

The  
Pragmatic  
Programmers

# Practical Programming

Second Edition

An Introduction to  
Computer Science  
Using Python 3

Paul Grzes  
Jennifer Campbell  
Jason Montojo

Edited by Ethan Leighley



## What Readers Are Saying About *Practical Programming, Second Edition*

I wish I could go back in time and get this book in my library when I first learned programming! It's so much more engaging, practical, and accessible than the dry introductory programming books that I tried [and often failed] to comprehend as a kid. I love the authors' hands-on approach of writing explanations with code snippets that students can type into the Python prompt.

— Philip Guo

Creator of Online Python Tutor (<https://onlinepythontutor.com>); Assistant Professor, Department of Computer Science, University of Rochester

*Practical Programming* achieves just what it promises: a clear, accessible, useful introduction to programming for beginners. This isn't just a guide to how to type programs. The book provides foundations, collecting programming skills; a step-by-step, and visual, model of memory and execution and a design recipe that will help produce quality software.

— Steven Skiena

Senior Lecturer, Department of Computer Science, University of British Columbia

The second edition of this excellent text reflects the authors' many years of experience teaching Python to beginning students. Topics are presented so that each builds naturally to the next, and common errors and misconceptions are explicitly addressed. The exercises at the end of each chapter invite interested students to explore computer science and programming language topics.

—Kathleen M. Pearson

Director of Undergraduate Studies, Department of Computer and Information Science, University of Oregon

# Practical Programming, 2nd Edition

## An Introduction to Computer Science Using Python 3

Paul Gries

Jennifer Campbell

Jason Montojo

The Pragmatic Bookshelf

Durham, North Carolina

<http://pragprog.com>



# Pragmatic Bookshelf

Some of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where these designations appear in this book, and The Pragmatic Programmers LLC has no registered trademark or service mark registration for such designation, the designations are used in this book in a descriptive manner only. The Pragmatic Bookshelf logo, The Pragmatic Bookshelf, and the talking monkey are trademarks of The Pragmatic Programmers, LLC.

Every effort has been taken in the preparation of this book to ensure the quality of the content, to my knowledge to be accurate and valid; however, no guarantee can be made regarding the use of information, including programming examples, herein.

The Pragmatic authors, editors, and other people involved help you and your business to build software and have fun. For more information, see and visit the [Pragmatic](http://pragprog.com) site, those works at <http://pragprog.com>.

## The last few pages: the acknowledgments

Sam Ruby (edit)  
Patricia Lau (editing, LCC layout)  
Andy McCullough (composition)  
David J. Kelly (typesetting)  
David Munro (production)  
Juliette Rendell (index)  
P.J. McHugh (copyedit)

Copyright © 2005 The Pragmatic Programmers, LLC.  
All rights reserved.

This book is provided as sample material only and is not for resale. It is the copyright holder's intention that it be widely distributed to help software professionals develop their skills.

The Pragmatic Bookshelf logo  
ISBN 0-9712300-0-0  
Cover design by the Pragmatic Bookshelf design team  
Cover photo © iStockphoto.com/John T. Smith

# Contents

<u>Acknowledgments</u>	xi
<u>Preface</u>	xiii
<b>1. What's Programming?</b>	1
1.1 <u>Programs and Programming</u>	2
1.2 <u>What's a Programming Language?</u>	3
1.3 <u>What's a Run?</u>	4
1.4 <u>The Difference Between Directs, Directs, and Parentheses</u>	5
1.5 <u>Installing Python</u>	5
<b>2. Hello, Python</b>	7
2.1 <u>How Does a Computer Run a Python Program?</u>	7
2.2 <u>Expressions and Values: Arithmetic in Python</u>	8
2.3 <u>What Is a Type?</u>	12
2.4 <u>Variables and Computer Memory: Renaming Values</u>	16
2.5 <u>How Python Tells You Something Went Wrong</u>	29
2.6 <u>A Simple Statement That Starts Multiple Lines</u>	29
2.7 <u>Describing Code</u>	25
2.8 <u>Making Code Readable</u>	26
2.9 <u>The Goals of This Chapter</u>	27
2.10 <u>Exercises</u>	27
<b>3. Designing and Using Functions</b>	31
3.1 <u>Functions That Python Provides</u>	31
3.2 <u>Memory Addresses: How Python Keeps Track of Values</u>	33
3.3 <u>Defining Our Own Functions</u>	35
3.4 <u>Using New Variables for Temporary Storage</u>	38
3.5 <u>Stacked Function Calls in the Memory Model</u>	40
3.6 <u>Designing New Functions: A Recipe</u>	47
3.7 <u>Writing and Running a Program</u>	58

3.8	<u>Uninitializing A Return Statement With None</u>	60
3.9	<u>Dealing with Situations That Your Code Doesn't Handle</u>	61
3.10	<u>What Did You Call That?</u>	62
3.11	<u>Keywords</u>	63
<b>4.</b>	<b><u>Working with Text</u></b>	<b>65</b>
4.1	<u>Creating Strings of Characters</u>	65
4.2	<u>Using Special Characters in Strings</u>	68
4.3	<u>Creating &amp; Manipulating Strings</u>	71
4.4	<u>Formatting Information</u>	70
4.5	<u>Getting Information from the Keyboard</u>	73
4.6	<u>Multiple Assignment Statements in One Line</u>	73
4.7	<u>Exercises</u>	76
<b>5.</b>	<b><u>Making Choices</u></b>	<b>77</b>
5.1	<u>A Boolean Type</u>	77
5.2	<u>Choosing Which Statements to Execute</u>	80
5.3	<u>Nested If statements</u>	92
5.4	<u>Remembering the Structure of a Boolean Expression Evaluation</u>	92
5.5	<u>You Learned About Boolean: True or False?</u>	94
5.6	<u>Exercises</u>	94
<b>6.</b>	<b><u>A Modular Approach to Program Organization</u></b>	<b>98</b>
6.1	<u>Importing Modules</u>	100
6.2	<u>Defining Your Own Modules</u>	104
6.3	<u>Testing Your Code Semiautomatically</u>	108
6.4	<u>Decorators for Grouping Your Functions</u>	112
6.5	<u>Organizing Our Thoughts</u>	113
6.6	<u>Exercises</u>	114
<b>7.</b>	<b><u>Using Methods</u></b>	<b>115</b>
7.1	<u>Modules, Classes, and Methods</u>	115
7.2	<u>Calling Methods On Object-Oriented Ways</u>	117
7.3	<u>Exploring String Methods</u>	118
7.4	<u>What Are These Underscores?</u>	121
7.5	<u>A Mathematical Review</u>	126
7.6	<u>Exercises</u>	128
<b>8.</b>	<b><u>Sharing Collections of Data Using Lists</u></b>	<b>128</b>
8.1	<u>Sharing and Accessing Values in Lists</u>	129
8.2	<u>Modifying Lists</u>	132

<b>8.3</b>	<u>Operations on Lists</u>	34
<b>8.4</b>	<u>Slicing Lists</u>	37
<b>8.5</b>	<u>Altering: What's in a Name?</u>	39
<b>8.6</b>	<u>List Methods</u>	40
<b>8.7</b>	<u>Working with a List of Lists</u>	42
<b>8.8</b>	<u>A Summary List</u>	44
<b>8.9</b>	<u>Exercises</u>	44
<b>9.</b>	<b>Repeating Code Using Loops</b>	147
<b>9.1</b>	<u>Processing Items in a List</u>	147
<b>9.2</b>	<u>Processing Characters in Strings</u>	149
<b>9.3</b>	<u>Looping Through Nested Variables</u>	151
<b>9.4</b>	<u>Processing Lists Using Indices</u>	152
<b>9.5</b>	<u>Breaking Loops in Loops</u>	153
<b>9.6</b>	<u>Looping Until a Condition Is Reached</u>	155
<b>9.7</b>	<u>Repetition Based on User Input</u>	156
<b>9.8</b>	<u>Controlling Loops Using Break and Continue</u>	156
<b>9.9</b>	<u>Reporting What You've Learned</u>	158
<b>9.10</b>	<u>Exercises</u>	158
<b>10.</b>	<b>Reading and Writing Files</b>	171
<b>10.1</b>	<u>What Kinds of Files Are There?</u>	171
<b>10.2</b>	<u>Opening a File</u>	173
<b>10.3</b>	<u>Techniques for Reading Files</u>	178
<b>10.4</b>	<u>Files over the Internet</u>	181
<b>10.5</b>	<u>Writing Files</u>	182
<b>10.6</b>	<u>Writing Algorithms That Use File-Reading Techniques</u>	183
<b>10.7</b>	<u>Moving Data</u>	184
<b>10.8</b>	<u>Backing Up Data</u>	184
<b>10.9</b>	<u>Notes in File Format</u>	186
<b>10.10</b>	<u>Exercises</u>	187
<b>11.</b>	<b>Storing Data Using Other Collection Types</b>	198
<b>11.1</b>	<u>Storing Data Using Lists</u>	199
<b>11.2</b>	<u>Storing Data Using Tuples</u>	203
<b>11.3</b>	<u>Storing Data Using Dictionaries</u>	208
<b>11.4</b>	<u>Inverting a Dictionary</u>	210
<b>11.5</b>	<u>Using the In Operator on Tuples, Sets, and Dictionaries</u>	217
<b>11.6</b>	<u>Comparing Collections</u>	218

<b>11.7 A Collection of New Information</b>	218
<b>11.8 Exercises</b>	219
<b>12 Designing Algorithms</b>	229
<b>12.1 Searching for the Smallest Number</b>	231
<b>12.2 Time of the Function</b>	232
<b>12.3 At a Minimum, You Saw This</b>	234
<b>12.4 Exercises</b>	235
<b>13. Searching and Sorting</b>	237
<b>13.1 Searching a List</b>	237
<b>13.2 Binary Search</b>	245
<b>13.3 Sampling</b>	248
<b>13.4 More Efficient Searching Algorithms</b>	250
<b>13.5 MergeSort: A Faster Sorting Algorithm</b>	261
<b>13.6 Sorting Fun When You Learned</b>	265
<b>13.7 Exercises</b>	266
<b>14 Object-Oriented Programming</b>	269
<b>14.1 Understanding a Problem Domain</b>	270
<b>14.2 Function “Instances,” Class Object, and Stack-Object</b>	271
<b>14.3 Writing a Method in Class Bank</b>	274
<b>14.4 Digging Into Python Syntax: More Special Methods</b>	280
<b>14.5 A Little Bit of GOF Theory</b>	289
<b>14.6 Case Study: Milestones, Albums, and Photo Files</b>	290
<b>14.7 Classifying What You’ve Learned</b>	292
<b>14.8 Exercises</b>	293
<b>15 Testing and Debugging</b>	297
<b>15.1 Why Do You Need to Test?</b>	297
<b>15.2 Case Study: Testing above freezing</b>	308
<b>15.3 Case Study: Testing <code>minify.com</code></b>	310
<b>15.4 Debugging the Errors</b>	310
<b>15.5 Hunting Bugs</b>	311
<b>15.6 Things Write Down in Your Kep</b>	312
<b>15.7 Exercises</b>	312
<b>16 Creating Graphical User Interfaces</b>	317
<b>16.1 Using Module Tkinter</b>	317
<b>16.2 Building a Tkinter GUI</b>	318
<b>16.3 Models, Views, and Controllers, Oh My!</b>	323
<b>16.4 Customizing the Visual Style</b>	327

<b>18.3 Introducing a Few More Widgets</b>	332
<b>18.4 Object-Oriented GUIs</b>	336
<b>18.5 Keeping the Connection from Dying a GUIT Werk</b>	336
<b>18.6 Exercises</b>	338
<b>17. Databases</b>	339
<b>17.1 Overview</b>	339
<b>17.2 Creating and Populating</b>	340
<b>17.3 Deleting Data</b>	344
<b>17.4 Updating and Deleting</b>	347
<b>17.5 Using NULL for Missing Data</b>	348
<b>17.6 Using <i>where</i> in Compound Queries</b>	349
<b>17.7 Keys and Constraints</b>	353
<b>17.8 Advanced Features</b>	354
<b>17.9 Some Data Based On What You Learned</b>	360
<b>17.10 Exercises</b>	361
<b>Bibliography</b>	365
<b>Index</b>	367

# Acknowledgments

This book would not be continuing and finished with errors if it weren't for a bunch of awesome people who patiently and carefully read our drafts.

We had a great team of people provide technical review; in no particular order, Sean Bellman, Julian Carter, Chris Nelson, Auburn University Professor, C. Phillip Fox, Michael Gammie, Derek Gray, Peter Bevan, Fabian Breider, Paul Holbrook, Eric Lechler, Mike Riley, Sean Schickle, Tim Ottobre, Bill Parfrey, Dan Chapman, and Justin Stanley. We also appreciate all the people who reported errors as we were turning each later phase of our feedback was invaluable.

Greg Williams started us on this journey when he proposed that we write a textbook and he was our guide and mentor as we worked together to create the first edition of this book.

Asel and Irememor thank our patient editor, Lynn Taughton, who never gave up on us even though we thoroughly deserved it many times. Lynn, we can't imagine having anyone better giving us feedback and guidance. Thank you so much.

# Preface

This book uses the Python programming language to teach introductory computer science topics and a handful of useful applications. You'll certainly learn a fair amount of Python as you work through this book, but along the way you'll also learn about topics that every programmer needs to know: ways to approach a problem and break it down into parts, how and why to document your code, how to test your code to help ensure your program does what you want it to, and more.

We chose Python for several reasons:

- It is free and well documented. In fact, Python is one of the largest and best-supported open source projects today.
- It runs everywhere. The reference implementation, written in C, is used everywhere from cell phones to supercomputers, and it is supported by professional-quality installers for Windows, Mac OS X, and Linux.
- It has a clear syntax. Yes, every language makes this claim, but during the several years that we have been using it at the University of Toronto, we have found that students make noticeably fewer ‘punctuation’ mistakes with Python than with C-like languages.
- It is relevant. Thousands of companies use it every day. It is one of the languages used in Google, Industrial Light & Magic, Pixar, extensively, and large portions of the game EVE Online are written in Python. It is also widely used by academic research groups.
- It is well supported by tools. Many editors like vim and Emacs all have Python editing modes, and several professional-quality IDEs are available. (We use IDLE, the free development environment that comes with a standard Python installation.)

## Our Approach

We have an “*object first, classes second*” approach: students are shown how to use objects from the standard library early on but do not create their own classes until after they have learned about some control and basic data structures. This allows students to get familiar with what is special in Python; all types, including integers and floating-point numbers, are classes before they have to write their own types.

We have organized the book into two parts. The first covers fundamental programming ideas: elementary data types (numbers, strings, lists, sets, and dictionaries), modules, control flow, functions, testing, debugging, and algorithms. Depending on the audience, this material can be covered in a couple of months.

The second part of the book consists of more or less independent chapters on more advanced topics that assume all the basic material has been covered. The first of these chapters shows students how to create their own classes and introduces encapsulation, inheritance, and polymorphism (useful for computer science majors who will probably track this material). The other chapters cover testing, databases, and GUI construction; these will appeal to both computer science majors and students from the sciences and will allow the book to be used for both.

## Further Reading

Lots of other good books on Python programming exist. Some are available online, such as [Introduction to Computing and Programming in Python: A Multimodal Approach](#), [Pythonds](#) and [Python Programming: An Introduction to Computer Science](#) [2018], others are for those with my previous programming experience ([How to Think Like a Computer Scientist: Learning with Python](#) [2018]), [Object-Oriented Programming in Python](#) (2017) and [Learning Python](#) (2018). You may also want to take a look at [Python Education Special Interest Group \(EDUCSSG\)](#) (#pyr7) the special interest group for education using Python.

## Python Resources

Information about a variety of Python books and other resources is available at [http://readpythonopenminded.org](#).

Learning a second programming language can be challenging. There are many possibilities, such as well-known languages like C, Java, C#, and Ruby. Python is similar in concept to those languages. However, you will likely learn

more and become a better programmer if you learn a programming language that requires a different mindset, such as Racket,<sup>1</sup> Eiffel,<sup>2</sup> or Haskell.<sup>3</sup> In any case, we strongly recommend learning a second programming language.

## What You'll See

In this book, we'll do the following:

- We'll show you how to develop and run programs that solve real-world problems. Most of the examples will come from science and engineering, but the ideas can be applied to any domain.
- We'll start by teaching you the core features of Python. These features are included in every modern programming language, so you can use what you learn in Python when you switch to another.
- We'll also teach you how to think mathematically about programming. In particular, we'll show you how to break complex problems into simple ones and how to combine the solutions to those simpler problems to create complex applications.
- Finally, we'll introduce frameworks that will help make your programming more productive, as well as some others that will help your applications cope with larger problems.

## Online Resources

All the `answer`, `extra`, `discrepancy`, `errata`, `translation`, `mathematica`, and `recursion` sections are available at <http://pressbooks.saylorproject.org/python/>.

1. <https://racket-lang.org/>

2. <https://eiffel.org/>

3. <https://haskell.org/>

## CHAPTER 1

# What's Programming?



Photo credit: NASA/JPL-Caltech image from the Center for Remote Sensing Visualization and Study

Take a look at the pictures above. The first one shows forest cover in the Amazon basin in 1975. The second one shows the same area twenty-five years later. Anyone can see the amount of the rainforest has been destroyed, but how much is "much"?

Now look at this:

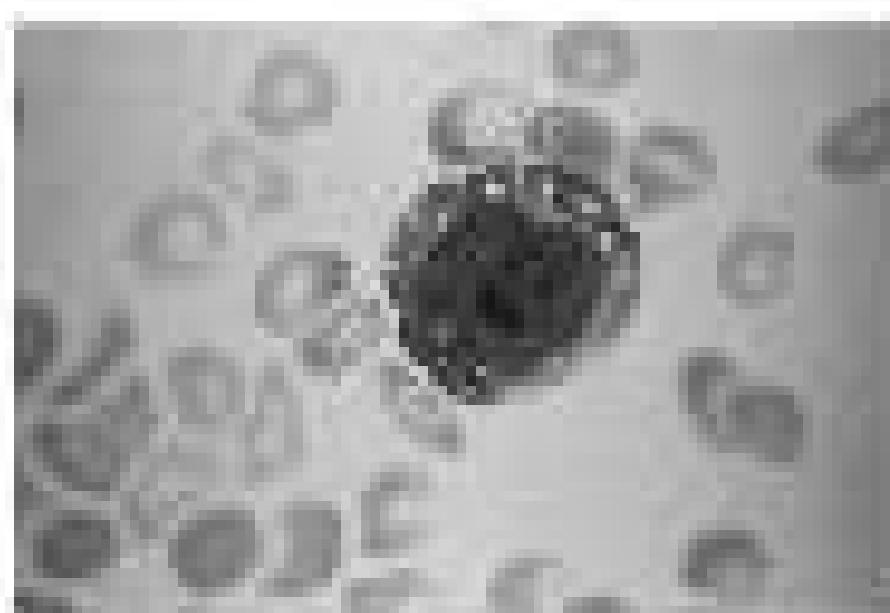


Photo credit: CDC

Are these blood cells healthy? Or any of them show signs of leukemia? It would take an expert doctor a few minutes to tell. Multiply those minutes by the number of people who need to be screened. There simply aren't enough human doctors in the world to check everyone.

This is where computers come in. Computer programs can measure the differences between red globules and count the number of oddly shaped platelets in a blood sample. Geologists use programs to analyze gene sequences; statisticians to analyze the output of that. Astronomers, geologists, to predict the effects of earthquakes; economists, to analyze fluctuations in the stock market; and meteorologists, to study global warming. More and more scientists are writing programs to help them do their work. In fact, these programs are making entirely new kinds of science possible.

Of course, computers are good for a lot more than just science. We used computers to write this book. You probably used one today to chat with friends, find out where your lectures are, or look for a restaurant that serves pizza and Pizzas. And every day, someone figures out how to make a computer do something that has never been done before. Together, these "somethings" are changing the world.

This book will teach you how to make computers do what you want them to do. You may be planning to be a doctor, a linguist, or a scientist rather than a full-time programmer, but whatever you do, being able to program is as important as being able to write a letter or do basic arithmetic.

We begin in this chapter by explaining what programs and programming are. We then define a few terms and present a few useful bits of information for complete beginners.

## 1.1 Programs and Programming

A program is a set of instructions. When you write down directions to your house for a friend, you are writing a program. Your friend "executes" that program by following each instruction in turn.

Every program is written in terms of a few basic operations that the reader already understands. For example, the set of operations that your friend can understand might include the following: "Turn left at Durbin Street," "Go forward three blocks," and "If you get to the gas station, turn around—you've gone east far."

Computers are similar but have a different set of operations. Some operations are mathematical, like "Take the square root of a number," while others track the "location" from the "list memory register" and "Move a pixel blue."

The most important difference between a computer and an old-fashioned calculator is that you can “teach” a computer new operations by defining them in terms of old ones. For example, you can teach the computer that “Take the average” means “Add up the numbers in a sequence and divide by the sequence’s size.” You can then use the operations you have just defined to create still more operations, each layered on top of the ones that came before. It’s sort of like creating life by getting atoms higher in matter, protists and then combining protists to build cells, combining cells to make organs, and combining organs to make a creature.

Writing new operations and combining them to do useful things is the heart and soul of programming. It is also a tremendously powerful way to think about other kinds of problems. As Professor Jeannette Wing wrote in *Computational Thinking [PDF]*, computational thinking is about the following:

- Conceptualizing, not programming. Computer science isn’t “computer programming.” Thinking like a computer scientist means more than being able to program a computer. It requires thinking at multiple levels of abstraction.
- A very few humans, not computers, think. Computational thinking is a way humans solve problems. It isn’t trying to get humans to think like computers. Computers are dull and boring; humans are clever and imaginative. We humans make computers exciting. Equipped with computing devices, we use our imaginations to tackle problems we wouldn’t dare take on. In fact, the scale of computing and built systems is so fantastically limited only by our imaginations.
- Generalizing, reusing, etc. Computational thinking will be a reality when it becomes so integral to human endeavor it disappears as an explicit philosophy.

