary

list: A named collection of objects, where each object is identified by an index.

index: An integer variable or value that indicates an element of a list.

element: One of the values in a list (or other sequence). The bracket operator selects elements of a list.

sequence: Any of the data types that consist of an ordered set of elements, with each element identified by an index.

nested list: A list that is an element of another list.

list traversal: The sequential accessing of each element in a list.

object: A thing to which a variable can refer.

aliases: Multiple variables that contain references to the same object.

clone: To create a new object that has the same value as an existing object. Copying a reference to an object creates an alias but doesn't clone the object.

delimiter: A character or string used to indicate where a string should be split.

Chapter 9

Tuples

9.1 Mutability and tuples

So far, you have seen two compound types: strings, which are made up of characters; and lists, which are made up of elements of any type. One of the differences we noted is that the elements of a list can be modified, but the characters in a string cannot. In other words, strings are **immutable** and lists are **mutable**.

There is another type in Python called a **tuple** that is similar to a list except that it is immutable. Syntactically, a tuple is a comma-separated list of values:

```
>>> tuple = 'a', 'b', 'c', 'd', 'e'
```

Although it is not necessary, it is conventional to enclose tuples in parentheses:

```
>>> tuple = ('a', 'b', 'c', 'd', 'e')
```

To create a tuple with a single element, we have to include the final comma:

```
>>> t1 = ('a',)
>>> type(t1)
<type 'tuple'>
```

Without the comma, Python treats `('a')` as a string in parentheses:

```
>>> t2 = ('a')
>>> type(t2)
<type 'str'>
```

Syntax issues aside, the operations on tuples are the same as the operations on lists. The index operator selects an element from a tuple.

```
>>> tuple = ('a', 'b', 'c', 'd', 'e')
>>> tuple[0]
'a'
```

And the slice operator selects a range of elements.

```
>>> tuple[1:3]
('b', 'c')
```

But if we try to modify one of the elements of the tuple, we get an error:

```
>>> tuple[0] = 'A'
TypeError: object doesn't support item assignment
```

Of course, even if we can't modify the elements of a tuple, we can replace it with a different tuple:

```
>>> tuple = ('A',) + tuple[1:]
>>> tuple
('A', 'b', 'c', 'd', 'e')
```

9.2 Tuple assignment

Once in a while, it is useful to swap the values of two variables. With conventional assignment statements, we have to use a temporary variable. For example, to swap `a` and `b`:

```
>>> temp = a
>>> a = b
>>> b = temp
```

If we have to do this often, this approach becomes cumbersome. Python provides a form of **tuple assignment** that solves this problem neatly:

```
>>> a, b = b, a
```

The left side is a tuple of variables; the right side is a tuple of values. Each value is assigned to its respective variable. All the expressions on the right side are evaluated before any of the assignments. This feature makes tuple assignment quite versatile.

Naturally, the number of variables on the left and the number of values on the right have to be the same:

```
>>> a, b, c, d = 1, 2, 3
ValueError: unpack tuple of wrong size
```

9.3 Tuples as return values

Functions can return tuples as return values. For example, we could write a function that swaps two parameters:

```
def swap(x, y):
  return y, x
```

Then we can assign the return value to a tuple with two variables:

```
a, b = swap(a, b)
```

In this case, there is no great advantage in making `swap` a function. In fact, there is a danger in trying to encapsulate `swap`, which is the following tempting mistake:

```
def swap(x, y):      # incorrect version
  x, y = y, x
```

If we call this function like this:

```
swap(a, b)
```

then `a` and `x` are aliases for the same value. Changing `x` inside `swap` makes `x` refer to a different value, but it has no effect on `a` in `__main__`. Similarly, changing `y` has no effect on `b`.

This function runs without producing an error message, but it doesn't do what we intended. This is an example of a semantic error.

> *As an exercise, draw a state diagram for this function so that you can see why it doesn't work.*

9.4 Random numbers

Most computer programs do the same thing every time they execute, so they are said to be **deterministic**. Determinism is usually a good thing, since we expect the same calculation to yield the same result. For some applications, though, we want the computer to be unpredictable. Games are an obvious example, but there are more.

Making a program truly nondeterministic turns out to be not so easy, but there are ways to make it at least seem nondeterministic. One of them is to generate random numbers and use them to determine the outcome of the program. Python provides a built-in function that generates **pseudorandom** numbers, which are not truly random in the mathematical sense, but for our purposes they will do.

The `random` module contains a function called `random` that returns a floating-point number between 0.0 and 1.0. Each time you call `random`, you get the next number in a long series. To see a sample, run this loop:

```
import random

for i in range(10):
  x = random.random()
  print x
```

To generate a random number between 0.0 and an upper bound like `high`, multiply `x` by `high`.