We hope that by the time you have finished reading this book, you will see the world in a slightly different way.

## 1.2 What's a Programming Language?

Instructions to the nearest bus station can be given in English, Portuguese, Mandarin, Hindi, and many other languages. So long as the people you’re talking to understand the language, they’ll get to the bus station.

In the same way, there are many programming languages, and they all can add numbers, read information from files and make user interfaces with windows and buttons and scroll bars. The instructions look different, but

they accomplish the same task. For example, in the Python programming language, here's how you add 3 and 4:

```
3 + 4
```

But here's how it's done in the Scheme programming language:

```
(+ 3 4)
```

They both express the same idea—but they look kind of different.

Every programming language has a way to write mathematical expressions. Repeat a list of instructions lots of numbers at once, choose which of two instructions to do based on the current information you have, and much more. In this book, you'll learn how to do these things in the Python programming language. Once you understand Python, learning for new programming languages will be much easier.

## 1.3 What's a Bug?

Pretty much everyone has had a program crash. A standard story is that you were trying to print a paper when, all of a sudden, your word processor crashed. You had forgotten to save, and you had to start all over again. Old versions of Microsoft Windows used to crash more often than they should have, showing the dreaded “Blue screen of death.” Happily, they’ve gotten a lot better in the past several years. Usually, your computer shows some kind of cryptic error message when a program crashes.

What happened in each case is that the people who wrote the program told the computer to do something it couldn’t do: open a file that didn’t exist, perhaps, or keep track of more information than the computer could handle, or maybe repeat a task with no way of stopping other than by restarting the computer. (Programmers don’t mean to make these kinds of mistakes, but they are very hard to avoid.)

Worse, some bugs don’t cause a crash; instead, they give incorrect information. (This is worse because at least with a crash you’ll notice the there’s a problem.) As a real-life example of this kind of bug, the calendar program kept one of the authors’ mom’s birthday entry for a friend who was born in 1879. That friend, according to the calendar program, had his 1879th birthday this past February. It’s embarrassing, but it can also be embarrassingly frustrating. Every year of silliness like you can only have ways in it. Even if you’re a programmer, it’s important to remember the number of bugs out there: four billion. In order to find a bug, you need to track down where you gave the wrong instructions, then you need to figure out the right instructions, and then you

need to update the program without introducing other bugs. That is a hard, hard task that requires a lot of planning and care.

Every time you get a software update for a program, it is for one of two reasons: new features were added to a program or bugs were fixed. It is always a kind of economics for the software company: are there few enough bugs, and are they minor enough or infrequent enough in order for people to pay for the software?

In this book, we'll show you some fundamental techniques for finding and fixing bugs and also show you how to prevent them from first place.

## 1.4 The Difference Between Brackets, Braces, and Parentheses

One of the pieces of terminology that causes confusion is what to call certain characters. The Python style guide (and several dictionaries) use these names, so this table lists them:

- ( ) Parentheses
- { } Brackets
- { } Braces [Some people call these curly brackets or curly braces, while others will stick to just braces.]

## 1.5 Installing Python

Installation instructions and how to the IDLE programming environment are available on the book's website (<http://pythoncrazypatterns.com/installing/>).

# Hello, Python

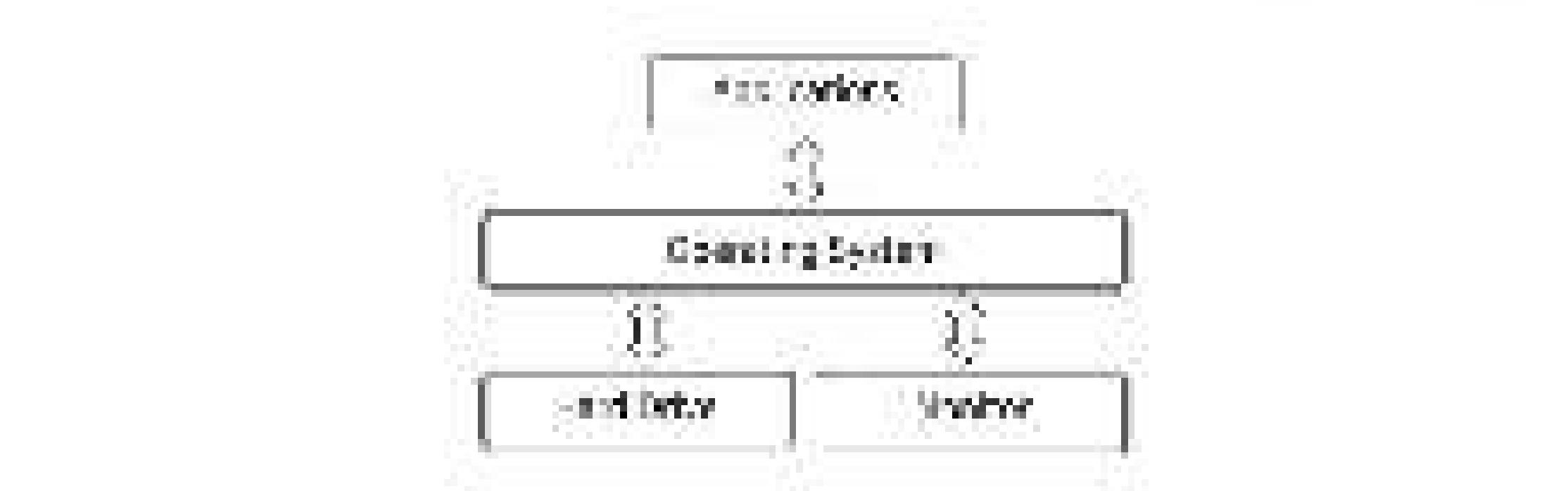
Programs are made up of commands that tell the computer what to do. These commands are called **statements**, which the computer executes. This chapter describes the simplest of Python's statements and shows how they can be used to do arithmetic, which is one of the most common tasks for computers and also a great place to start learning to program. We start the book in section [Getting Started](#) that follows.

## 2.1 How Does a Computer Run a Python Program?

In order to understand what happens when you're programming, you need to know a basic understanding of how a computer executes a program. The computer is assembled from pieces of hardware, including a processor that can execute instructions such as arithmetic, a place to store data such as a hard drive, and various other pieces, such as a monitor or monitor, a keyboard, a card for connecting to a network, and so on.

To deal with all these pieces, every computer runs some kind of operating system such as Mac OS, Windows, Linux, or Mac OS X. An operating system, or OS, is a program that makes it special to that it's the only program on the computer that's allowed direct access to the hardware. When any other application (such as your browser, a spreadsheet program, or a game) wants to draw on the screen, find out what key was just pressed on the keyboard, or fetch data from the hard drive, it sends a request to the OS (see [Figure 1: Talking to the operating system](#), on page 3).

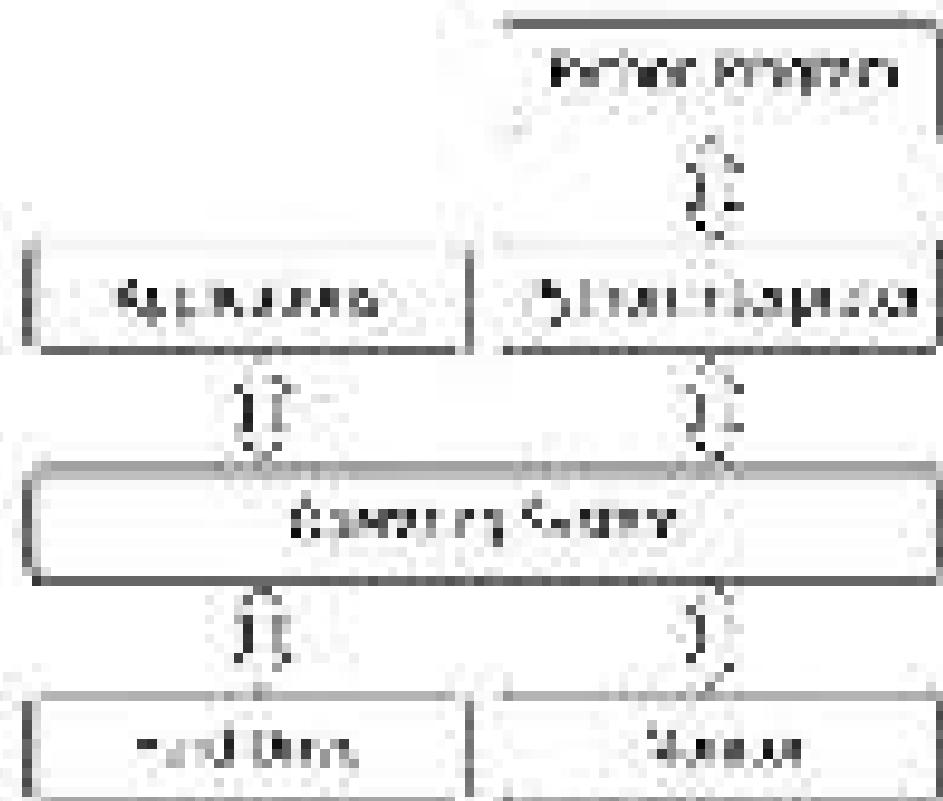
This may seem like a redundant way of doing things, but it means that only the people writing the OS have to worry about the differences between one graphics card and another and whether the computer is connected to a network through ethernet or wireless. The rest of us—everyone analyzing scientific data or creating 3D virtual chat rooms—only have to learn our way



**Figure 1—Talking to the operating system**

around the OS, and our programs will thus run on thousands of different kinds of hardware.

Twenty-five years ago, that's how most programs worked. Today, though, it's common to add another layer between the program and the computer's hardware. When you write a program in Python, Java, or Visual Basic, it doesn't run directly on top of the OS. Instead, another program, called an interpreter or virtual machine, takes your program and translates it for you, translating your commands into language the OS understands. It's a lot easier, more secure, and more portable across operating systems than writing programs directly on top of the OS.



There are two ways to use this Python interpreter. One is to save it to become a Python program that is saved as a file with a .py extension. Another is to interact with it via a program called a *shell*, where you type statements one at a time. The interpreter will execute each statement when you type it, do what the statements says to do, and show any output as text all in one window. We will explore Python in this chapter using a Python shell.

## Install Python Now! If You Haven't Already!

If you haven't yet installed Python 3, please do so now. Python 2 won't die out any significant difference between Python 2 and Python 3, and this book uses Python 3. Local installations instructions in this book's website help you get up-and-running quickly.

For your enjoyment, you can also download a copy of our book by visiting the book's website. You should learn how to play with it just by reading a book on how to program.

Python comes with a program called IDLE, which we use to write Python programs. IDLE has a Python shell, and commands will use Python instead of the usual `sh` when you want to read and run programs that have saved in a file.

We strongly recommend that you open IDLE and follow along with the examples. Typing in the code in this book is the programming equivalent of reading a novel or listening to music; you're better off speaking a new language.

## 2.2 Expressions and Values: Arithmetic in Python

You're familiar with mathematical expressions like  $2 + 4$  ("three plus four") and  $2 \cdot 3 / 5$  ("two minus three divided by five"). Each expression is made out of values like 2, 4, and 5 and operators like + and  $\cdot$ , which combine their components in different ways. In the expression  $4 \cdot 5$ , the operator is  $\cdot$ , and the operands are 4 and 5.

Expressions don't have to involve an operator: a number by itself is an expression. For example, we consider 212 to be an expression as well as a value.

In the very programming language Python, you evaluate basic mathematical expressions. For example, the following expression adds 4 and 13:

```
>>> 4 + 13
17
```

The >>> symbol denotes a prompt. When you opened IDLE, a window should have opened with this symbol shown; you don't type it. It is prompted you to type something. Here we typed  $4 + 13$ , and then we pressed the Return or Enter key in order to signal that we were done entering that expression. Python then evaluated the expression.

When an expression is evaluated, it produces a single value. In the previous expression, the evaluation of  $4 + 13$  produced the value 17. When I type this in the shell, Python shows the value that is produced.

Subtraction and multiplication are similarly straightforward:

```
>>> 15 - 3
12
>>> 4 * 3
12
```

The following expression divides 5 by 2:

```
>>> 5 / 2
2.5
```

This result has a decimal point. In fact, the result of division always has a decimal point even if the result is a whole number:

```
>>> 8 / 3
2.666666666666667
```

## Types

Every value in Python has a particular type, and the types of values determine how they behave when they're combined. Values like 7 and 17 have type int (short for integer), and values like 2.5 and 17.0 have type float. The word float is short for floating point, which refers to the decimal point that moves around between digits of the number.

An expression involving two float numbers will:

```
>>> 17.0 + 18.0
35.0
```

When an expression's operands are an int and a float, Python automatically converts the int to a float. That's why the following two expressions both return the same answer:

```
>>> 17.0 + 18
35.0
>>> 17 + 18.0
35.0
```

If you want, you can omit the zero after the decimal point when writing a floating-point number:

```
>>> 17 + 18.
35.0
>>> 17. + 18
35.0
```

However, most people think this is bad style, since it makes your programs harder to read. It's very easy to miss a digit when you're reading "17" instead of "17.",

## Integer Division, Modulo, and Exponentiation

Every now and then, we want only the integer part of a division result. For example, we might want to know how many 24-hour days there are in 181 hours (which is two 24-hour days plus another 5 hours). To calculate the number of days, we can use integer division:

```
>>> 181 // 24
7
```

We can find out how many hours are left over using the modulo operator, which gives the remainder of the division:

```
>>> 181 % 24
5
```

Python doesn't round the result of integer division. Instead, it takes the *floor* of the result of the division, which means that it rounds down to the nearest integer:

```
>>> -17 // 10
-2
```

The convention when using `%` and `//` with negative numbers: Because Python takes the floor of the result of an integer division, the result is one smaller than you might expect if the result is negative.

```
>>> -17 % 10
7
```

When using modulo, the sign of the result matches the sign of the divisor (i.e., second operand):

```
>>> -17 % 10
7
>>> 17 % -10
-3
```

For the mathematically inclined, the relationship between `%` and `//` comes from this equation: for any two numbers `a` and `b`:

$$(b // a) * b + a % b \text{ is equal to } a$$

For example, because  $-17 // 10$  is  $-2$ , and  $-17 \% 10$  is  $7$ , then  $-17 // -10 * -10 + -17 \% 10$  is the same as  $10 * -2 + 7$ , which is  $-17$ .

floating-point numbers can be equivalent to `%` as well. With `y`, the result is rounded down to the nearest whole number, although the type is a floating-point number:

```
>>> 2.5 // 1
2.0
>>> 3 // 1.0
3.0
>>> 3 // 1.1
2.0
>>> 3.5 // 1.1
2.0
>>> 3.5 // 1.0
2.0
```

The following expression calculates 3 raised to the 0.0 power:

```
>>> 3 ** 0
1.0
```

Operations that take two operands are called binary operators. Negation is a unary operator because it applies to one operand.

```
>>> 5
5
>>> -5
-5
>>> -5.0
-5.0
```

## 2.3 What Is a Type?

We've now seen two types of numbers (integers and floating-point numbers), so we might be asking what we mean by a type. In computing, a type consists of two things:

- a set of values, and
- a set of operations that can be applied to those values.

For example, in type `int`, the values are ..., -3, -2, -1, 0, 1, 2, 3, ... and we have seen that these operators can be applied to these values: `+`, `-`, `*`, `/`, `%`, and `**`.

The values in type `float` are a subset of the real numbers, and it happens that the same set of operations can be applied to these values. If an operator can be applied to more than one type of value, it is called an over-loaded operator. We can see what happens when these are applied to various values in [Table 1, Arithmetic Operators](#) (on page 13).

### Floating Precision

Floating-point numbers are not exactly the fractions you learned in grade school. For example, look at Python's version of the fractions  $\frac{1}{3}$  and  $\frac{2}{3}$ :

Sybol	Operator	Example	Result
-	Negation	7	-5
+	Addition	12 + 31	43
-	Subtraction	5 - 10	-5
*	Multiplication	6.5 * 4	26.0
/	Division	12 / 2	6.0
//	Integer Division	12 // 2	6
%	Remainder	6.5 % 3.5	0.5
**	Exponentiation	2 ** 3	8.0

Table 1—Arithmetic Operators

```
ans = 2 / 3
0.6666666666666667
ans = 5 / 3
1.666666666666667
```

The first value ends with a  $\bar{6}$ , and the second with a  $\bar{7}$ . This is likely both of them should have an infinite number of digits after the decimal point. The problem is that computers have a finite amount of memory, and the calculations time and memory efficiency most programming languages limit how much information can be stored for any single number. The number 0.6666666666666667 can be the closest value to  $\frac{2}{3}$ , but the computer can actually store in that limited amount of memory, and 0.6666666666667 is as close as we get to the real value of  $\frac{2}{3}$ .

## Operator Precedence

Let's put our knowledge of integers back to use by considering operator precedence. To do this, we calculate  $32 \times \pi^2 + 100$  from the expression in Python. It will then multiply by  $\frac{1}{10}$ :

```
(32 * pi**2 + 100) / 10
191.39999999999998
```

Python claims the result is 191.39999999999998 degrees Celsius, when in fact it should be 190. The problem is that multiplication and division have higher precedence than subtraction; in other words, when an expression contains a mix of operations, the  $*$  and  $/$  are evaluated before the  $-$  and  $=$ . This means that, when we actually calculate  $32 \times \pi^2 + 100$ , the subtraction in  $100 - \frac{1}{10}$  is evaluated before the division is applied, and that division is evaluated before the subtraction occurs.

## More on Numeric Precision

Integer values of type int in Python can have up to 231 bits (more, but values are only representable as real numbers). For example,  $\frac{1}{3}$  can't be stored exactly, but we can already know  $\frac{1}{3}$  exactly. Using float accuracy would solve the problem, though it will make the approximation closer to the next value, just as adding enough numbers of 6s after the decimal - decimal value is exactly equal to  $\frac{1}{3}$ .

The difference between  $\frac{1}{3}$  and 0.3333333333333333 may look tiny, but if you use it in another calculation, then the error may get compounded. For example, if we add 1 to  $\frac{1}{3}$ , the resulting number is at .3333, so imagine programming language  $(1 + \frac{1}{3}) - \frac{1}{3}$  is not equal to  $\frac{1}{3}$ :

```
>>> a = 3.0/9
>>> print(str(a))
0.3333333333333333
>>> b = 1.0 + a
>>> print(str(b))
```

As we do more calculations, the rounding errors can get larger and larger, potentially causing mixing very large and very small numbers. For example, compare 1000000000000000 (one billion) and 0.000000001 (there are 19 zeros after the decimal point):

```
>>> print(str(1000000000000000))
1000000000000000.0
```

The result is not a very good comparison between the two numbers, that's because it's only for the computer to care. In fact, you might see millions of an addition never took place. Also, big or small numbers at a time can interfere from each other at all, which is not what a bank wants when it takes up the values of its customers' savings accounts.

It's important to be aware of the floating point issue. It can be very difficult to avoid, because computers are limited in both memory and speed. Nevertheless, the study of algorithms and programmatical errors is the first step of being a skilled programmer and mathematician.

These simple guidelines should help the programmers to add them into practice in larger programs to minimize the error.

We can alter the order of operations by putting parentheses around subexpressions:

```
>>> (123 + 321 * 5) / 9
100.0
```

**Table 2: Arithmetic Operators Diction by Precedence from Highest to Lowest, on page 115** shows the order of precedence for arithmetic operators.

Operations with higher precedence are applied before those with lower precedence. There is an example that shows this:

```

>>> -2 * 4
-8
>>> -(2 * 4)
-8
>>> (-2) * 4
-8

```

Because exponentiation has higher precedence than negation, the subexpression `-2` is evaluated before negation is applied.

Precedence	Operator	Description
High	<code>*</code>	Exponentiation
	<code>-</code>	Negation
	<code>*, /, %, //</code>	Multiplication, division, integer division, and remainder
Lowest	<code>+, -</code>	Addition and subtraction

Table 2—Arithmetic Operators Listed by Precedence from Highest to Lowest

Operators on the same row have equal precedence and are applied left to right, except for exponentiation, which is applied right to left. So, for example, because binary operators `+` and `*` are at the same level, `3 * 4 + 5` is equivalent to `(3 * 4) + 5`, and `3 - 4 + 5` is equivalent to `3 - (4 + 5)`.

It's a good rule to put parentheses in arithmetic expressions even when you don't need to, since it helps the eye read things like `1 + 1.0 + 2.2 * 4.0 / 10.0`. On the other hand, it's a good rule to not use parentheses in simple expressions such as `3 * 4 + 5`.

## 2.4 Variables and Computer Memory: Remembering Values

Like mathematicians, programmers frequently name values so that they can refer them later. A name that refers to a value is called a variable. In Python, variable names can use letters, digits, and the underscore symbol (`_`; they can't start with a digit). For example, `x`, `apple`, `apple1`, and `dog_paws` are all allowed. (But `777` isn't [it would be confused with a number], and neither is `array` [it contains punctuation].)

You create a new variable by assigning it a value:

```
>>> degrees_celsius = 26.9
```

This statement is called an assignment statement; we see that `degrees_celsius` is assigned the value `26.9`. That makes `degrees_celsius` refer to the value `26.9`. We can use variables anywhere we can use values. Whenever Python sees a variable in an expression, it substitutes the value to which the variable refers.

```

>>> degrees_celsius = 21.0
>>> degrees_celsius
21.0
>>> h = 5 * degrees_celsius + 30
30.0
>>> degrees_celsius / degrees_celsius
1.0

```

Variables are called variables because their values can vary as the program executes. We can assign a new value to a variable:

```

>>> degrees_celsius = 21.0
>>> h = 5 * degrees_celsius + 30
30.0
>>> degrees_celsius = 0.0
>>> h = 5 * degrees_celsius + 30
30.0

```

Assigning a value to a variable that already exists does not create a second variable. Instead, the existing variable is reused, which means that the variable no longer refers to the old value.