> *As an exercise, generate a random number between* `low` *and* `high`*.*

> *As an additional exercise, generate a random* integer *between* `low` *and* `high`*, including both end points.*

9.5 List of random numbers

The first step is to generate a list of random values. `randomList` takes an integer argument and returns a list of random numbers with the given length. It starts with a list of `n` zeros. Each time through the loop, it replaces one of the elements with a random number. The return value is a reference to the complete list:

```
def randomList(n):
  s = [0] * n
  for i in range(n):
    s[i] = random.random()
  return s
```

We'll test this function with a list of eight elements. For purposes of debugging, it is a good idea to start small.

```
>>> randomList(8)
0.15156642489
0.498048560109
0.810894847068
0.360371157682
0.275119183077
0.328578797631
0.759199803101
0.800367163582
```

The numbers generated by `random` are supposed to be distributed uniformly, which means that every value is equally likely.

If we divide the range of possible values into equal-sized "buckets," and count the number of times a random value falls in each bucket, we should get roughly the same number in each.

We can test this theory by writing a program to divide the range into buckets and count the number of values in each.

9.6 Counting

A good approach to problems like this is to divide the problem into subproblems and look for subproblems that fit a computational pattern you have seen before.

In this case, we want to traverse a list of numbers and count the number of times a value falls in a given range. That sounds familiar. In Section 7.8, we wrote a program that traversed a string and counted the number of times a given letter appeared.

So, we can proceed by copying the old program and adapting it for the current problem. The original program was:

```
count = 0
for char in fruit:
  if char == 'a':
    count = count + 1
print count
```

The first step is to replace `fruit` with `t` and `char` with `num`. That doesn't change the program; it just makes it more readable.

The second step is to change the test. We aren't interested in finding letters. We want to see if `num` is between the given values `low` and `high`.

```
count = 0
for num in t:
  if low < num < high:
    count = count + 1
print count
```

The last step is to encapsulate this code in a function called `inBucket`. The parameters are the list and the values `low` and `high`.

```
def inBucket(t, low, high):
  count = 0
  for num in t:
    if low < num < high:
      count = count + 1
  return count
```

By copying and modifying an existing program, we were able to write this function quickly and save a lot of debugging time. This development plan is called **pattern matching**. If you find yourself working on a problem you have solved before, reuse the solution.

9.7 Many buckets

As the number of buckets increases, `inBucket` gets a little unwieldy. With two buckets, it's not bad:

```
low = inBucket(a, 0.0, 0.5)
high = inBucket(a, 0.5, 1)
```

But with four buckets it is getting cumbersome.

```
bucket1 = inBucket(a, 0.0, 0.25)
bucket2 = inBucket(a, 0.25, 0.5)
bucket3 = inBucket(a, 0.5, 0.75)
bucket4 = inBucket(a, 0.75, 1.0)
```

There are two problems. One is that we have to make up new variable names for each result. The other is that we have to compute the range for each bucket.

We'll solve the second problem first. If the number of buckets is `numBuckets`, then the width of each bucket is `1.0 / numBuckets`.

We'll use a loop to compute the range of each bucket. The loop variable, `i`, counts from 0 to `numBuckets-1`:

```
bucketWidth = 1.0 / numBuckets
for i in range(numBuckets):
  low = i * bucketWidth
  high = low + bucketWidth
  print low, "to", high
```

To compute the low end of each bucket, we multiply the loop variable by the bucket width. The high end is just a `bucketWidth` away.

With `numBuckets = 8`, the output is:

```
0.0 to 0.125
0.125 to 0.25
0.25 to 0.375
0.375 to 0.5
0.5 to 0.625
0.625 to 0.75
0.75 to 0.875
0.875 to 1.0
```

You can confirm that each bucket is the same width, that they don't overlap, and that they cover the entire range from 0.0 to 1.0.

Now back to the first problem. We need a way to store eight integers, using the loop variable to indicate one at a time. By now you should be thinking, "List!"

We have to create the bucket list outside the loop, because we only want to do it once. Inside the loop, we'll call `inBucket` repeatedly and update the `i`-eth element of the list:

```
numBuckets = 8
buckets = [0] * numBuckets
bucketWidth = 1.0 / numBuckets
for i in range(numBuckets):
  low = i * bucketWidth
  high = low + bucketWidth
  buckets[i] = inBucket(t, low, high)
print buckets
```

With a list of 1000 values, this code produces this bucket list:

```
[138, 124, 128, 118, 130, 117, 114, 131]
```

These numbers are fairly close to 125, which is what we expected. At least, they are close enough that we can believe the random number generator is working.