We can create other variables. For example, calculate the difference between the boiling point of water and the temperature stored in `degrees_celsius`:

```

>>> degrees_celsius = 21.0
>>> difference = 100 - degrees_celsius
>>> difference
79.0

```

### Warning: = Is Not Equality In Python!

The `=` operator, — means “the thing on the left is equal to the thing on the right.” In Python, it means something quite different. For example, `1 == 12` evaluates to `False` (two variables). But `12 == 12` evaluates to `True`. Because of this, we never describe the expression `1 == 12` as “1 equals 12.” Instead, we say “12 is equal to 12.”

## Values, Variables, and Computer Memory

We’re going to develop a model of computer memory—a memory model—that will let us trace what happens when Python executes a Python program. This memory model will help us accurately predict and explain what Python does when it executes code, a skill that is important for becoming a good programmer.

Every location in the computer’s memory has a memory address, much like an address for a house on a street, that uniquely identifies that location.

## The Online Python Tutor

Philip Guo wrote a web-based memory simulator that matches our memory model pretty well. Here's the URL: <http://www.csail.mit.edu/~phg/PyTutor/>. Python 2 and Python 3 code runs just fine (you pick the correct version). The web app has tabs. Check them out for more detailed coverage.

- Edit Boxes: Insert functions
- Revert all objects on the heap
- Click on memory pointer pointers
- Use watch lists for references

We strongly recommend that you use this visualizer whenever you want to inspect execution of a Python program.

Because you find it frustrating, we suggest trying Python's `__repr__` when you do not understand our memory model (and, after all, that's exactly what it does).

We're going to mark our memory addresses with `0x1000` (short for `memory[0]`) but note that they look different from integers: `0x10`, `0x20`, and `0x100`.

Here is how we draw the floating point value `25.0` using the memory model:



This picture shows the value `25.0` at the memory address `0x1000`. We will always show the type of the value as well—in this case, `float`. We will call this box an object: a value at a memory address with a type. During execution of a program, every value that Python keeps track of is stored inside an object in computer memory.

In our memory model, a variable contains the memory address of the object to which it refers:



In order to make the picture easier to interpret, we will usually draw arrows from variables to their objects.

We use the following terminology:

- Value `25.0` has the memory address `12`.
- The object at the memory address `12` has type `float` and the value `25.0`.
- Variable `degrees_celsius` contains the memory address `12`.
- Variable `degrees_celsius` refers to the value `25.0`.

Whenever Python needs to know which value `degrees_celsius` refers to, it looks at the object at the memory address that `degrees_celsius` contains. In this example, that memory address is `12`, so Python will use the value at the memory address `12`, which is `25.0`.

## Assignment Statement

Here is the general form of an assignment statement:

```
variable = expression
```

This is executed as follows:

1. Evaluate the expression on the right of the `=` sign to produce a value. This value has a memory address.
2. Store the memory address of the value in the variable on the left of the `=`. If no such variable (i.e. that name doesn't already exist); otherwise, now reuse the existing variable, replacing the memory address that it contains.

Consider this example:

```
nn. degrees_celsius = 25.0 + 5
nn. degrees_celsius
30.0
```

Notice how Python executes the statement `degrees_celsius = 25.0 + 5`:

1. Evaluate the expression on the right of the `=` sign: `25.0 + 5`. This produces the value `30.0`, which has a memory address. (Remember that Python stores all values in computer memory.)
2. Make the variable on the left of the `=` sign, `degrees_celsius`, refer to `30.0` by storing the memory address of `30.0` in `degrees_celsius`.

## Reassigning to Variables

Consider this code:

```
nn. difference = 10
nn. double = x * difference
nn. double
40
nn. difference = 5
nn. double
40
nn.
```

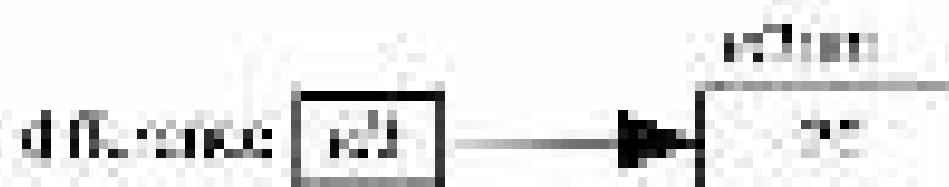
This code demonstrates that modifying to a variable does not change any other variable. We start by assigning value 42 to variable difference, and then we assign the result of multiplying 2 variables which produces 40 to variable double.

Next, we assign value 5 to variable difference. But when we examine the value of double, it still refers to 42.

Here's how it works according to our rules. The first statement, `difference = 42`, is executed as follows:

1. Evaluate the expression on the right of the = sign: `4 * 10`. As this evaluates to the value 40, which we'll put at memory address #10.
2. Make the variable on the left of the = sign, difference, refer to 40 by storing #10 in difference.

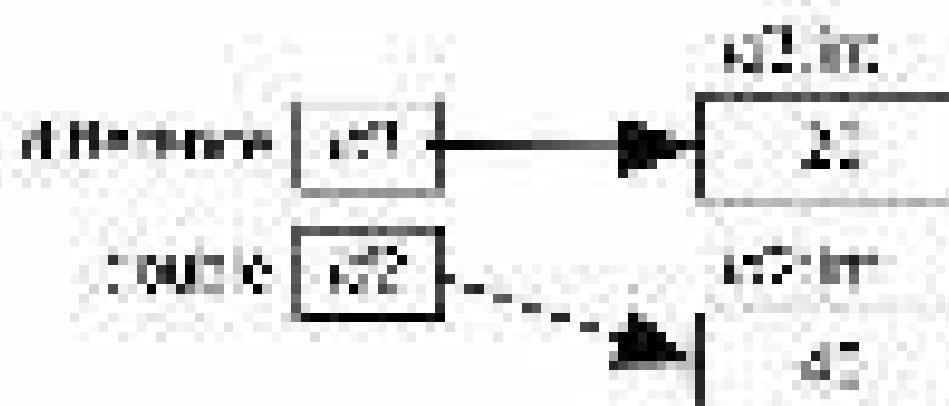
Here is the current state of the memory model. Variable `double` has not yet been created because we have not yet executed the assignment to it.



The second statement, `double = 2 * difference` is executed as follows:

1. Evaluate the expression on the right of the = sign: `2 * difference`. As we see in the memory model, difference refers to the value 40, so this expression is equivalent to `2 * 40`, which produces 80. We'll put the memory address #20 for the value 80.
2. Make the variable on the left of the = sign, double, refer to 40 by storing #20 in double.

Here is the current state of the memory model:

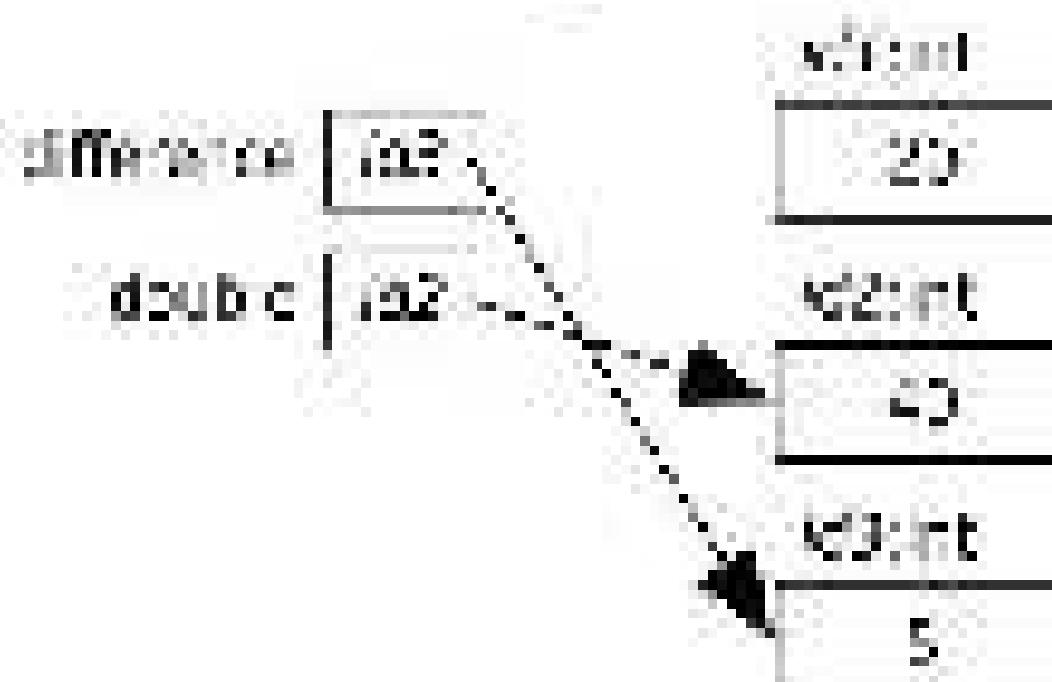


When Python executes the third statement, `double = 5`, it merely looks up the value the variable has (40) and replaces it.

The fourth statement, `difference = 5`, is executed as follows:

- Evaluate the expression on the right of the = sign: `s`. This produces the value `5`, which we'll put at the memory address `s`.
- Make the variable on the left of the = sign, `d` because refer to `s` by starting `at` in otherwise.

Here is the current state of the memory model:



Variable `double` still contains `20`, but it still refers to `20`. Neither variable refers to `20` anymore.

The third and last assignment statement, `double` takes the value that `double` refers to, which is still `20`, and displays it.

We can even use a variable on both sides of an assignment statement:

```

>>> number = 3
>>> number
3
>>> number = 2 * number
>>> number
6
>>> number = number * number
>>> number
36
  
```

We'll now explain how Python executes this code, but we won't explicitly mention memory addresses. Trace this on a piece of paper while we describe what happens; make up your own memory addresses as you go through.

Python executes the first statement, `number = 3`, as follows:

- Evaluate the expression on the right of the = sign: `3`. This causes `3` to evaluate. `3` is produced.
- Make the variable on the left of the = sign, `number` refer to `3`.

Python executes the second statement, `number = 2 * number`, as follows:

- Evaluate the expression on the right of the = sign: `2 * number`. `number` currently refers to `3`, so this is equivalent to `2 * 3`, and is produced.

- Make the variable on the left of the = sign, `t`, refer to `t`.

Python executes the third statement, `t = t * number`, as follows:

- Evaluate the expression on the right of the = sign: `t = a * b * number`. This statement refers to `a`, so this is equivalent to `t * b * number`, which produces:
- Make the variable on the left of the = sign, `t`, now refer to `t`.

## Augmented Assignment

In this example, the variable `score` appears on both sides of the assignment statement:

```
score = 50
score
50
score = score + 20
score
70
```

This is an instance that Python provides a shorthand mechanism for this operation:

```
score = 50
score
50
score += 20
score
70
```

An augmented assignment combines an assignment statement with an operator to make the statement more concise. An augmented assignment statement is evaluated as follows:

- Evaluate the expression on the right of the = sign to produce a value.
- Apply the operator attached to the = sign to the variable on the left of the = and the value that was produced. This produces another value. Store the memory address of that value in the variable on the left of the =.

Note that the operator is applied after the evaluation on the right is evaluated:

```
score = 2
score *= 3
score
6
```

All the operators you’ve learned in [Table 2: Arithmetic Operators](#) listed by precedence from highest to lowest on page 10 have also found [examples](#). For example, we can square a number by multiplying it by itself:

```
>>> number = 10
>>> number *= number
>>> number
100
```

This code is equivalent to this:

```
>>> number = 10
>>> number = number * number
>>> number
100
```

[Table 3: Augmented Assignment Operators](#), contains a summary of the augmented operators you’ve learned so far, how many bytes it will take to use them, the operators you learned about in [Section 2.2, Expressions with Values, Arithmetic, and Python](#) on page 8.

Symbol	Example	Result
<code>+=</code>	<code>x = 5 x += 2</code>	<code>x refers to 9</code>
<code>-=</code>	<code>y = 10 y -= 2</code>	<code>y refers to 8</code>
<code>*=</code>	<code>a = 5 a *= 2</code>	<code>a refers to 10</code>
<code>/=</code>	<code>x = 10 x /= 2</code>	<code>x refers to 5.0</code>
<code>//=</code>	<code>y = 10 y //= 2</code>	<code>y refers to 5</code>
<code>%=</code>	<code>r = 10 r %= 2</code>	<code>r refers to 1</code>
<code>**=</code>	<code>s = 10 s **= 2</code>	<code>s refers to 100</code>

**Table 3—Augmented Assignment Operators**

## 2.5 How Python Tells You Something Went Wrong

Broadly speaking, there are two kinds of errors in Python: [syntax errors](#), which happen when you type something that isn’t valid Python code, and

another’s syntax, which happens when you tell Python to do something that it just can’t do, like divide a number by zero or try to use a variable that doesn’t exist.

Here is what happens when I try to use a variable that hasn’t been defined yet:

```
❷>>> print x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

This is pretty cryptic; Python error messages aren’t meant for people who already know Python. (You’ll get used to them and soon find them helpful.) The first two lines aren’t much use right now, though they’ll be indispensable when we start writing longer programs. The last line is the one that tells us what went wrong: the name `x` was undefined.

There’s another error message you might sometimes see:

```
❸>>> print x +
      ^          .line 1
      ^          ^
SyntaxError: invalid syntax
```

The rules governing what is and isn’t legal in a programming language are called its **syntax**. The message tells us that we violated Python’s syntax rules—in this case, by asking it to add something to `x`, but not telling it what to add.

Earlier, in “Warning: Is Now Equal to Python!” on page 16, we claimed that `17 = 2` results in an error. Let’s try it:

```
❹>>> 17 = 2
      ^          .line 1
SyntaxError: can't assign to literal
```

A literal is any value, like `12` and `2.5`. This is a **spike**, or because when Python executes that assignment statement, it knows that you can’t assign a value to a number you’ve already taken: so even though you can’t change the value of `17` to anything else, `17 = 2` isn’t allowed.

## 2.6 A Single Statement That Spans Multiple Lines

Sometimes statements get pretty tall. The recommended Python style is to limit lines to 80 characters, including spaces, tabs, and other whitespace characters, and that’s a common limit throughout the programming world.

Here's what to do when faced with too long of a line you want to split it up for clarity:

In order to split up a statement into more than one line, you need to do one of two things:

1. Make sure your line break occurs inside parentheses, or
2. Use the line continuation character, which is a backslash (\).

Note that the line continuation character is a backslash (\) and the division symbol (/)

Here are examples of both:

```
>>> (2 +  
... 3)  
5  
>>> 2 + 3  
... 3  
5
```

Notice how we don't get a SyntaxError. Each triple-dot prompt in our examples indicates that we are in the middle of entering an expression; we use them to make the code look up nicely. You do not have the idea any later than you type the greater-than signs to the usual >>> prompt, and if you are using IDLE, you won't see them at all.

Here is a more realistic (and briefer) example: let's say we're baking cookies. The authors live in Canada, which uses Celsius, but we own cookbooks that use Fahrenheit. We are wondering how long it will take to preheat our oven. Here are our facts:

- The room temperature is 10 degrees Celsius.
- Our oven heats up to Celsius, and the heat rises up at 30 degrees per minute.
- Our oven takes 15 minutes, and it takes no longer than 15 minutes to preheat.

We can convert 1 degree Fahrenheit to 1 degree Celsius like this:  $(F - 32) * 5 / 9$ . Let's use this information to try to solve our problem:

```
>>> room_temperature_c = 10  
>>> cooking_temperature_f = 350  
>>> oven_heating_rate_c = 30  
>>> oven_heating_time = (  
... (cooking_temperature_f - 32) * 5 / 9 - room_temperature_c) / 30  
... oven_heating_rate_c  
>>> oven_heating_time  
7.00000000000000
```

No, that just makes right minutes too precious.

The assignment statement to variable `oven_heating_time` spans three lines. The first line ends with an open parenthesis, so we've retained the continuation character. The second ends outside the parenthesis, so we've used the line-continuation character. The third line completes the assignment statement.

That's still hard to read. Once we've eliminated an expression from the line, we can indent the opening durability or by keeping the space from a blank to our heart's content to make it clearer:

```
oven_heating_time = (
    (cooking_temperature_f - 32) * 5 / 9 + room_temperature_f) / 5
    oven_heating_time
```

Or even this—indulge how the two subexpressions involved in the subtraction line up:

```
oven_heating_time = (
    (cooking_temperature_f - 32) * 5 / 9 +
    room_temperature_f) / 5
    oven_heating_time
```

In the previous example, we clarified the expression by working with induction. However, we could have made this process even clearer by converting the cooking temperature to Celsius before calculating the heating time:

```
room_temperature_c = 20
cooking_temperature_f = 320
cooking_temperature_c = (cooking_temperature_f - 32) * 5 / 9
oven_heating_rate_c = 20
oven_heating_time = (cooking_temperature_c - room_temperature_c) / oven_heating_rate_c
oven_heating_time
7.000000000000001
```

The message to take away here is that well-named temperature variables can make code much clearer.

## 2.7 Describing Code

Programs can be quite cryptic and are often thousands of lines long. It can be helpful to write a comment describing parts of the code so that when you, or someone else, reads it they don't have to spend much time figuring out why the code is there.

In Python, any line the `#` character is encountered, Python will ignore the rest of the line. This allows you to write English sentences:

```
# A python program that converts degrees of the f scale to c
```

The `#` symbol does not have to be the first character on the line; it can appear at the end of a statement.

```
print(123 - 32) # 91, 9 = created 22 objects. Estimated to use 200000  
188,8
```

Notice that the comment doesn't describe how Python works. Instead, it is meant for humans reading the code to help them understand why the code exists.

## 2.8 Making Code Readable

Much like there are conventions in English sentence structure to make the words easier to read, we also expect in Python code to work in a similar way. In particular, we always put a space before and after every binary operator. For example, we write `x = 4 + 2.5 * 16` instead of `x = 4+2.5*16`. There are situations where it may not make a difference, but that's a detail we don't want to fuss about, so we always do it if it's almost never harder to read if there are spaces.

Psychologists have discovered that people can keep track of only a handful of things at any one time ([Distracted Thinking That Causes Bad Programming Habits](#)). Since programs can get quite complicated, it's important that you choose names for your variables that will help you remember what they're for—`x2`, `z2`, and `b2` won't remind you of anything when you come back to look at your program next week; use names like `ok123`, `escape`, and `big_guy` instead.

Other studies have shown that your brain automatically notices differences between things—in fact, there's no way to stop it from doing this. As a result, the more inconsistencies there are in a piece of text, the longer it takes to read. (Also think about how long it would TAKE you to read a book if the first few chapters had titles like this.) It's therefore also important to use consistent names for variables. If you call something `rainbow` in one place, don't call it `rain_worm` another; if you use the name `magical`, don't also use the name `magic`, and so on.

These rules are so important that many programming teams require members to follow a style guide for whatever language they're using, just as newspapers and book publishers specify how to capitalize headings and whether to use a colon before the last item in a list. If you search the Internet for `python style guide` (<https://www.google.com/search?q=python+style+guide>), you'll discover links to hundreds of examples. In this book, we follow the style guide for Python from <http://www.python.org/doc/essays/StyleGuide.html>.

You will also discover that lots of people have wasted many hours arguing over what the "best" style for code is. None of your decisions or jerkish ways

most interested in how their opinions about Linux are. If they do, ask them what data they have to back up their belief. Asking questions and strong evidence must be taken seriously.

## 2.9 The Object of This Chapter

In this chapter, you learned the following:

- An operating system is a program that manages your computer's hardware or behalf of other programs. An interpreter or virtual machine is a program that sits atop of the operating system and runs your programs for you. The Python shell is an interpreter, translating your Python statements into language the operating system understands, and translating the results back so you can see and use them.
- Programs are made up of *statements* or *instructions*. These can be simple expressions like `3 + 4` and assignment statements like `x = 21` which create new variables or change the values of existing ones. There are many other kinds of statements in Python, and we'll introduce them throughout the book.
- Every value in Python has a *visible type*, which determines what operations can be applied to it. The two types used to represent numbers are `float` and `int`. Floating-point numbers are approximations to real numbers.
- Python evaluates an expression by applying higher-precedence operators before lower-precedence operators. You can change that order by putting parentheses around subexpressions.
- Python stores every value in computer memory. It memory location containing a value is called an *object*.
- Variables are created by executing assignment statements. If a variable already exists because of a previous assignment statement, Python will use that one instead of creating a new one.
- Variables contain memory addresses of values. We say that variables refer to *values*.
- Variables make the easiest place to store the results of mathematical expressions.

## 2.10 Exercises

Here are some exercises for you to try for your own. Solutions are available at <https://nostarch.com/pythonforbeginners>.

1. For each of the following expressions, what value will the expression have? Verify your answers by typing the expressions into Python.
- 9.7
  - 8 \* 2.5
  - 9 / 2
  - 9 // 2
  - 9 % 2
  - 9 // -2
  - 9 // -2.5
  - 9 % -2
  - (-9) // 2
  - 9 // -2.5
  - 9 + 3 \* 5
2. Unary minus negates a number. Unary plus exists as well. For example, Python understands `-5`. It's just the value `-5`. What do you think `+5` should do? Should it leave the sign of the number alone? Should it act like absolute value, removing any negative? Use the Python shell to find out its behavior.
3. Write two assignment statements that do the following:
- Create a new variable, `temp`, and assign it the value `24`.
  - Convert the value in `temp` from Celsius to Fahrenheit by multiplying by `1.8` and adding `32`; make `ans` refer to the resulting value.  
What is `temp` now?
4. For each of the following expressions, in which order are the subexpressions evaluated?
- $6 * 3 + 7 * 4$
  - $5 + 2 / 4$
  - $5 * 2 / 3 * 4$
5. a. Create a new variable `x`, and assign it the value `10.5`.  
b. Create a new variable `y`, and assign it the value `4`.  
c. Sum `x` and `y`, and make `x` refer to the resulting value. After this statement has been executed, what are `x` and `y`'s values?

6. Write a bullet list description of what happens when Python evaluates the statement `t = x - y` when `x` has the value 2.
7. When a variable to most likely has been assigned a value, a developer executes in the Python shell an interactive expression that results in a `TypeError`.
8. Which of the following expressions results in `TypeError`?
- `6**-2`
  - `s = print`
  - `obj[1][2][0]`
  - `obj&obj`
  - `A = 1 + 2 / 2`

# Designing and Using Functions

Mathematicians create functions to make calculations (such as Fahrenheit to Celsius conversion) easy to reuse and to make future calculations easier to read because they can use those functions instead of repeatedly writing out equations. Programmers do this too, at least as often as mathematicians. In this chapter you will explore several of the built-in functions that come with Python, and we'll also show you how to define your own functions.

## 3.1 Functions That Python Provides

Python comes with many built-in functions that perform common operations. One example is `abs`, which produces the absolute value of a number:

```
>>> abs(-9)
9
>>> abs(3.2)
3.2
```

Each of these statements is a function call.

### Keep Your Shell Open

As a reminder, we recommend that you have IDLE open whenever you're working with code so that you can edit the code, run it, and see the results. It's a good way to learn programming.

The general form of a function call is as follows:

`function_name(argument)`

An argument is an expression that appears between the parentheses of a function call. In the `abs(9)` line, the argument is `9`.

Here, we calculate the difference between a day temperature and a night temperature, as might be seen on a weather report. In our weather system it's cold in overnight:

```
>>> day_temperature = 2
>>> night_temperature = 10
>>> abs(day_temperature - night_temperature)
7
```

In this call on function `abs`, the argument is `day_temperature - night_temperature`. Because `day_temperature` refers to 2 and `night_temperature` refers to 10, Python evaluates this expression to -8. This would be then passed to function `abs`, which then returns, or produces, the value 7.

Here are the rules to executing a function call:

1. Evaluate each argument expression, working from left to right.
2. Pass the resulting values into the function.
3. Execute the function. When the function call finishes, if `return` a value, that value function calls produce values, they can be used in expressions:

```
>>> abs(-7) + abs(3.3)
10.3
```

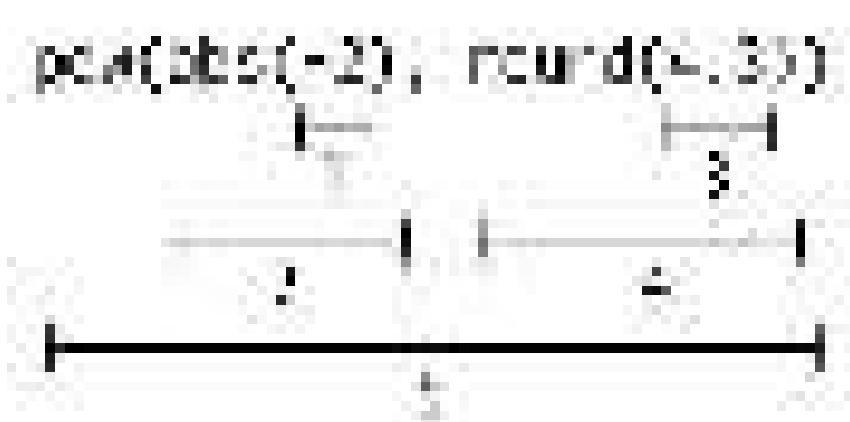
We can also use function calls as arguments to other functions:

```
>>> print(abs(-2), round(4.5))
18
```

Python sees the call on `abs` and wants to evaluate() the arguments from left to right. The first argument is a call on function `abs`, so Python evaluates it,得到 2 (additive 2, or that's the first value for the call on `abs`). Then Python evaluates `round(4)`, which produces 4.

Now that the arguments to the call on function `print` have been evaluated, Python thinks calling `print`, sending in 2 and 4 as the arguments `values`. That means that `print(2)` would be equivalent to `print(2)`, and `X` is 16.

Here is a diagram indicating the order in which the various pieces of this expression are evaluated by Python:



We have underlined each subexpression and given it a number to indicate when Python executes it to evaluate that subexpression.

Some of the most useful built-in functions are ones that convert from one type to another. Type names of variables can be used as function:

```
>>> int(34.6)
34
>>> int(1.4)
1
>>> float(23.1)
23.1
```

In this example, we see that when a floating-point number is converted to an integer, it is truncated, not rounded.

If you’re not sure what a function does, try calling built-in function `help`, which shows documentation for any function:

```
>>> help(abs)
Help on built-in function abs in module builtins:

abs(...)

    abs(number) -> number
```

Return the absolute value of the argument.

The first line states which function is being described and which module it belongs to. Modules are an organizational tool in Python and are discussed in Chapter 8, [A Modular Approach to Program Development](#), on page 125.

The next part describes how to call the function. The arguments are described within the parentheses—here, `number` means that you can call `abs` with either an integer or a float. After that is the `function`, which either is called without a float. After that is an English description of what the function does when it is called.

Another built-in function is `round`, which rounds a floating-point number to the nearest integer:

```
>>> round(0.5)
0
>>> round(2.5)
2
>>> round(2.5)
2
>>> round(-2.5)
-2
>>> round(-2.5)
-2
```

The `round` word can be called with one or two arguments. If called with one, as we have done here, it rounds to the nearest integer. If called with two, it rounds to a指定的位数 number, where the second argument increases the precision:

```
<in> round(2.141592653, 2)
2.14
```

The documentation for `round` indicates that the second argument is optional by surrounding it with brackets:

```
<in> help(round)
Help on built-in function round in module builtins:
```

```
round(...)

round(number[, ndigits]) -> number
```

Round a number to a given precision in decimal digits (default 0 digits). This returns an int when called with one argument; otherwise the same type as the number. ndigits may be negative.

Let's explore built-in function `pow` by starting with its help documentation:

```
<in> help(pow)
Help on built-in function pow in module builtins:
```

```
pow(...)

pow(x, y[, z]) -> number
```

With two arguments, equivalent to `x**y` with three arguments,  
equivalent to `(x**y) % z`, but may be more efficient (e.g., for floats).

This documented function can only be called with either two or three arguments. The English description mentions that when called with two arguments it is equivalent to  $x^y$ . Let's try it:

```
<in> pow(2, 4)
16
```

This call calculates  $2^4$ . So far, so good. How about with three arguments?

```
<in> pow(2, 4, 3)
1
```

We know that  $2^4$  is 16, and evaluation of  $16 \% 3$  produces 1.

## 3.2 Memory Addresses: How Python Keeps Track of Values

In [Data in Values, Variables, and Computer Memory](#) on page 106, you learned that Python keeps track of each value in a separate object and that each object has a memory address. You can discover the actual memory address of an object using built-in function `id`:

```
>>> help(id)
Help on built-in function id in module builtins:

id(...)

    id(object) -> integer

        Return the identity of an object. This is guaranteed to be unique among
        simultaneously existing objects. (Hint: it's the object's memory
        address.)
```

How cool is that? Let's try it:

```
>>> id(42)
4391177652
>>> id(23.1)
4290657108
>>> id(3.14)
4290657109
>>> id(fahrenheit)
4290657110
>>> id(fahrenheit)
4290657110
```

The addresses you get will probably be different from what's listed above since values are stored wherever there happens to be free space. Function objects also have memory addresses:

```
>>> id(fabs)
4297064712
>>> id(round)
4297071160
```

### 3.3 Defining Our Own Functions

The built-in functions are useful but pretty generic. Often there aren't built-in functions that do what we want, such as calculate mileage or play a game of solitaire. When we want some time to do those sorts of things, we have to write them ourselves.

Because we live in Toronto, Canada, we often deal with our neighbor to the south. The United States typically uses Fahrenheit, so we convert from Fahrenheit to Celsius and back a lot. It would be nice to be able to do this:

```
>>> convert_to_celsius(71.7)
19.9
>>> convert_to_celsius(70.0)
21.0
>>> convert_to_celsius(10.4)
-12.0
```

## Python Remembers and Reuses Some Objects

A **name** is a reference to data. However, small programs—say, to about 250 or so, depending on the version of Python you’re using—aren’t memory. Python creates new objects and it also updates old objects with new objects when variables change. This saves up space and, by doing these changes, Python “remembers” them.

```
>>> i = 3
>>> j = 3
>>> k = i + 1
>>> l = k
>>> m = l
>>> n = l
>>> o = l
>>> p = q = r
```

What did happen to that variable *i*... and it refers to the same *new* object. That’s called **aliasing**.

It’s good and interesting performance, but it can be very bad!

```
>>> i = 1000000
>>> j = 1000000
>>> k = i
>>> l = j
>>> m = k
>>> n = i
>>> o = j
>>> p = k
>>> q = l
>>> r = m
```

Python didn’t do what you expect because the only reason you had to do *more* work is so that you aren’t surprised when it happens: the original *new* program is not affected by what Python decides to do!

However, for the `print` function there’s one step, an optional one, that lets it focus only on the last line of the print message for *now*:

```
now.print_line_to_console(217)
#which is just a print call now
#The sending = None is to simulate
#some other name 'connect_to_console' to print_lines
```

In the code, we have to write a `lambda` expression that tells Python *what* to do when this function is called.

We’ll go over the syntax of function definitions soon, but we’ll start with an example:

```
200 def convert_to_celsius(fahrenheit):
...     return (fahrenheit - 32) * 5 / 9
...
```

The `function` body is **indented**. Here, we indent four spaces, as the Python style guide recommends. If you forget to indent, you get this error:

```
200 def convert_to_celsius(fahrenheit):
...     return (fahrenheit - 32) * 5 / 9
    File "script.py", line 2
        return (fahrenheit - 32) * 5 / 9
        ^
IndentationError: expected an indented block
```

Now that we've defined function `convert_to_celsius`, our earlier function calls will work. We can even use built-in function `help` to do it:

```
200 help(convert_to_celsius)
Help on function convert_to_celsius in module __main__:

convert_to_celsius(fahrenheit)
```

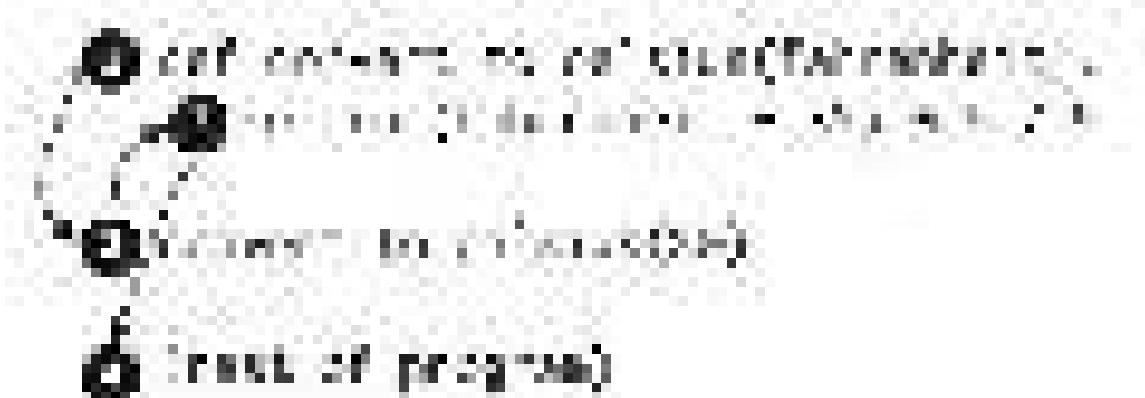
This shows the raw text of the function definition, which we call the **function header**. Later in this chapter, we'll show you how to add more help documentation to a function.

Here's a quick overview of how Python executes the following code:

```
200 def convert_to_celsius(fahrenheit):
...     return (fahrenheit - 32) * 5 / 9
...
201 convert_to_celsius(20)
202 #####
```

1. Python encounters the function definition, which creates the function object (but doesn't execute it yet).
2. Next, Python executes function call `convert_to_celsius`. To do this, it assigns `20` to `fahrenheit` (which is a variable). For the duration of this function call, `fahrenheit` is `20`.
3. Python now evaluates the `return` statement. `fahrenheit` refers to `20`, so the expression that appears after `return` is evaluated as `(20 - 32) * 5 / 9`. When Python evaluates that expression, `#####` is produced. We use the `#####` placeholder to tell Python what value to produce as the result of the function call, or the result of calling `convert_to_celsius(20)` as `20.000000000000006`.
4. Once Python has finished executing the function call, it returns to the place where the function was originally called.

Here is a picture showing this sequence:



A **function definition** is a **block** of Python statements. The general form of a function definition block looks like this:

```
def function_name(parameters):
    block
```

### Keywords Are Words That Are Special to Python

Keywords are words that Python reserves for its own use. We can't use them except as Python intends. Two of them are `def` and `return`. If we try to use them as other variable names, we get function blocks by mistake, and Python gives us an error:

```
def def(x):
    print("Hello")
    return x

MyError: invalid syntax
def def(x):
    print("Hello")
    return x
    def return x
```

**SError:** invalid syntax

There is a complete list of Python keywords. They're another important part of the language.

Name	mean	new	for	in	or	and
else	else	else	else	else	else	else
if	if	if	if	in	if	if
try	try	except	if	for loop	try	try
as	as	finally	lambda	in	as	as

The function header (that's the new line of the function definition) starts with `def`, followed by the name of the function, then a comma-separated list of parameters without any colons, and then a colon. A parameter is a variable.

You can't have two functions with the same name; it will be an error. But if you do it, the second function definition replaces the first one, much like assigning a value to a variable a second time replaces the first value.

Below the function header and indented from spaces, as per Python's style guide, is a block of statements called the function body. The function body must contain at least one statement.

Most function definitions will include a `return` statement that, when executed, exits the function and produces a value. The general form of a `return` statement is as follows:

```
return expression;
```

When Python executes a `return` statement, it evaluates the expression and then produces the result of that expression as the result of the function call.

## 3.4 Using Local Variables for Temporary Storage

Some computations are complex, and breaking them down into separate steps can lead to clearer code. Here, we break down the evaluation of the quadratic polynomial  $a x^2 + b x + c$  into several steps (notice that all the statements inside the function are indented the same amount of space in order to be aligned with each other):

```
>>> def quadratic(a, b, c, x):
...     first = a * x ** 2
...     second = b * x
...     third = c
...     return first + second + third
...
>>> quadratic(2, -4, 5, 1)
5.0
>>> quadratic(2, -4, 5, 2)
11.0
```

Variables like `first`, `second`, and `third` that are created within a function are called **local variables**. Local variables get created each time that line is to be called, and they are erased when the function returns. Because they only exist when the function is being evaluated, they can't be used outside the function. This means that trying to access a local variable from outside the function is an error, just like trying to access a variable that has never been defined is an error:

```
>>> quadratic(2, -4, 5, 3)
11.0
>>> first
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'first' is not defined
```

A function's parameters are also local variables, so we get the same error if we try to use them outside of a function definition:

```
>>> a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

The scope of a variable is the part of the program where the variable can be used. The scope of a local variable is from the line in which it is defined up until the end of the function.

As you might expect, if a function is defined to take a certain number of parameters, a call to that function must have the same number of arguments:

`ValueError: 2 != 3`

`Traceback (most recent call last):`

`File "testline", line 1, in <module>`

`TypeError: quadratic() takes exactly 4 arguments (3 given)`

Remember that you can call built-in functions to find out information about the parameters of a function:

### 3.5 Tracing Function Calls In the Memory Model

Read the following code. Can you predict what it will do when we run it?

```
>>> def f(x):
...     x = 2 * x
...     return x
...
>>> x = 1
>>> x = f(x) + f(x - 1)
```

That code is confusing, so I hope just because we used all over the place. However, it is pretty short and it only uses Python features that we have seen so far: assignment statements, expressions, function definitions, and function calls. We're interested in one question: Are all four `x`s the same variable? Does Python make a new `x` for each assignment? For each function call? For each function definition?

Here's the answer: whenever Python executes a function call, it creates a namespace (literally, a space for names) in which to store local variables for the call. You can think of a namespace as a scrap piece of paper. Python writes down the local variables on that piece of paper, keeps track of them as long as the function is being executed, and throws that paper away when the function returns.

Separately, Python keeps another namespace for variables created in the shell. That means that the `x` that is a parameter of `f` is a different variable than the `x` in the shell!

Let's refine our rules from [Section 3.1, Functions That Python Provides](#), on page 31: for executing a function call to include the namespace creation:

## Reusing Variable Names Is Common

Using the same name for local variables in different functions is quite common. For example, imagine a program that deals with documents—something that refers to a file value or the size of that file, perhaps. In that program, there would be several functions that all deal with these documents, and it would be entirely reasonable to have one variable name mean two very different things.

1. Evaluate the arguments left to right.
2. Create a namespace to hold the function call's local variables, including the parameters.
3. Pass the resulting argument values into the function by associating them to the parameters.
4. Evaluate the function body. As before, when a return statement is executed, the value of the body terminates and the value of the expression in the return statement is used as the value of the function call.

From now on in our memory model, we will draw a separate box for each namespace to indicate that the variables inside it are in a separate area of computer memory. The programming work exists this *within* frame. We separate the frame from the object by a vertical called line:

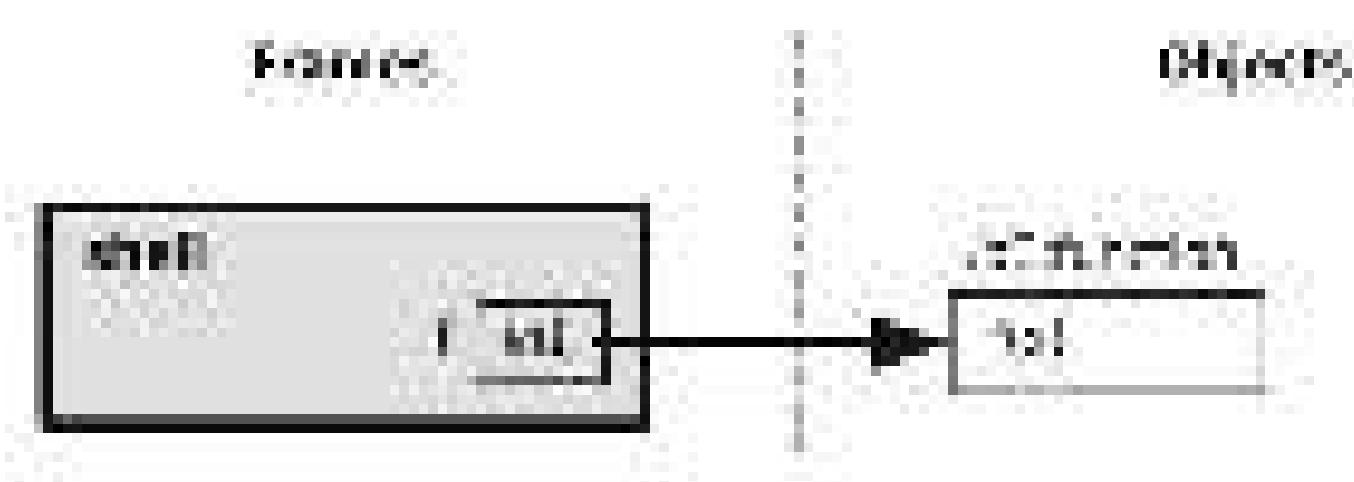


Using our newfound knowledge, let's trace that confusing code. At the beginning, no variables have been created; Python is about to execute the function definition. We have indicated this with an arrow:

```
* def f(x):
...     x = 2 * x
...     return x

** x = 1
*** x = f(x + 1) + f(x + 2)
```

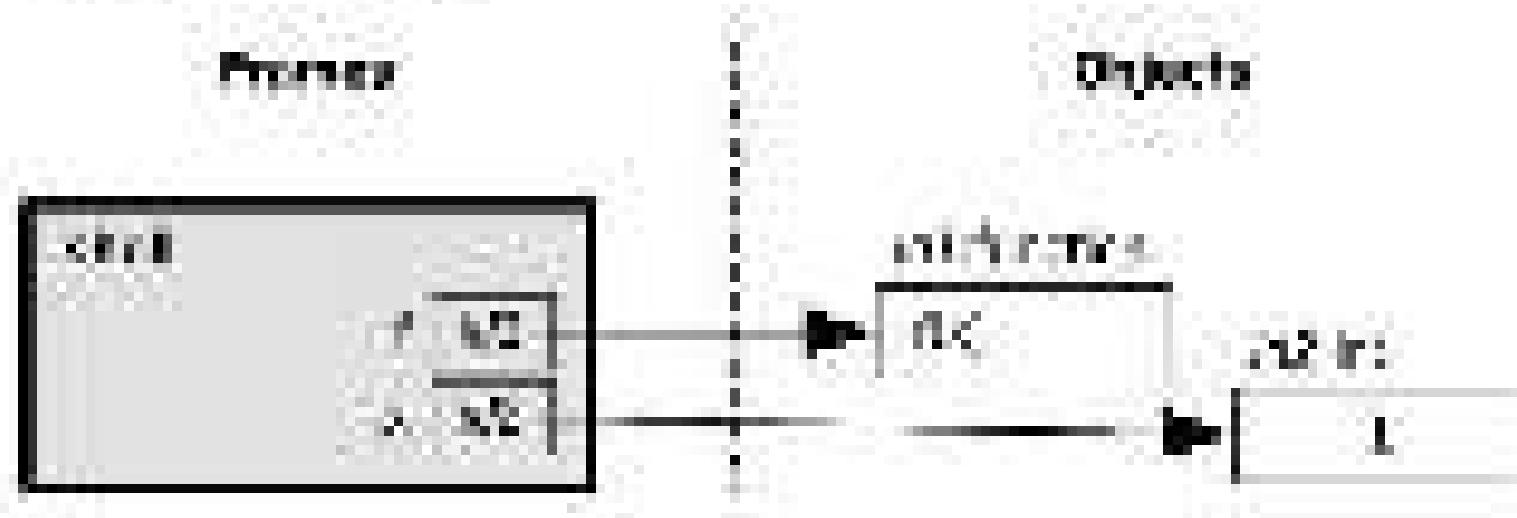
As you've seen in this chapter, when Python executes that first line of function, it creates a variable `f` in the frame for the shell's namespace plus a function object. If Python didn't execute the body of the function, that won't happen until the function is called. Here is the result:



Now we are about to evaluate the first assignment to `x` in the shell:

```
>>> def f(x):
...     x = 2 * x
...     return x
...
> x = 1
>>> x = f(x) + f(x - 1)
```

Observe that assignment happens high-level and *nowhere* in the frame or the shell:



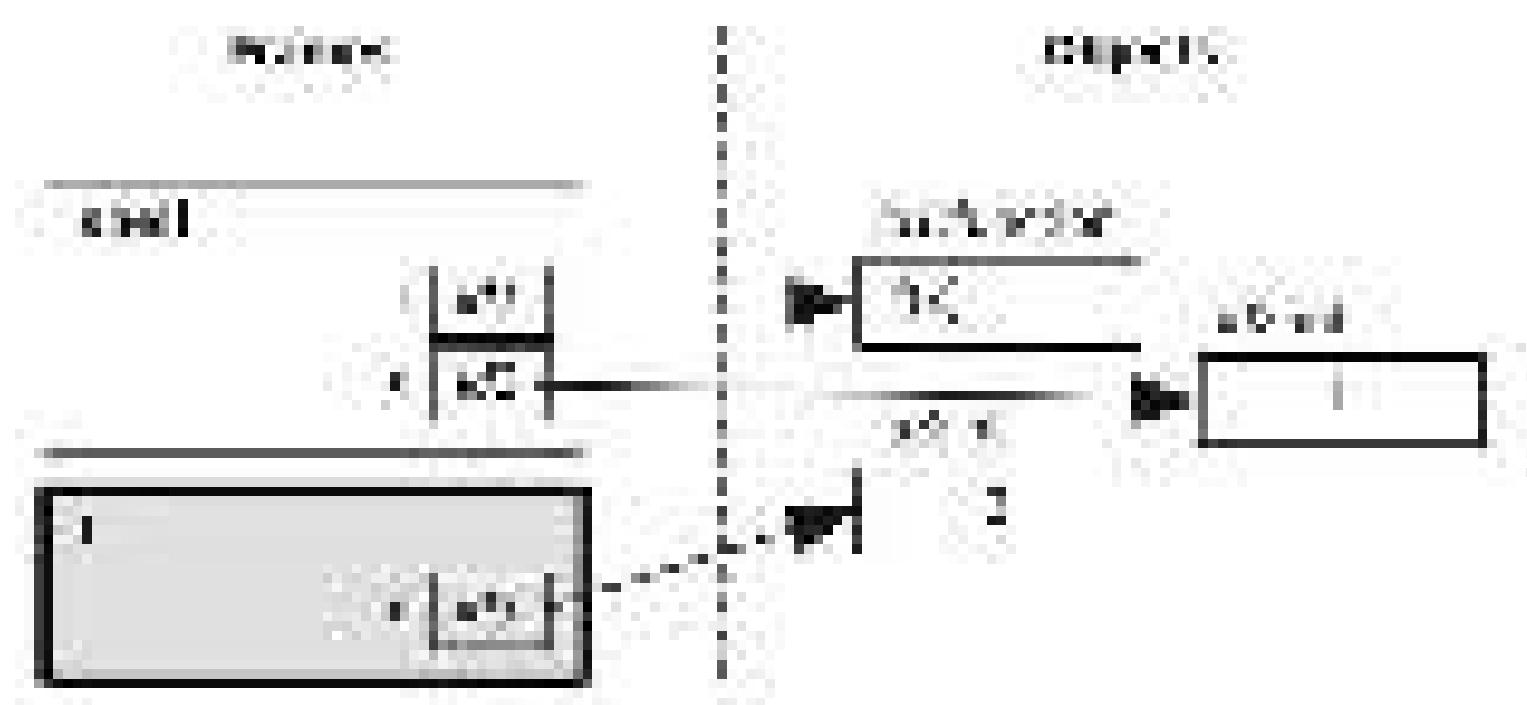
Now we are about to execute the second assignment to `x` in the shell:

```
>>> def f(x):
...     x = x * x
...     return x
...
> x = 1
>>> x = f(x) + f(x - 1)
```

Following the rules for evaluating an assignment to [from assignment functions](#), on page 18, we first evaluate the expression on the right of the `=`, which is `f(x) + f(x - 1)`. Python evaluates the left function call first: `f(x + 1)`.

Following the rules for evaluating a function call, Python evaluates the argument `x + 1` in order to find the value for `x`. Python looks in the current frame. The current frame is the frame for the shell, and the variable `x` there is 1. So `x + 1` evaluates to 2.

Now we have evaluated the argument to `f`. The next step is to create a `parameter` for the function call. We have a frame, with its parameter `x` and `value 2` in that parameter:

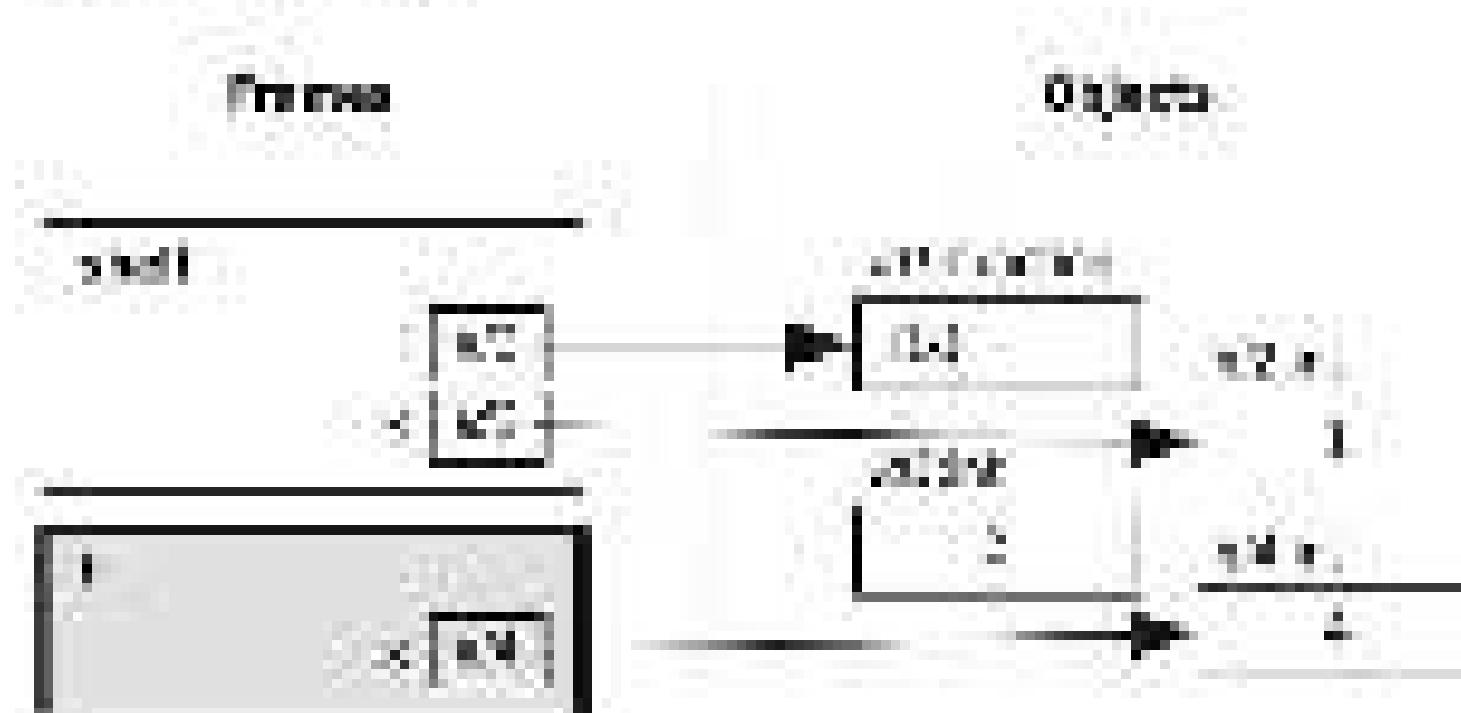


Note: that there are two variables called `x`, and they refer to different values. Python will always look in the current frame, which we will draw with a thicker border.

We are now about to execute the `y = 2 * x` statement in function `f`:

```
>>> def f(x):
...     x = 2 * x
...     return x
...
>>> x = 1
>>> x = f(x + 1) + f(x + 2)
```

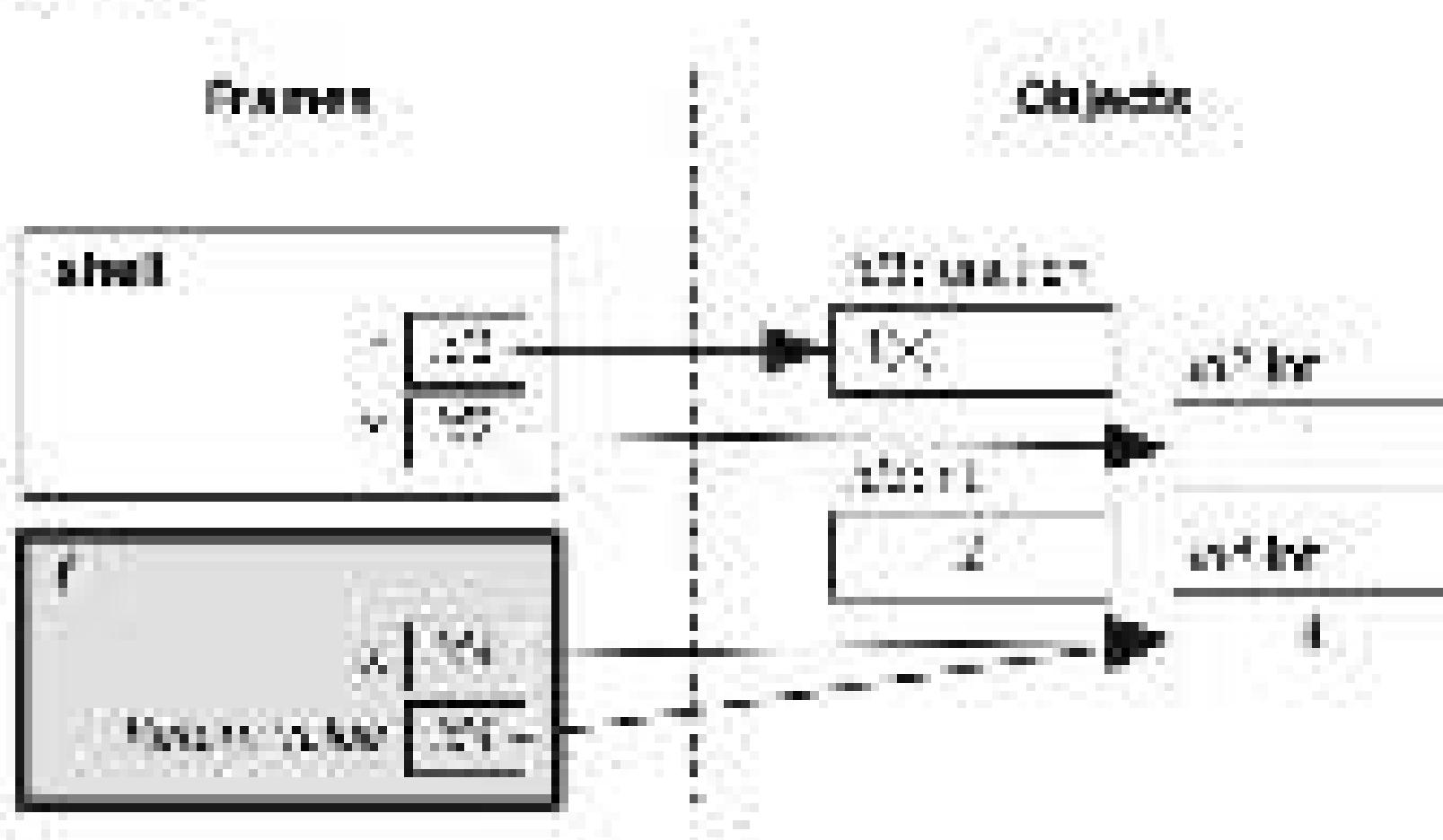
`y = 2 * x` is an assignment statement. The right side is the expression `2 * x`. Python looks up the value of `x` in the current frame and finds 3, so the expression evaluates to 4. Python finishes executing that assignment statement by making `y` refer to that 4.



We are now about to execute the `return x` statement in function `f`:

```
>>> def f(x):
...     x = 2 * x
...     return x
...
>>> x = 1
>>> x = f(x + 1) + f(x + 2)
```

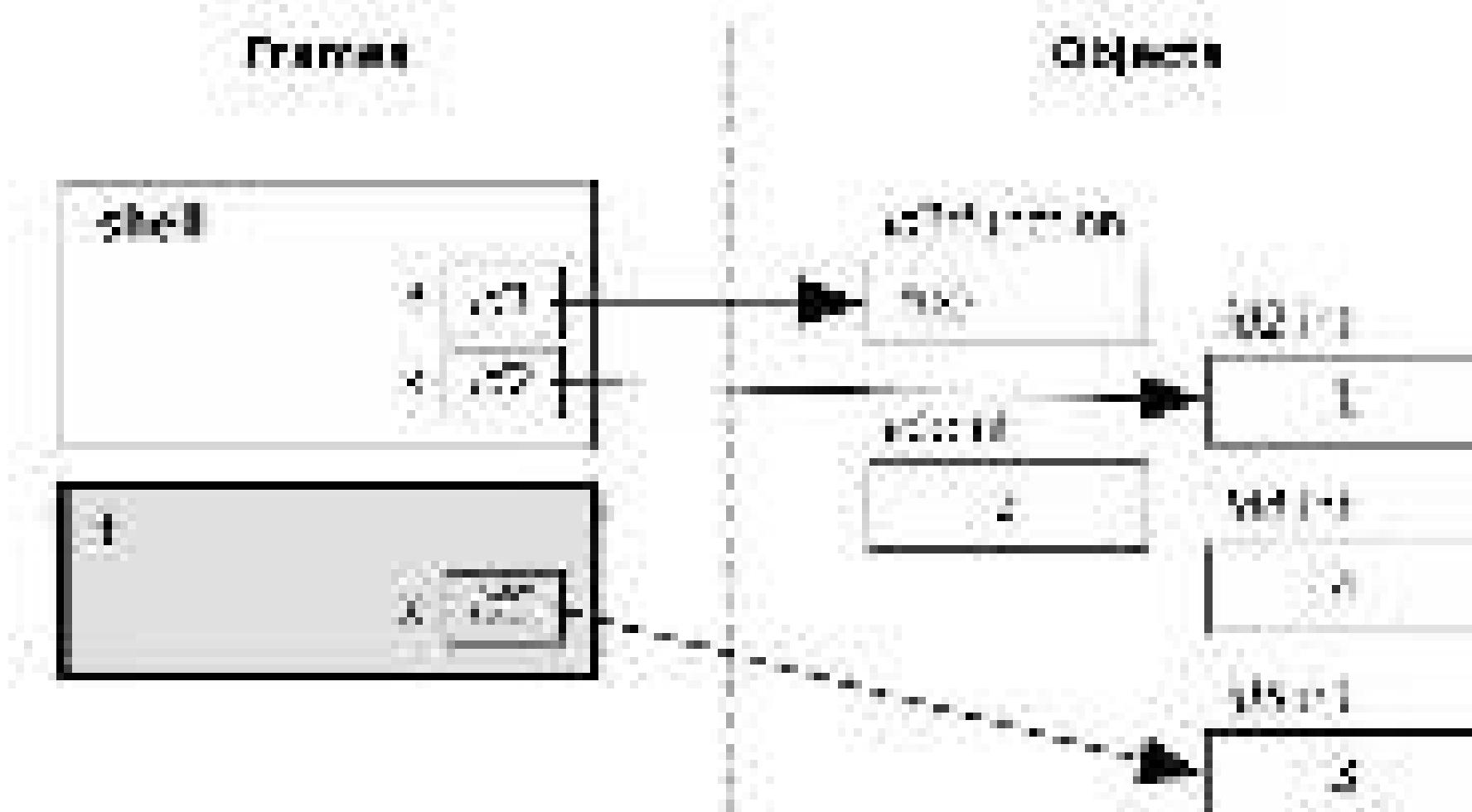
This is a return statement, so we evaluate the expression, which, in example 7, Python looks up the value for *x* in the current frame and finds 4, so that is the return value:



When the function returns, Python comes back to the expression  $x + 1 + (x + 2)$ . Python just finished evaluating  $x = 4$ , which produced the value 4. It then executes the right portion of  $x + 2$ .

Following the rules for evaluating a function call, Python evaluates the argument  $x = 2$ . In order to find the value for *x*, Python looks in the current frame. The call to function *f* has returned, so that frame is erased; the only frame left is the frame for the shell, and the variable *x* still refers to 4, so  $x + 2$  evaluates to 5.

Now we have evaluated the argument to *f*. The next step is to create a connection for the function call. We draw a frame, set it in the parameter *x*, and assign 5 to that parameter:

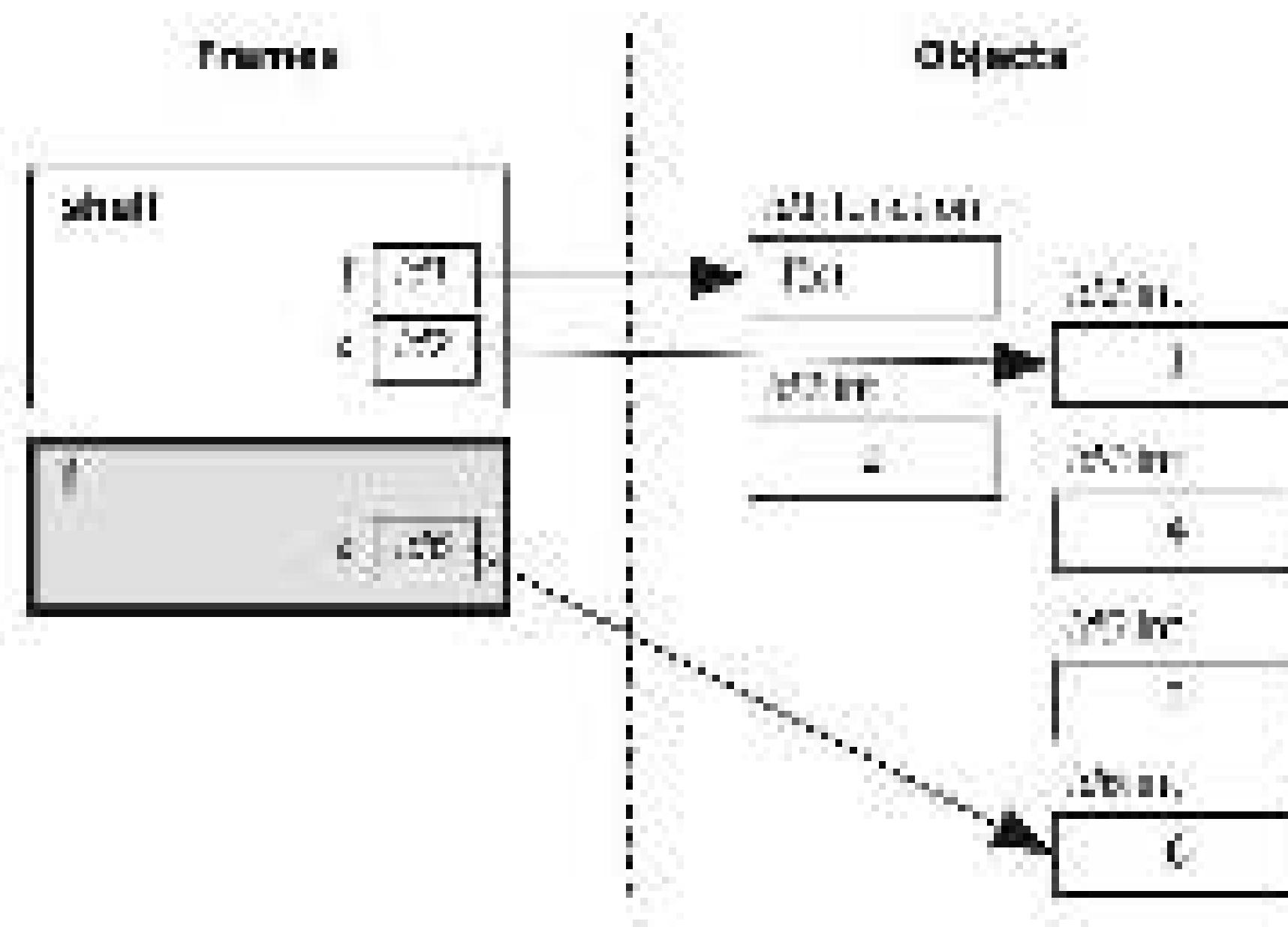


Again, we have two variables called *x*.