> *As an exercise, test this function with some longer lists, and see if the number of values in each bucket tends to level off.*

9.8 A single-pass solution

Although this program works, it is not as efficient as it could be. Every time it calls `inBucket`, it traverses the entire list. As the number of buckets increases, that gets to be a lot of traversals.

It would be better to make a single pass through the list and compute for each value the index of the bucket in which it falls. Then we can increment the appropriate counter.

In the previous section we took an index, `i`, and multiplied it by the `bucketWidth` to find the lower bound of a given bucket. Now we want to take a value in the range 0.0 to 1.0 and find the index of the bucket where it falls.

Since this problem is the inverse of the previous problem, we might guess that we should divide by `bucketWidth` instead of multiplying. That guess is correct.

Since `bucketWidth = 1.0 / numBuckets`, dividing by `bucketWidth` is the same as multiplying by `numBuckets`. If we multiply a number in the range 0.0 to 1.0 by `numBuckets`, we get a number in the range from 0.0 to `numBuckets`. If we round that number to the next lower integer, we get exactly what we are looking for—a bucket index:

```
numBuckets = 8
buckets = [0] * numBuckets
for i in t:
  index = int(i * numBuckets)
  buckets[index] = buckets[index] + 1
```

We used the `int` function to convert a floating-point number to an integer.

Is it possible for this calculation to produce an index that is out of range (either negative or greater than `len(buckets)-1`)?

A list like `buckets` that contains counts of the number of values in each range is called a **histogram**.

> *As an exercise, write a function called* `histogram` *that takes a list and a number of buckets as arguments and returns a histogram with the given number of buckets.*

9.9 Glossary

immutable type: A type in which the elements cannot be modified. Assignments to elements or slices of immutable types cause an error.

mutable type: A data type in which the elements can be modified. All mutable types are compound types. Lists and dictionaries are mutable data types; strings and tuples are not.

tuple: A sequence type that is similar to a list except that it is immutable. Tuples can be used wherever an immutable type is required, such as a key in a dictionary.

tuple assignment: An assignment to all of the elements in a tuple using a single assignment statement. Tuple assignment occurs in parallel rather than in sequence, making it useful for swapping values.

deterministic: A program that does the same thing each time it is called.

pseudorandom: A sequence of numbers that appear to be random but that are actually the result of a deterministic computation.

histogram: A list of integers in which each element counts the number of times something happens.

pattern matching: A program development plan that involves identifying a familiar computational pattern and copying the solution to a similar problem.

Chapter 10

Dictionaries

The compound types you have learned about—strings, lists, and tuples—use integers as indices. If you try to use any other type as an index, you get an error.

Dictionaries are similar to other compound types except that they can use any immutable type as an index. As an example, we will create a dictionary to translate English words into Spanish. For this dictionary, the indices are `strings`.

One way to create a dictionary is to start with the empty dictionary and add elements. The empty dictionary is denoted `{}`:

```
>>> eng2sp = {}
>>> eng2sp['one'] = 'uno'
>>> eng2sp['two'] = 'dos'
```

The first assignment creates a dictionary named `eng2sp`; the other assignments add new elements to the dictionary. We can print the current value of the dictionary in the usual way:

```
>>> print eng2sp
{'one': 'uno', 'two': 'dos'}
```

The elements of a dictionary appear in a comma-separated list. Each entry contains an index and a value separated by a colon. In a dictionary, the indices are called **keys**, so the elements are called **key-value pairs**.

Another way to create a dictionary is to provide a list of key-value pairs using the same syntax as the previous output:

```
>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
```

If we print the value of eng2sp again, we get a surprise:

```
>>> print eng2sp
{'one': 'uno', 'three': 'tres', 'two': 'dos'}
```

The key-value pairs are not in order! Fortunately, there is no reason to care about the order, since the elements of a dictionary are never indexed with integer indices. Instead, we use the keys to look up the corresponding values:

```
>>> print eng2sp['two']
'dos'
```

The key 'two' yields the value 'dos' even though it appears in the third key-value pair.