We are now about to execute the first statement of function `f`:

```
>>> def f(x):
...     x = 2 * x
...     return x
...
>>> x = 1
>>> x = f(x + 1) + f(x + 2)
```

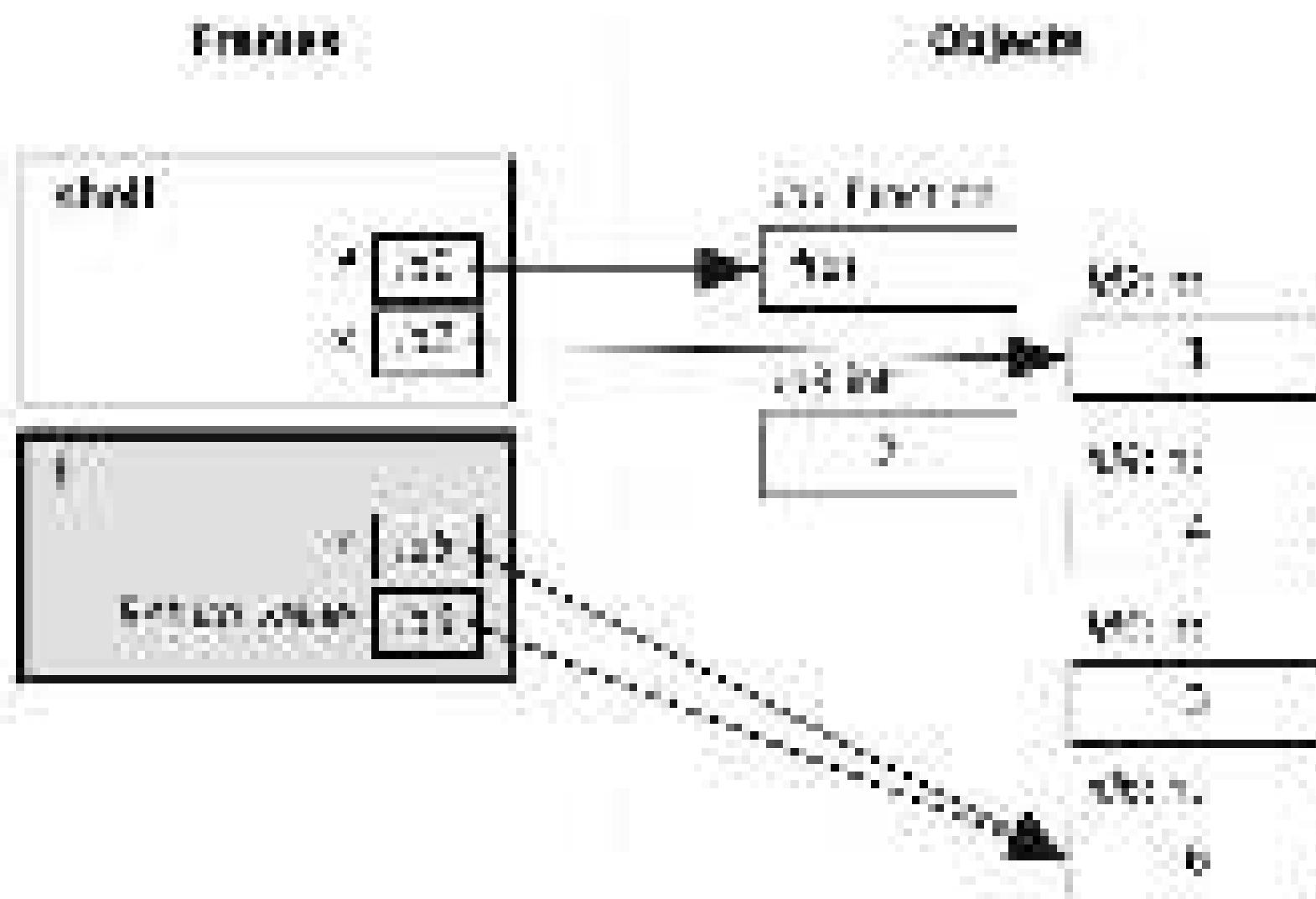
`x = f(x)` is an assignment statement. The right side to the expression `f(x)` Python looks up the value of `x` in the current frame and finds 1, so that expression evaluates to 2. Python finished executing that assignment statement by making `x` refer to that 2:



We are now about to execute the second statement of function `f`:

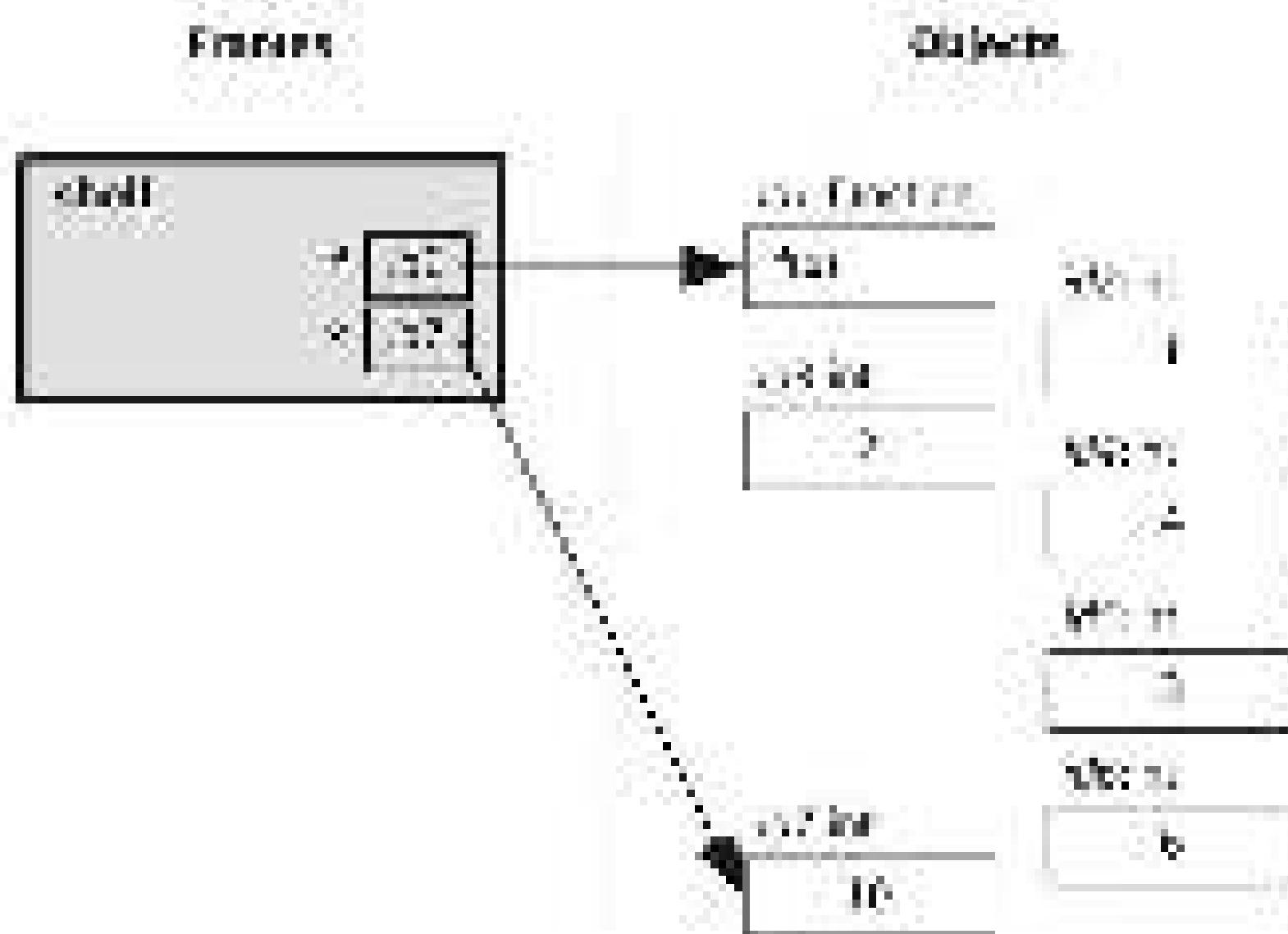
```
>>> def f(x):
...     x = 2 * x
...     return x
...
>>> y = 1
>>> y = f(y + 1) + f(y + 2)
```

This is a `return` statement, so we evaluate the expression which is simply `y`. Python looks up the value for `x` in the current frame and finds 3, so that is the `return` value:



When the function returns, Python comes back to the expression  $(x + 1) * (y + 2)$ . Python just finished calculating  $(x + 1)$ , which produced the value 5. Both function calls have been executed, so Python applies the `*` operator to 5 and 2, giving us 10.

We have now evaluated the right side of the assignment statement. Python completes it by making the variable on the left side, `a`, refer to 10:



*Note:* That's a lot to keep track of. Python does all that bookkeeping for us, but as a good programmer it's important to understand such individual steps.

## 3.6 Designing New Functions: A Recipe

Writing a good essay requires planning, deciding on a topic, learning the background material, writing an outline, and then filling in the outline until you're done.

Writing a good function also requires planning. You have an idea of what you want the function to do, but you need to decide on the details. What do you name the function? What are the parameters? What does it return?

This section describes a step-by-step recipe for designing and writing a function. First of all, the outcome will be a working function, but almost as important is the documentation for the function. Python uses three double quotes to start and end this documentation, consisting in between is meant for humans to read. This notation is called a docstring, which is short for documentation string.

Here is an example of a completed function. We'll show you how we came up with this using a function design recipe (FDR), but it helps to see a completed example first:

```
<<< day_difference(day1, day2)
    """
    (int, int) -> int
    ...
    ...
    Return the number of days between day1 and day2, which are
    both in the range 1-365 (thus indicating the day of the
    year).
    ...
    >>> day_difference(100, 124)
    24
    >>> day_difference(50, 50)
    0
    >>> day_difference(100, 99)
    -1
    ...
    >>> return day2 - day1
    ...>
```

Here are the parts of the function, including the docstring:

- The first line is the function header.
- The comment line has three double quotes to start the docstring. The first part describes the types of values expected to be passed to parameters `day1` and `day2`, and the last part is the type of value the function will return.
- After that is a description of what the function will do when it is called. It mentions both parameters and describes what the function returns.

- Next are some example calls and return values we would expect to see in the Python shell. (We chose the first example because that made day smaller than day2, the second example because the two days are equal, and the third example because that meant day1 bigger than day2.)
- The last line is the body of the function.

There are six steps in the function design process. In this section, there is lots of work at first, but this stage can save you hours of time when you’re working on more complicated functions.

1. **examples:** The first step is to figure out what arguments you want to pass to your function and what information it will return. Pick a name (often a verb or verb phrase); this name is often a short answer to the question, “What does your function do?” Type a couple of example calls and return values.

We start with the examples because they’re the easiest before we write anything: we need to decide what information we have [the argument values] and what information we want the function to produce [the return value]. Here are the examples from `days_difference`:

```
--> >>> days_difference(200, 100)
200
--> >>> days_difference(0, 100)
100
--> >>> days_difference(200, 90)
-100
```

2. **Type contract:** The second step is to figure out the types of information your function will handle. Are you giving it integers? Floating-point numbers? Maybe both? We’ll see a lot of other types. In the upcoming chapters, we won’t do this step now while you only have a few choices will help you later. If the answer is, “Both integers and floating-point numbers,” then use the word `number`.

Also, what type of value can `abs` return? An integer, a floating-point number, or possibly either one of them?

This is called a *contract* because we are claiming that if you call this function with the right types of values, we’ll give you back the right type of value. (We’re not saying anything about what will happen if we get the wrong types of values.) Here is the type contract from `days_difference`:

```
--> >>> abs((int, int)) -> int
```

3. Header. Write the function header. Pick meaningful parameter names to make it easy for other programmers to understand what information to give to your function. Here is the header from `days_difference`:

```
>>> def days_difference(day1, day2):
```

4. Description. Write a short paragraph describing your function: this is what other programmers will read in order to understand what your function does. It's important to get this right. Mention every parameter in your description and describe the return value. Here is the description from `days_difference`:

Return the number of days between day1 and day2, which are both in the range 0-365 (thus indicating the day of the year).

5. Body. By now, you should have a good idea of what you must do. In order to get your function to behave properly, it's time to write some code. Here is the body for `days_difference`:

```
>>> return day1 - day2
```

6. Test. Run the examples to make sure your function body is correct. Test first, or add more example calls if you happen to think of them. For `days_difference`, we copy and paste our examples into the shell and compare the results to what we expected:

```
>>> days_difference(200, 201)
21
>>> days_difference(50, 50)
0
>>> days_difference(100, 99)
-1
```

## Designing Three Birthday-Related Functions

We'll now apply our functions design recipe to solve this problem: Which day of the week will a birthday fall upon, given what day of the week it is today and what day of the year the birthday is on? For example, if today is the third day of the year and it's a Thursday, and a birthday is on the 119th day of the year, what day of the week will it be on that birthday?

We'll design three functions that together will help us do this calculation. We'll write them in the same file (and we'll do this in [Chapter 8: A More Structured Approach to Program Organization](#), on page 196), we'll need to put functions that we write in the same file. How can we be able to have them call one another?

We will represent the day of the week using 1 for Sunday, 2 for Monday, and so on:

Day of the Week	Number
Sunday	1
Monday	2
Tuesday	3
Wednesday	4
Thursday	5
Friday	6
Saturday	7

We are using these numbers simply because we don't yet have the tools to easily convert between days of the week and their corresponding numbers. We'll have to do that translation to our heads.

For the same reason, we will also ignore months and use the numbers 1 through 365 to indicate the day of the year. For example, we'll represent November 1 as 309, since it's the ninety-ninth day of the year.

### How Many Days Difference?

We'll start by seeing how we come up with function `days_difference`. Here are the function design recipe steps. Try following along in the Python shell.

- Example: We want a clear name for the difference in days, so I use `days_difference`. In our examples, we want to call this function and state what it returns. If we want to know how many days there are between the 20th day of the year and the 22nd day, we can expect this will happen.

```
>>> days_difference(20, 22)
2
```

What if the two days are the same? How about if the second one is before the first?

```
>>> days_difference(99, 99)
0
>>> days_difference(100, 99)
-1
```

Now that we have a few examples, we can move on to the next step.

2. **Type Check:** The arguments in our function call examples are all integers and the return values are integers too, so here's our type contract:
- $$\text{int}, \text{int} \rightarrow \text{int}$$

3. **Header:** We have a copy of example calls, and we know what types the parameters are, so we can now write the header. Both arguments are day numbers, so we'll use `day1` and `day2`:

```
<del>def days_difference(day1, day2):
```

4. **Description:** We'll now overwrite our `#call` with `#doc`. However, the documentation should compactly describe the behavior of the function, so we need to make sure that it's clear what the parameters mean:

~~return~~ The number of days between `day1` and `day2`, which are both in the range 0-365 (thus indicating a day of the year).

5. **Body:** We've left everything out. Looking at the examples, we see that we can compute this using subtraction. Here is the whole function again, including the body:

```
<del>def days_difference(day1, day2):
    <del>    """ int, int ) -> int
```

~~return~~ The number of days between `day1` and `day2`, which are both in the range 0-365 (thus indicating the day of the year).

```
<del>
<del>def days_difference(day1, day2):
<del>    """ int, int ) -> int
<del>    return day2 - day1
```

6. **Test:** To test it, we fire up the Python shell and copy and paste the calls into the shell, checking that we get back what we expect:

```
<del>>> days_difference(200, 221)
21
<del>>> days_difference(50, 50)
0
<del>>> days_difference(300, 90)
-60
<del>>>
```

Here's something really cool. Now that we have a function with a default arg, we can call it just like that function:

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


```

```
days_difference(d1, d2)
    (int, int) -> int
```

**Return the number of days between d1 and d2, which are both in the range 1-365 (thus indicating the day of the year).**

```
>>> days_difference(200, 224)
24
>>> days_difference(50, 50)
0
>>> days_difference(100, 86)
-1
```

## What Day Will It Be in the Future?

We will build our first today calculation. If we write a function to calculate what day of the week it will be given the current weekday and how many days ahead we're interested in. Remember that we're using the numbers 1 through 7 to represent Sunday through Saturday.

Again, we'll follow the function design recipe:

1. **Example:** We want a short name for what it means to calculate what weekday it will be in the future. We could choose something like `add_weekday` or `next_day`; we'll use `get_weekday`. There are lots of choices.

We'll start with an example that asks what day it will be tomorrow (Tuesday [May 3 of the year]) and we want to know what tomorrow will be [if they sleep].

```
>>> get_weekday(3, 1)
3
```

Whenever we have a function that should return a value in a particular range, we should write example calls where we expect either end of that range as a result.

What is it? Friday [May 4]? If we ask what day it will be tomorrow, we expect to get Saturday [May 5]:

```
>>> get_weekday(3, 2)
5
```

What if it's Sunday (day 7)? If we ask what day it will be tomorrow, we expect to get Sunday (day 1):

```
*** get_weekday(7, 1)
1
```

We're also trying about 10 days in the future; as well as a workday; but first three days should then mark the end of the week we started with:

```
*** get_weekday(2, 0)
0
*** get_weekday(4, 7)
4
```

Let's also try 10 weeks, and 2 days in the future so we have a case where there are several intervening weeks:

```
*** get_weekday(2, 72)
3
```

2. **Type Contract:** The arguments in our function call examples are all integers, and the return values are integers too, so here's our type contract:  
`(int, int) → int`
3. **Example:** We have a couple of example calls, and we know what type the parameters are, so we can move with the leader. The function name is clear, so we'll stick with it.

The first argument is the current day of the week, as well as current date to. The second argument is how many days from now to calculate. We'll pick `days_ahead`, although `days_from_today` would also be fine:

```
def get_weekday(current_weekday, days_ahead).
```

4. **Description:** We need a complete description of what this function will do. We'll start with a sentence describing what the function does, and then we'll describe what the parameters mean:

`Return a day of the week; it will be days_ahead days from current_weekday.`

`current_weekday` is the current day of the week and is in the range 0-6, associating whether today is Sunday (0), Monday (1), ..., Saturday (6).

`days_ahead` is the number of days after today.

Note that our first sentence uses both parameters and also describes what the function will return.

- b. Recalling back to the example we saw that we can add the first example with this `int % int = int` idea. That, however, won't work for all of the examples; we need to wrap around from day 7 (Saturday) back to day 1 (Sunday). When you have this kind of **exceptional**, usually the remainder operator, `%`, will help. Notice that evaluation of  $0 - 1 \% 7$  produces 1,  $7 + 2 \% 7$  produces 2, and so on.

Let's try taking the remainder of the sum: `start_weekday + days_ahead % 7`. Here is the whole function again, including the body:

```
def get_weekday(current_weekday, days_ahead):
    """Int, int) -> int

    Returns which day of the week it will be days_ahead days from
    current_weekday.

    current_weekday is the current day of the week and is in the
    range 0-6, indicating whether today is Sunday (0), Monday (1),
    ..., Saturday (6).

    days_ahead is the number of days after today.

    """
    return current_weekday + days_ahead % 7
```

- c. Next, let's test it—we fire up the Python shell and type and paste the calls into the shell, checking that we get back what we expect:

```
>>> get_weekday(0, 1)
1
>>> get_weekday(0, 3)
3
>>> get_weekday(7, 1)
1
>>> get_weekday(2, 9)
1
>>> get_weekday(4, 7)
4
>>> get_weekday(7, 72)
2
>>>
```

Well, that's not right. We expected to get that third example, not an 8, because 8 isn't a valid number for a day of the week. We should have wrapped `modulus` to 1.

Taking another look at our function body, we see that `lambda` has higher precedence than `*`, we need parentheses:

```
def get_weekday(current_weekday, days_ahead):
    """(int, int) -> int
```

return which day of the week it will be `days_ahead` days from `current_weekday`.

`current_weekday` is the current day of the week and is in the range 1-7, indicating whether today is Sunday (1), Monday (2), ..., Saturday (7).

`days_ahead` is the number of days after today.