10.1 Dictionary operations

The del statement removes a key-value pair from a dictionary. For example, the following dictionary contains the names of various fruits and the number of each fruit in stock:

```
>>> inventory = {'apples': 430, 'bananas': 312, 'oranges': 525,
'pears': 217}
>>> print inventory
{'oranges': 525, 'apples': 430, 'pears': 217, 'bananas': 312}
```

If someone buys all of the pears, we can remove the entry from the dictionary:

```
>>> del inventory['pears']
>>> print inventory
{'oranges': 525, 'apples': 430, 'bananas': 312}
```

Or if we're expecting more pears soon, we might just change the value associated with pears:

```
>>> inventory['pears'] = 0
>>> print inventory
{'oranges': 525, 'apples': 430, 'pears': 0, 'bananas': 312}
```

The len function also works on dictionaries; it returns the number of key-value pairs:

```
>>> len(inventory)
4
```

10.2 Dictionary methods

A **method** is similar to a function—it takes arguments and returns a value—but the syntax is different. For example, the `keys` method takes a dictionary and returns a list of the keys that appear, but instead of the function syntax `keys(eng2sp)`, we use the method syntax `eng2sp.keys()`.

```
>>> eng2sp.keys()
['one', 'three', 'two']
```

This form of dot notation specifies the name of the function, `keys`, and the name of the object to apply the function to, `eng2sp`. The parentheses indicate that this method has no parameters.

A method call is called an **invocation**; in this case, we would say that we are invoking `keys` on the object `eng2sp`.

The `values` method is similar; it returns a list of the values in the dictionary:

```
>>> eng2sp.values()
['uno', 'tres', 'dos']
```

The `items` method returns both, in the form of a list of tuples—one for each key-value pair:

```
>>> eng2sp.items()
[('one','uno'), ('three', 'tres'), ('two', 'dos')]
```

The syntax provides useful type information. The square brackets indicate that this is a list. The parentheses indicate that the elements of the list are tuples.

If a method takes an argument, it uses the same syntax as a function call. For example, the method `has_key` takes a key and returns true (1) if the key appears in the dictionary:

```
>>> eng2sp.has_key('one')
True
>>> eng2sp.has_key('deux')
False
```

If you try to call a method without specifying an object, you get an error. In this case, the error message is not very helpful:

```
>>> has_key('one')
NameError: has_key
```

10.3 Aliasing and copying

Because dictionaries are mutable, you need to be aware of aliasing. Whenever two variables refer to the same object, changes to one affect the other.

If you want to modify a dictionary and keep a copy of the original, use the `copy` method. For example, `opposites` is a dictionary that contains pairs of opposites:

```
>>> opposites = {'up': 'down', 'right': 'wrong', 'true': 'false'}
>>> alias = opposites
>>> copy = opposites.copy()
```

`alias` and `opposites` refer to the same object; `copy` refers to a fresh copy of the same dictionary. If we modify `alias`, `opposites` is also changed:

```
>>> alias['right'] = 'left'
>>> opposites['right']
'left'
```

If we modify `copy`, `opposites` is unchanged:

```
>>> copy['right'] = 'privilege'
>>> opposites['right']
'left'
```

10.4 Sparse matrices

In Section 8.14, we used a list of lists to represent a matrix. That is a good choice for a matrix with mostly nonzero values, but consider a sparse matrix like this one:

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \end{bmatrix}$$

The list representation contains a lot of zeroes:

```
matrix = [ [0,0,0,1,0],
           [0,0,0,0,0],
           [0,2,0,0,0],
           [0,0,0,0,0],
           [0,0,0,3,0] ]
```

An alternative is to use a dictionary. For the keys, we can use tuples that contain the row and column numbers. Here is the dictionary representation of the same matrix:

```
matrix = {(0,3): 1, (2, 1): 2, (4, 3): 3}
```

We only need three key-value pairs, one for each nonzero element of the matrix. Each key is a tuple, and each value is an integer.

To access an element of the matrix, we could use the `[]` operator:

```
matrix[0,3]
1
```

Notice that the syntax for the dictionary representation is not the same as the syntax for the nested list representation. Instead of two integer indices, we use one index, which is a tuple of integers.

There is one problem. If we specify an element that is zero, we get an error, because there is no entry in the dictionary with that key:

```
>>> matrix[1,3]
KeyError: (1, 3)
```

The `get` method solves this problem:

```
>>> matrix.get((0,3), 0)
1
```

The first argument is the key; the second argument is the value `get` should return if the key is not in the dictionary:

```
>>> matrix.get((1,3), 0)
0
```

`get` definitely improves the semantics of accessing a sparse matrix. Shame about the syntax.

10.5 Hints

If you played around with the `fibonacci` function from Section 5.7, you might have noticed that the bigger the argument you provide, the longer the function takes to run. Furthermore, the run time increases very quickly. On one of our machines, `fibonacci(20)` finishes instantly, `fibonacci(30)` takes about a second, and `fibonacci(40)` takes roughly forever.