```
xxx get_weekday(3, 1)
4
xxx get_weekday(3, 2)
5
xxx get_weekday(7, 1)
1
xxx get_weekday(1, 3)
3
xxx get_weekday(4, 7)
4
xxx get_weekday(7, 72)
2
...
return (current_weekday + days_ahead) % 7
```

Testing again, we see that we fixed that bug in our test, but now we're getting the wrong answer for the second test:

```
xxx get_weekday(3, 1)
4
xxx get_weekday(3, 2)
5
xxx get_weekday(7, 3)
5
```

The problem here is that when `current_weekday + days_ahead` evaluates to a multiple of 7, then `(current_weekday + days_ahead) % 7` will evaluate to 0, not 7. All the other results work well. It's just that pesky 7.

Remember we want a number in the range 1 through 7 that we're getting an answer in the range 0 through 6 and all the answers are correct except that we're seeing a 0 instead of a 7. We can use this trick:

- Subtract 1 from the expression:  $\text{ans} = \text{ans} + \text{ans} - 1$ .
- Take the remainder.
- Add 1 to the entire result:  $\text{ans} = \text{ans} + \text{ans} + 1 \% 7 - 1$ .

Let's test it again:

```
>>> get_weekday(4, 0)
0
>>> get_weekday(4, 1)
1
>>> get_weekday(4, 2)
2
>>> get_weekday(4, 3)
3
>>> get_weekday(4, 7)
0
>>> get_weekday(4, 720)
0
```

We've passed all the tests so we can move on.

## What Day Is My Birthday On?

We now have two functions related to day-of-year calculations. One of them calculates the difference between two days of the year. The other calculates the weekday for a day in the future given the weekday today. We can use these two functions to help figure out what day of the week a birthday falls on given what day of the week it is today, what the current day of the year is, and what day of the year the birthday falls on.

Again, we'll follow the function design recipe:

- conceptual: We want a name for what it means to calculate what weekday a birthday will fall on. Since there are lots of things we'll use `get_weekday`:

If today is a Thursday (day 5 of the week), and today is the third day of the year, what day will it be on the fourth day of the year? **Thursday**.  
**Finals**

```
>>> get_birthday_weekday(5, 3, 4)
5
```

What if it's the second day (Thursday, the 2nd day of the year), but the birthday is the 119th day of the year? Furthermore, we can verify externally (looking at a calendar) that it turns out to be a Friday:

```
>>> get_birthday_weekday(5, 2, 119)
```

```
5
```

What if today is Friday, 28 April, the 118th day of the year, but the birthday we want is the 3rd day of the year? This is interesting because the birthday is a couple months before the current day:

```
>>> get_birthday_weekday(5, 3, 3)
```

```
6
```

2. **Type Check.** The arguments in our function call examples are all integers, and the return values are integers too, so here's our type contract:

```
def get_birthday_weekday(current_weekday, current_day, birthday_day):
```

3. **Flair.** We have a couple of example calls, and we know what types the parameters are, so we can move with the flair. We're happy enough with the function name so again we'll stick with it.

The first argument is the current day of the week, so we'll use `current_weekday`, as we did for the previous function. [It's a good idea to be consistent with naming when possible.] The second argument is what day of the year it is today, and will answer `current_day`. The third argument is the day of the year the birthday is, and will choose `birthday_day`:

```
def get_birthday_weekday(current_weekday, current_day, birthday_day):
```

4. **Description.** We need a complete description of what this function will do. We'll start with a sentence describing what the function does, and then we'll describe what the parameters mean:

Returns the day of the week it will be on `birthday_day`, given that the day of the week is `current_weekday` and the day of the year is `current_day`.

`current_weekday` is the current day of the week and is in the range 1-7, indicating whether today is Sunday (1), Monday (2), ..., Saturday (7).

`current_day` and `birthday_day` are both in the range 1-365.

Again, notice that our function uses all parameter annotations to tell the function what it will return. If a parameter is redundant, we'll want to write multiple statements to describe what the function does, but we managed to squeeze it in here.

6. **Begin:** Now time to write the body of the function. You have a puzzle:
- a. Using `days_2` formula, we can figure out how many days there are between two days.

Except yet weekday, we can figure out what day of the week it will be given the current *day* of the week and the number of days away

- With that by running math how many days from now the birthday falls:

```
days_diff = days_difference(current_day, birthday_day)
```

Now that we know that... we can use it to solve our problem: given the current `weekday` and that number of days ahead, we can call function `get_weekday` to get our answer:

```
return get_weekday(current_weekday, days_diff)
```

Let's put it all together:

```
def get_birthday_weekday(current_weekday, current_day,
                           birthday_day):
    """(int, int, int) -> int
```

Return the day of the week it will be on `birthday_day`,  
given that the day of the week is `current_weekday` and the  
day of the year is `current_day`.

`current_weekday` is the current day of the week and is in  
the range 1-7, indicating whether today is Sunday (1),  
Monday (2), ..., Saturday (7).

`current_day` and `birthday_day` are both in the range 1-365.

```
    days_diff = days_difference(current_day, birthday_day)
```

```
    #
```

```
    return get_weekday(current_weekday, days_diff)
```

```
    #
```

```
    #
```

```
    days_diff = days_difference(current_day, birthday_day)
```

```
    return get_weekday(current_weekday, days_diff)
```

6. Test: To test it we fire up the Python shell and copy and paste the calls into the shell. Checking that we get back what we expect:

```
get_birthday_weekday(5, 41)
```

```
#
```

```
get_birthday_weekday(5, 100)
```

```
5
>>> get_birthday_weekday(1998, 10, 10)
```

And we're done!

### 3.7 Writing and Running a Program

So far, we have used the shell to evaluate Python. As you have seen, the shell will show you the result of evaluating an expression:

```
>>> 3 + 5 * abs(-2)
```

```
16
```

In a program that is supposed to interact with a human, showing the result of every expression to produce no destructive behavior. (Imagine if your bank teller showed you the result of every calculation it performed.)

[Section 9.1, What Does a Computer Run a Python Program?](#), on page 7, explained that in order to save code for later use, you can put it in a file with a .py extension. You can then tell Python to run the code in that file either by type commands at the interactive prompt.

Here is a program that we wrote using IDLE and saved in a file called `convert.py`. This program consists of a function definition for `convert_to_celsius` (from earlier in the chapter) and three calls to this function that convert three different Fahrenheit temperatures to their Celsius equivalents.



```
convert.py
```

```
def convert_to_celsius(F):
    """Convert Fahrenheit to Celsius"""
    celsius = (F - 32) * 5/9
    return celsius

convert_to_celsius(75)
convert_to_celsius(65)
convert_to_celsius(50)
```

Note that there is no >>> prompt. This never appears in a Python program; it is used exclusively in the shell.

Now open IDLE, select File—New Window and type this program in. (Either this or download the [code](#) and the back-slash-and-up-dir(.) file.)

To run the program in IDLE, select Run—Run Module. This will open the Python shell and show the results of running the program. Here is our result:

(The line containing `del all` is letting us know that the shell has restarted, wiping out any previous work done in the shell.)

```
Python 3.6.5 (v3.6.5:3afe7c857161, Mar 29 2018, 16:56:38)
[GCC 4.2.1 Compatible Apple LLVM 9.0.0 (clang-900.0.39.2) (based on LLVM 9.0.0~rc1-2-g45769ac)]
Type "help", "copyright", "credits" or "license" for more information.
>>> print(2+2)
4
>>>
```

Note that no values are shown, unlike in [Section 3.3, “Getting Our First Function”](#) on page 38, where we typed the equivalent code, `func() + 2`. In order to have a program print the value of an expression, we use built-in function `print`. Here is the entire program but with calls to `function print`:

```
def add(x,y):
    return x+y

print(add(2,2))
print(add(2,2))
print(add(2,2))

>>> print(2+2)
4
>>>
```

And here is what happens when we run this program:

```
Python 3.6.5 (v3.6.5:3afe7c857161, Mar 29 2018, 16:56:38)
[GCC 4.2.1 Compatible Apple LLVM 9.0.0 (clang-900.0.39.2) (based on LLVM 9.0.0~rc1-2-g45769ac)]
Type "help", "copyright", "credits" or "license" for more information.
>>> print(2+2)
4
>>> print(2+2)
4
>>> print(2+2)
4
>>>
```

## 3.8 Omitting a Return Statement: None

If you don't have a `return` statement in a function, nothing is produced:

```
>>> def f(x):
...     x = 2 * x
...
>>> res = f(2)
>>> res
None
```

Well, that can't be right—if we don't have a value, shouldn't we get a `None`? Let's take a little more time to print out:

```
>>> print(x)
```

```
None
```

```
>>> type(x)
```

```
<type 'NoneType'>
```

Variables can have a value. We know that there can be *no* value. If you don't have a return statement in your function, your function will return `None`. You can return `None` yourself if you like:

```
>>> def f(x):
```

```
...     y = x + 1
```

```
...     return None
```

```
...>
```

```
>>> print(f(1))
```

```
None
```

This `value None` is used to signal that there is no value. We'll see some cases for it later in the book.

## 3.9 Dealing with Situations That Your Code Doesn't Handle

You'll often write a function that only works in some situations. For example, you might write a function that takes as a parameter a number of people who want to eat a pie and returns the percentage of the pie that each person gets to eat. If there are three people, each person gets 33%; if there are two people, each person gets 50%; if there is one person, that person gets 100%. But if there are zero people, what should the answer be?

Here is an implementation of this function:

```
def pie_percent(n):
    ... (not) :: (not)
```

Assuming there are n people who want to eat a pie, return the percentage of the pie that each person gets to eat.

```
>>> pie_percent(3)
33
>>> pie_percent(2)
50
>>> pie_percent(1)
100
...
>>>
```

```
>>> return int(pie / n)
```

Reading the code, if someone calls `pct()`, then you probably see that this will result in a `ZeroDivisionError`. There isn't anything that anyone can do about this situation: there isn't a sensible answer.

As a programmer, you warn other people about situations that your function isn't set up to handle by describing your assumptions in a precondition. Here is the same function with a precondition:

```
def pct_percent(n):
    """(int) --> float
```

**Precondition:**  $n > 0$

Assuming there are  $n$  people who want to eat a meal, return the percentage of the meal that each person gets to eat.

```
def pct_percent(n):
    """(int) --> float
    Precondition: n > 0
    """
    return n / 100
```

`return n / 100` ; )

Whenever you write a function and you're assuming something about the parameter values, write a precondition that lets other programmers know your assumptions. If they ignore your warning and call it with invalid values, the fault does not lie with you!

### 3.10 What Did You Call That?

- A **function definition** introduces a new variable that refers to a function object. The `return` statement describes the value that will be produced as a result of the function when this function is done being executed.
- A **parameter** is a variable that appears between the parentheses of a function header.
- A **local variable** is a variable that is used in a function definition to store an intermediate result in order to make code easier to write and read.
- A **function call** tells Python to execute a function.
- An **argument** is an expression that appears between the parentheses of a function call. The value that is produced when Python evaluates the expression is assigned to the corresponding parameter.

- If you make assumptions about the values of parameters or you know that your function won't work with particular values, write a precondition to warn other programmers.

### 3.11 Exercises

Here are some exercises for you to try on your own. Solutions are available at <http://www.pythontutorial.net/problems-solutions/>.

- Two of Python's built-in functions are `min` and `max`. In the Python shell, execute the following function calls:
  - `min(2, 3, 4)`
  - `max(2, 3, 4, 5)`
  - `max(2, 3, min(2, 3), 5)`
- For the following function calls, in what order are the subexpressions evaluated?
  - `min(max(2, 4), 10/5)`
  - `abs(min(-5, max(2, 10)))`
  - `max(100/100, 100*100)`
- Following the function design recipe, define a function that has one parameter, a number, and returns that number tripled.
- Following the function design recipe, define a function that has two parameters, both of which are numbers, and returns the absolute value of the difference of the two. Hint: Call built-in function `abs`.
- Following the function design recipe, define a function that has one parameter, a distance in kilometers, and returns the distance in miles. (There are 1.6 kilometers per mile.)
- Following the function design recipe, define a function that has three parameters: `gpa` (between 0 and 100 inclusive), and `credits` (the weight of those grades).
- Following the function design recipe, define a function that has four parameters, all of them `gpa` (inclusive 0 and 100 inclusive), and returns the average of the best 3 of those grades. Hint: Call the function that you defined in the previous exercise.
- Complete the examples in the doctesting and then write the body of the following function:

```

def weeks_between(day1, day2):
    """ (date, date) -> float

    day1 and day2 are dates in the same year. Return the number of full weeks
    that have elapsed between the two days.

    >>> weeks_between(20, 23)
    2
    >>> weeks_between(20, 27)
    2
    >>> weeks_between(5, 5)
    0
    >>> weeks_between(40, 30)
    -1
    """

```

### 5. (Somewhat trickier)

```

def square(n):
    """ (number) -> number

    Returns the square of n.

    >>> square(2)
    4
    >>>
    """

```

In the table below, fill in the Example column by writing square, `square(1)`, and `1` next to the appropriate descriptions.

Description	Example
Parameters	
return value	
Function name	
Function call	

### 6. Write the body of the `square` function from the previous exercise.

# Working with Text

From email readers and web browsers to calendars and games, text plays a central role in computer programs. This chapter introduces a non-numeric data type that represents text, such as the words in this sentence or the sequence of bases in a strand of DNA. Along the way, we will see how to make programs a bit more informative by putting messages in our programs' users and getting input from them.

## 4.1 Creating Strings of Characters

Computers may have been invented to do arithmetic, but these days, most of them spend a lot of their time processing text. Many programs create text, since it's much fun to move it around than to read it.

In Python, text is represented as a string, which is a sequence of characters (letters, digits, and symbols). This type stores continuous sequences of characters (e.g., The characters consist of those from the Latin alphabet found on most North American keyboards, as well as Chinese morphograms, chemical symbols, musical symbols, and much more).

In Python, we indicate that a value is a string by putting either single or double quotes around it. As we will see in Section 4.2, Using Special Characters in Strings, on page 62, single and double quotes are equivalent except for strings that contain quotes. You can use whatever you prefer. (For consistency, the Python style guidelines say that double quotes are preferred.)

Here are four examples:

```
>>> 'Ariane'
'Ariane'
>>> "Isaac Newton"
'Isaac Newton'
```

The opening and closing quotes must match:

```
>>> 'Charles Baudin'
'Charles Baudin'
>>> len('Charles Baudin')
13
```

**SyntaxError: EOL while scanning string literal**

EOL stands for “end of line.” The error occurs because that the end of the line was reached before the end of the string which should be marked with a closing single quote was found.

Strings can contain any number of characters, limited only by computer memory. The shortest string is the empty string containing no characters, `''`.

```
>>> ''
''
>>> len('')
0
>>>
```

## Operations on Strings

Python has a built-in function, `len`, that returns the number of characters between the opening and closing quotes:

```
>>> len('Albert Einstein')
15
>>> len('123')
3
>>> len(' ')
1
>>> len('')
0
```

We can add two strings using the `+` operator, which produces a new string containing the same characters as in the two operands:

```
>>> 'Albert' + ' Einstein'
'Albert Einstein'
```

When `+` has two string operands, it is referred to as the concatenation operator. Operator `+` is probably the most common operator in Python. For fun, let’s apply it to integers, floating-point numbers, and strings, and we’ll apply it to several more types in later chapters.

The following example shows adding an empty string to another string produces a new string that is just like the nonempty operand:

```
>>> "Alan Turing" + ""
'Alan Turing'
>>> "" + "Grace Hopper"
'Grace Hopper'
```

Here is an interesting question: Can operator + be applied to a string and a numeric value? If so, what will happen or concatenated? We'll check it out:

```
>>> 'HI' + 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't concat 'str' object to int
```

This is the second time that we have encountered a type error. The first time, in [Section 3.4, Data Types via Values for Numerical Summation](#), on page 32, the problem was that we didn't pass the right number of parameters to a function. Now, Python took exception to our attempt to combine values of different data types because it didn't know which version of + we want: the one that adds numbers or the one that concatenates strings. Because the first operand was a string, Python expected the second operand to also be a string but instead it was an integer. Now consider this example:

```
>>> 9 + ' planets'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Here, because Python isn't sure, it expected the second operand to also be numeric. The order of the operands affects the error message.

The concatenation operator must be applied to two strings. If you want to join a string with a number, function str can be applied to the number to get a string representation of it and then the concatenation can be done:

```
>>> "Four score and " + str(7) + " years ago"
'Four score and 7 years ago'
```

Function int can be applied to a string whose contents look like an integer, and float can be applied to a string whose contents are numeric:

```
>>> int('8')
8
>>> int('11')
11
>>> float('3.14')
3.14
>>> float('2e-10')
2e-10
>>> float('56.34')
56.34
```

It isn't always possible to get an integer or a floating-point representation of a string, and when an attempt to do so fails, an error occurs:

```
>>> int('a')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'a'
>>> float('b')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: could not convert string to float: 'b'
```

In addition to +, -, \*, /, and %, operator \* can be applied to strings. A string can be repeated using operator \* and an integer, like this:

```
>>> 'AT' * 5
'ATATATA'
>>> 'a' * 5
'aaaaa'
```

If the integer is less than or equal to zero, the operation yields the empty string (or tuple containing no characters):

```
>>> '' * 0
''
>>> 'TATATATA' * -3
''
```

Strings are values, so you can assign a string to a variable. Many mutation operators can be applied to those variables:

```
>>> sequence = 'ATTCGCCCCC'
>>> len(sequence)
16
>>> new_sequence = sequence[:10] + 'GGCCCTCTTC'
>>> new_sequence
'ATTCGCCCCCGGCCCTCTTC'
>>> sequence[2] = 'A'
'ATTCGCCCCCGGCCCTCTTC'
```

## 4.2 Using Special Characters in Strings

Sometimes you want to put a single quote inside a string. If you write it directly, an error occurs:

```
>>> 'that's not going to work'
  File "<stdin>", line 1
    'that's not going to work'
          ^
SyntaxError: invalid syntax
```

When Python encounters the second quote—the one that is intended to be part of the string—it thinks the string is ended. Then it doesn't know what to do with the text that comes after the second quote.

One simple way to fix this is to use double quotes around the string, or you can also put single quotes around a string containing a double quote:

```
said "that's better"
'that's better
said 'She said, "that's better."
She said, "there is home..."'
```

If you need to put a double quote in a string, you can use single quotes around the string. But what if you want to put both kinds of quotes in one string? You could do this:

```
said 'she said, "there is no hard to read."
she said, "there's hard to read..."'
```

The result is a valid Python string. The backslash is called an escape character, and the combination of the backslash and the single quote is called an escape sequence. The name comes from the fact that we're “escaping” from Python's usual syntax rules for a moment. When Python sees a backslash before a string, it means that the next character represents something that Python uses for other purposes, such as marking the end of a string.

The escape sequence “\” is indicated using two symbols but these two symbols represent a single character. The length of an escape sequence is one:

```
said 'backslash'
1
said 'backslash'
```

Python recognizes several escape sequences. In order to see how they are used, we will introduce multiple strings and a few built-in functions first. Here are some common escape sequences:

Escape Sequence	Description
'	Single quote
'	Double quote
\	Backslash
\t	Tab
\n	Newline
\r	Carriage return

Table 4—Escape Sequences

## 4.3 Creating a Multiline String

If you create a string using single or double quotes, the whole string must fit onto a single line.

Here's what happens when you try to print a string across multiple lines:

```
>>> one
  File "c:\temp\test.py", line 1
    'one
      ^
```

**SyntaxError: EOL while scanning string literal**

As we saw in Section 4.2, Creating Strings of Characters on page 60, EOL stands for “end of line” and in this error report, Python is saying that it reached the end of the line before it found the end of the string.

To span multiple lines, put two single quotes or three double quotes around the string instead of one of each. The string can then span as many lines as you want:

```
>>> '''one
... two
... three'''
'one\n\ttwo\n\tthree'
```

Notice that the string Python creates contains a \n sequence everywhere our input started a new line. As programmers, we usually expect newlines in strings. In that last code, Printing Information on page 70 we show that when simple strings are printed, users see the properly rendered strings rather than the escape sequences. That is, they see a line of a quote rather than \n or \t.

### Normalizing Line Endings

In reality, each of the three major operating systems uses a different set of characters to indicate the end of a line. The set of characters used is called a newline. On Unix and Linux, a newline is just a character with ASCII value 10, called a Newline. It is represented as \n. And on Windows, the ends of lines are marked with both carriage return and \n.

Python always uses a single way to indicate a newline, even on operating systems like Windows that do things differently. This is called normalizing the newline. You can do this so that you can write easily, or save programs so that no matter what kind of machine you're running on,

## 4.4 Printing Information

In Section 3.2: Writing and Running a Program on page 56, built-in function `print` was used to print values to the screen. We will use `print` to print messages

to the needs of our program. These messages may include the values that expressions produce and the values that the variables refer to. Here are two examples of printing:

```
>>> print(1 + 1)
2
>>> print("The Latin 'Oryctolagus cuniculus' means 'domestic rabbit'.")
The Latin 'Oryctolagus cuniculus' means 'domestic rabbit'.
```

Some functions don't follow any styling of the output message from Python, like `help`. All output is plain text:

The first function call does what you would expect from the numeric example we have seen previously, but the second does something slightly different than previous string examples: it strips off the quotes around the string and shows us the string's contents rather than its representation. This example makes the difference between the two even clearer:

```
>>> print("In 1859, Charles Darwin revolutionised biology")
In 1859, Charles Darwin revolutionised biology
>>> print("and our understanding of ourselves")
and our understanding of ourselves
>>> print("by publishing "The Origin of Species")
by publishing "The Origin of Species"
```

And the following example shows that when Python prints a string, it prints the values of any escape sequences in the string rather than their individual representations:

```
>>> print('one\nthree\nfive')
one
three
five
```

The example above shows how the backslash can be used to lay values out in columns.

In [Section 6.3, Creating a Multiline String](#), on page 70, we saw that `\n` indicates a new line in multiline strings. When a multiline string is printed, these line sequences are displayed like this:

```
>>> numbers = ['one',
...     'two',
...     'three',
...     'four']
>>> print(numbers)
['one', 'two', 'three', 'four']
```

Function `print` takes a comma-separated list of values to print and prints the values with a single space between them and a newline after the last value:

```
>>> print(1, 2, 3)
```

```
1 2 3
```

```
>>>
```

When called with no arguments, `print` does nothing. This, interestingly, is its next command:

```
>>> print()
```

```
>>>
```

Function `print` can print values of any type, and it can even print values of different types in the same function call:

```
>>> print(1, 'two', 'three', 4.0)
```

```
1 two three 4.0
```

It is also possible to call `print` with an expression as an argument. It will print the value of that expression:

```
>>> radius = 5
```

```
>>> print("The diameter of the circle is", radius * 2, "cm.")
```

```
The diameter of the circle is 10 cm
```

Function `print` has a few extra helpful features. Here is the help documentation for it:

```
>>> help(print)
```

```
Help on built-in function print in module builtins:
```

```
print(...)
```

```
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
```

Prints the values to a stream, or to sys.stdout by default.

**Optional keyword arguments:**

**file:** a file-like object (stream) to write to; the current sys.stdout.

**sep:** string inserted between values; default a space.

**end:** string appended after the last value; default a newline.

**flush:** whether to forcibly flush the stream.

The parameters `sep`, `end`, `file`, and `flush` have assignment statements in the function header. These are called **default parameter values** ‘by default’. If we call function `print` with a comma-separated list of values, the separator is a space; similarly, a newline character appears at the end of every printed string. We won’t discuss `file` and `flush`: they are beyond the scope of this text.

We can supply different values by using keyword arguments. (In the Python documentation, these are often referred to explicitly as *kwargs*.) That's a fancy term for assigning a value to a parameter name in the function call. Here we separate each value with a colon and a space instead of just a space by including `xp=` as an argument:

```
>>> print('a', 'b', sep='<', end='>')
a b >
>>> print('a', 'b', sep='>')
a b >
```

Or, if you'll want to press return/line but not start a new line, do this, using the keyword argument `end=""` to tell Python to end with an empty string instead of a new line:

```
>>> print('a', 'b', sep='', end='')
a b
```

Note, however, that prompt appeared right after the `y`. Typically, you'd need tabs in programs, not in the shell. Here is a program that converts three temperatures from Fahrenheit to Celsius (in) points using keyword arguments:

```
def convert_to_celsius(fahrenheit):
    """Convert fahrenheit to Celsius"""
    celsius = (fahrenheit - 32) * 5 / 9
```

```
>>> convert_to_celsius(32)
27.77777777777778
>>> convert_to_celsius(212)
100.0
>>> convert_to_celsius(50)
10.0
```

```
>>> fahrenheit = 22.0
>>> convert_to_celsius(fahrenheit)
```

```
print('22.0 degrees Fahrenheit are equal to', end=' ')
print(convert_to_celsius(22.0), end=' ')
print(convert_to_celsius(70.0), end=' and ')
print(convert_to_celsius(10.4), end=' Celsius.\n')
```

Here's the output of running this program:

```
>>> fahrenheit = 22.0
>>> convert_to_celsius(fahrenheit)
22.0 degrees Fahrenheit are equal to 7.777777777777778
and 21.1 Celsius.
```

## 4.5 Getting Information from the Keyboard

In the earlier chapter, we explored some built-in functions. Another built-in function that you will find useful is `raw_input`, which reads a single line of text from the keyboard. It returns whatever the user entered as a string, even if it looks like a number:

```
<<< species = input()
Human sapiens.
<<< species.
'Human sapiens.'
<<< population = input()
6973730433
<<< population
'6973730433'
<<< type(population)
<class 'str'>
```

The second and sixth lines of this example, `species` and `6973730433`, were typed by us in response to the calls to function `input`.

If you are expecting the user to enter a number, you must use `int` or `float` to get an integer or a floating point representation of the string:

```
<<< population = input()
6973730433
<<< population
'6973730433'
<<< population = int(population)
<<< population
6973730433
<<< population = population + 1
<<< population
6973730434
```

We then actually need to cast the value that the user has produced before converting it. This time function `int` is called on the result of the call to `input` and is equivalent to the code above:

```
<<< population = int(input())
6973730433
<<< population = population + 1
6973730434
```

Finally, `input` can be given a string argument, which is used to prompt the user for input (notice the space at the end of our prompt):

```
<<< species = input("Please enter a species: ")
Please enter a species. Python curta
<<< print(species)
Python curta
```

## 4.6 Quotes About Strings in This Text

In this chapter, you learned the following:

- Python uses type `str` to represent text or strings of characters.
- Strings are created by placing pairs of single or double quotes around the text. Multi-line strings can be created using matching pairs of triple quotes.
- Special characters like newline and tabs are represented using escape sequences that begin with a backslash.
- Values can be printed using built-in function `print`, and instructions can be issued by the user using built-in function `input`.

## 4.2 Exercises

Here are some exercises for you to try on your own. Solutions are available at <https://proptutorial.ca/2020/07/06/python-programming/>.

1. What value does each of the following expressions evaluate to? Verify your answers by typing the code into the Python shell.
  - a. `'Computer' + 'Science'`
  - b. `'2018' * 3`
  - c. `'100' * 10`
2. Express each of the following phrases as Python strings using the appropriate type of quotation marks (single, double, or triple) and, if necessary, escape sequences. There is more than one correct answer for each of these phrases.
  - a. They'll illuminate during the winter.
  - b. "absolutely not." he said.
  - c. "The salt, (absolutely not!) called Met."
  - d. hydrogen sulfide
  - e.  $\pi$
3. Rewrite the following string using single or double quotes instead of triple quotes:
 

```
    \"\"\"I am a multi-line string.\n    I am\n    composed\n    of three\n    lines.\"\"\"
```
4. Use built-in function `len` to find the length of the empty string.
5. Given variables `x` and `y`, which refer to values 3 and 225 respectively, use function `print` to print the following messages. When numbers appear in the messages, variable `x` and `y` should be used.
 

```
    print("The value of x is", x)
    print("The value of y is", y)
    print("The value of x squared is", x*x)
    print("The value of y squared is", y*y)
    print("The value of x cubed is", x*x*x)
    print("The value of y cubed is", y*y*y)
```

- a. The rabbit is 3.  
 b. The rabbit is 3 years old.  
 c. 12.5 is average.  
 d. 12.5 \* 3  
 e. 12.5 \* 3 is 37.5.

**10. Consider this code:**

```
*** first = 'John'
*** last = 'Doe'
*** print(last + ', ' + first)
```

What is printed by the code above?

- a. It prompts the user for a number, stores the number received as a float in a variable named `num`, and then prints the contents of `num`.  
 b. Complete the examples in the doctesting and then write the body of the following function:

```
def repeat(s, n):
    """(str, int) -> str

    Returns s repeated n times. If n is negative, return the empty string.

    Examples:
    >>> repeat('hi', 2)
    'hihi'
    >>> repeat('hi', 3)
    'hihihi'
    >>> repeat('hi', -1)
    ''
```

- c. Complete the examples in the doctesting and then write the body of the following function:

```
def total_length(s1, s2):
    """(str, str) -> int

    Returns the sum of the lengths of s1 and s2.

    Examples:
    >>> total_length('abc', 'xyz')
    6
    >>> total_length('abc', '')
    3
    >>> total_length('aaaa', 'aaaaaa')
    10
    >>>
```

# Making Choices

This chapter introduces another fundamental concept of programming: making choices. You do this whenever you want your program to behave differently depending on the data it's working with. For example, we might want to do different things depending on whether a solution is valid or bəse, or depending on whether a user types `yes` or `no` in response to a call to built-in function `input`.

We'll introduce statements for making choices in this chapter called `control flow statements` because they control the way the computer executes programs. These statements involve a Python type that is used to represent truth and falsehood. Unlike the `integers`, `floating point numbers`, and `strings` we have already seen, this type has only two values and three operations.

## 5.1 A Boolean Type

In Python, there is a type called `bool` (without an ‘e’). Unlike `int` and `list`, which have billions of possible values, `bool` has only two: `True` and `False`. `True` and `False` are values, just as much as the numbers 3 and -41.5.

### George Boole

In the 1800s, the mathematician George Boole showed that the classical rules of logic could be expressed in purely mathematical terms using only the two values true and false. A century later, Charles Karpinski demonstrated that information theory and Boolean algebra could be used to optimize the design of early mechanical calculators. His work laid the basis for the rise of Boolean logic and digital computers.

In honor of Boole, we’ll need code in programming languages to type-check whether a logical expression like `x > 10` and `also > 100`.

## Boolean Operators

There are only three basic Boolean operators: `and`, `or`, and `not`. `or` has the highest precedence, followed by `and`, followed by `not`.

`not` is a unary operator. It is applied to just one value. Like the negation in the expression `(7 + 7)`. An expression involving `or` produces `True` if the original value is `True`, and it produces `False` if the original value is `False`:

```
not not True
```

```
False
```

```
not not False
```

```
True
```

In the previous example, instead of `not True`, we could simply use `False`, and instead of `not False`, we could use `True`. Rather than apply `not` directly to a Boolean value, we would typically apply `not` to a Boolean variable in a more complex Boolean expression. The values given for the following examples of the Boolean operators, `and` and `or`, are although we apply them to Boolean constants. In the following examples, we'll give an example of how they are typically used at the end of this section.

`and` is a binary operator; the expression `left and right` produces `True` if both `left` and `right` are `True`, and it produces `False` otherwise:

```
True and True
```

```
True
```

```
True and False
```

```
False
```

```
False and True
```

```
False
```

```
False and False
```

```
False
```

```
False or True
```

```
True
```

```
True or False
```

```
True
```

```
True or True
```

```
True
```

```
False or False
```

```
False
```

```
True or False
```

```
True
```

```
False or True
```

```
True
```

`or` is also a binary operator. It produces `True` if either operand is `True`, and it produces `False` only if both are `False`:

```
True or True
```

```
True
```

```
True or False
```

```
True
```

```
False or True
```

```
True
```

```
False or False
```

```
False
```

probably don't care that you ever have both. Unlike English (but like most programming languages), Python always interprets `or` as inclusive.

## Building an Exclusive Or Expression

If you were an architect or even need to build a "not-in-between" logic, you'll want to understand the `exclusive-or` expression.

Let's say you have two Boolean variables, `b1` and `b2`, and you want an expression that, no matter if one and only one of `b1` or `b2` is `True`. Combination of `not` and `or` will translate `True or b2 or not b1 and b2 is False`. But since `not b1 and b2` will combine `True if b1 is True and b2 is False`.

It's not possible to build these expressions by putting the `Not`, `Or`, and `And` cells `True` or both. Both cells expressions will combine to form `True` and therefore combine the two expressions with an `Or`:

```
not b1 = False
not b2 = False
not (b1 and not b2 or b2 and not b1)
False
not b1 = False
not b2 = True
not (b1 and not b2 or b2 and not b1)
True
not b1 = True
not b2 = False
not (b1 and not b2 or b2 and not b1)
False
not b1 = True
not b2 = True
not (b1 and not b2 or b2 and not b1)
False
```

In a real trace, we'd see a much simpler solution:

We mentioned earlier that Boolean operators are usually applied to Boolean expressions rather than Boolean constants. If we want to express "It is not cold and windy" using own variables, `cold` and `windy`, that refer to Boolean values, we first have to decide what the ambiguous English expression means: Is it not cold but at the same time windy, or is it both not cold and not windy? A truth table for each alternative is shown in [Table 3: Boolean Operators](#), on page 80, and the following code snippet shows what they look like translated from Python:

```
not cold = True
not windy = False
not (not cold and windy)
False
not (not (cold and windy))
True
```

and	or	not and	not or	not and and	not or or and
True	True	True	True	False	False
True	False	False	True	False	True
False	True	True	False	True	True
False	False	False	False	False	True

TABLE 3—Boolean Operators

### Boolean Operators in Other Languages

If you already know another language such as C or C++, you might be used to seeing `||` for or, `|||` for and, and `!|||` for not. These won't work in Python, but the logic is the same.

### Relational Operators

We said earlier that `True` and `False` are values. Typically those values are not written down directly in expressions, but rather created by expressions. The most common way to do that is to insert a comparison using a relational operator. For example, `5 < 6` is a comparison using the relational operator `<` that generates the value `True`, while `15 > 77` uses `>` and generates the value `False`.

As shown in Table 4, Relational and Equality Operators, Python has all the operators you're used to using. Some of them are represented using two characters instead of one, like `<=` instead of `<`.

Symbol	Operation
<code>&lt;</code>	Greater than
<code>&lt;=</code>	Less than
<code>&gt;</code>	Greater than or equal to
<code>&gt;=</code>	Less than or equal to
<code>=</code>	Equal to
<code>!=</code>	Not equal to

TABLE 4—Relational and Equality Operators

The most important programming rule is that Python uses `==` for equality instead of just `=`. In other words, `=` is used for assignment. Avoid writing `c = d` when you mean to check whether variable `c` is equal to `d`.

All relational operators are binary operators; they compare two values and produce True or False as appropriate. The greater-than (>) and less-than (<) operators work as follows:

```
>>> 45 > 24
```

**True**

```
>>> 45 > 79
```

**False**

```
>>> 45 < 79
```

**True**

```
>>> 45 < 24
```

**False**

We can compare integers to floating-point numbers with any of the relational operators. Integers are automatically converted to floating point when we do this, just as they are when we add them to floats:

```
>>> 23.1 == 23
```

**True**

```
>>> 23.1 == 23.1
```

**True**

```
>>> 23.1 == 23.1
```

**True**

```
>>> 23.1 == 23
```

**False**

The seven built-in “equality” and “not-equality”:

```
>>> 45.0 == 45
```

**False**

```
>>> 67.0 == 67
```

**False**

```
>>> 67.0 == 67
```

**True**

```
>>> 67.0 != 67
```

**False**

```
>>> 67.0 != 23
```

**True**

Of course, it doesn’t make much sense to compare two numbers that you know in advance, since you would also know the result of the comparison. Relational operators return simple Boolean variables, True or False:

```
>>> def is_positive():
...     """ (number) --> bool
```

```
>>>
```

```
>>> is_positive(10)
```

```
>>>
```

```
>>> is_positive(2)
```

```
>>> True
```

```

... >>> x < 0 positive(-4.6)
False
...
... >>> minimum(3, 5, 0)
3
...
... >>> x < 0 positive(2)
True
... >>> x < 0 positive(-4.6)
False
... >>> x < 0 positive(0)
False

```

In the doctesting above, we use the acronym “**T/F**,” which stands for “True and False.” An equivalent phrase is “*Exactly when*.” This type construct states that the function will return **True** if the preceding descriptive condition holds under which the will be returned. It is implied that when those conditions aren’t met the function will return **False**.

We can now write our **anagram** expression from [Anagrams on Condition Expressions](#) on page 79, much more simply:

```
def f(x, y):
```

Evaluate or **return** that exactly one of **x** and **y** has to be **True**. **False** is **True**, **x** isn’t **True**, and vice versa.

## Combining Comparisons

We have now seen three types of operators: arithmetic (**+**, **-**, and **\***); and Boolean (**not**, **and**, and **or**), plus relational (**<**, **>**, **==**, and so on).

There are two rules for combining them:

- Arithmetic operators have higher precedence than relational operators. For example, **+** and **\*** are evaluated before **<** or **>**.
- Boolean operators have higher precedence than relational operators. For example, comparisons are evaluated before **and**, **or**, and **not**.
- All relational operators have the same precedence.

These rules mean that the expression **1 + 3 > 7** is evaluated as **(1 + 3) > 7**, not as **1 + (3 > 7)**. These rules also mean that you can often skip the parentheses in complicated expressions:

```

>>> x == 2
True
>>> y == 3
False
>>> x == True
True
>>> x == y and y != x
True

```

It's usually a good idea to put the parentheses in, though, since it helps the eye find the subexpressions and clearly communicate the order to anyone reading your code:

```
>>> x = 5
>>> y = 10
>>> x < 20
>>> (x < y) and (y < x)
True
```

It's very common in math notation to check whether a value lies in a certain range, to other words, that it is between two other values. You can do this in Python by combining the comparisons with and:

```
>>> x = 2
>>> (1 < x) and (x < 5)
True
>>> x = 7
>>> (1 < x) and (x <= 5)
False
```

This comes up so often, however, that Python lets you chain the comparisons:

```
>>> x = 5
>>> 1 < x <= 5
True
```

Most comparisons work as you would expect, but there are cases that may startle you:

```
>>> 3 < 5 != True
True
>>> 3 < 5 != False
True
```

In some, impossible, the both of these expressions make True. However, the first one is equivalent to this:

```
(2 < 5) and (5 < 1) == True
```

while the second is equivalent to this:

```
(3 < 5) and (5 < False)
```

Since 5 is neither True nor False, the second half of each expression is True, so the expression makes a whole True as well.

This kind of expression is an example of something that is a bad idea even though it is legal. We strongly recommend that you only chain comparisons in ways that would seem natural in a mathematician's—*to other words*, that you use < and <= together, or > and >= together, and nothing else. If you feel

the computer to do something else. Instead, use simple comparisons and combine them with and in order to keep your code readable. It's also a good idea to use parentheses whenever you think the expression you are writing may not be entirely clear.

### Using Numbers and Strings with Boolean Operators

We have learned how to use and, or, and not with strings and numbers used in an expression involving a three-digit number. Along the same lines, numbers and strings can be used with Boolean operators. Python does not mind it, and treats all other numbers as True:

```
num = 3
if num:
    print("num is not zero")
else:
    print("num is zero")
```

Similarly, the code string below is True, and all other strings are treated as False:

```
str = " "
if str:
    print("str is not empty")
else:
    print("str is empty")
```

As we have tried to stress so much, you should only use Boolean operators on their natural types.

### Short-Circuit Evaluation

When Python evaluates an expression containing and or or, it does so from left to right. As soon as it knows enough to stop evaluating, it stops. Even if some operators haven't been looked at yet. This is called short-circuit evaluation.

In our example, if the first operand is True, we know that the expression is True. Python knows this as well, so it doesn't even evaluate the second operand. Similarly, in our expression if the first operand is False, we know that the expression is False. Python knows this as well, and the second operand isn't evaluated.

To demonstrate this, we use an expression that results in an error:

```
num = 1 / 0
print(num)
File "test.py", line 3, in <module>
    print(num)
  File "test.py", line 1, in <lambda>
    num = 1 / 0
ZeroDivisionError: division by zero
```

We now see that `==` is the second operand to `=`:

```
>>> 12 < 21 or 11 > 0
```

**True**

Since the first operand products `True`, the second operand isn't evaluated, so the interpreter never actually tries to think anything by zeros.

Of course, if the first operand turns out to false, the second operand must be evaluated. The second operand also needs to be evaluated when the first operand is null and is `True`.

## Comparing Strings

It's possible to compare strings just as you would compare numbers. The characters in strings are represented by integers: a capital A, for example, is represented by 65, while a square is 39, and a lowercase z is 122. This encoding is called ASCII, which stands for "American Standard Code for Information Interchange". One of its quirks is that all the uppercase letters come before all the lowercase letters, so a capital Z is less than a small c.

One of the most common reasons to compare two strings is to decide which one comes first alphabetically. This is often referred to as Python's `lexicographical ordering`. To see which string is greater than which by comparing corresponding characters from left to right, if the character from one string is greater than the character from the other, the first string is greater than the second. If all the characters are the same, the two strings are equal; if one string runs out of characters with the comparison is being done (in other words, is shorter than the other), then it is less. The following table fragment shows a few comparisons in action:

```
>>> 'A' < 'B'
```

**True**

```
>>> 'A' < 'Z'
```

**False**

```
>>> 'ab' < 'abc'
```

**True**

```
>>> 'abc' < 'abcd'
```

**False**

In addition to operators that compare strings lexicographically, Python provides an operator that checks whether one string appears inside another (the `in` operator):

```
>>> 'Dan' in '01 Jun 1838'
```

**True**

```
>>> 'Pete' in '01 Jun 1838'
```

**False**

Using this idea, we can prompt the user for a date in this format and report whether that date is a Sunday:

```
my_date = input('Enter a date in the format DD-MMM-YYYY: ')
Enter a date in the format DD-MMM-YYYY: 24-Feb-2028
my_date[6] == date
False
my_date = input('Enter a date in the format DD-MMM-YYYY: ')
Enter a date in the format DD-MMM-YYYY: 21-Jun-2028
my_date[6] == date
True
```

The `in` operator produces `True` exactly when the last string appears in the second string. This is case sensitive:

```
my_cat in 'cat'
True
my_cat in 'dog'
False
```

The `empty` string is a way of matching on every string:

```
'' in 'abc'
True
'' in ''
True
```

The `in` operator also applies to other types. You'll see examples of this in [Chapter 8, Storing Collections of Data Using Lists](#), on page 128, and in [Chapter 11, Summary Data Using Other Collection Types](#), on page 199.

## 5.2 Choosing Which Statements to Execute

An `if` statement lets you change how your program behaves based on a condition. The general form of an `if` statement is as follows:

```
if <condition>:
    <block>
```

The condition is an expression such as `x == 'Monday'` or `y > 5`. Note that this doesn't have to be a Boolean expression. As we discussed in [Using NumPy and Matplotlib](#) (page 114), non-Boolean values are treated as `True` or `False` when required.)

As with function bodies, the block of statements inside an `if` must be indented. As a reminder, the standard indentation for Python is four spaces.

If the condition is `True`, the statements in the block are executed; otherwise, they are not. As with functions, the block of statements must be indented to show that it belongs to the `if` statement. If you don't indent properly, Python

intentional or otherwise, or whether, might happily exceed the task that you were to do something you didn't intend because some statements were not indented properly. We'll briefly explore such problems in this chapter.

Here is a table of solution categories based on pH level:

pH Level	Solution Category
0–1	Strong acid
7	Neutral
9–14	Weak base
10–14	Strong base

We can use an `if` statement to print a message only when the pH level given by the program's user is acidic:

```
>>> ph = float(input('Enter the pH level: '))
Enter the pH level: 6.6
>>> if ph < 7.0:
...     print(ph, "is acidic.")
...
6.6 is acidic.
```

Recall from [Section 4.5: Getting Information from the Keyboard](#), on page 70, that we have to convert user input from a string to a floating-point number before doing the comparison. Also, here we are providing a prompt for the user by passing a string into `input`. Python prints this string so the user knows what information to type.

If the condition is false, the statements in the block won't execute:

```
>>> ph = float(input('Enter the pH level: '))
Enter the pH level: 8.8
>>> if ph < 7.0:
...     print(ph, "is acidic.")
...

```

If we don't indent the block, Python has some trouble:

```
>>> ph = float(input('Enter the pH level: '))
Enter the pH level: 6
>>> if ph < 7.0:
...     print(ph, "is acidic.")
    File "stdin", line 2
        print(ph, "is acidic."
               ^
IndentationError: expected an indented block
```

Now we're using a block, which has multiple statements that are executed only if the condition is true:

```
ph = float(input("Enter the pH level: "))
Enter the pH level: 6.8
if ph < 7.0:
    ... print(ph, "is acidic.")
    ... print("You should be careful with that!")
    ...
6.8 is acidic.
You should be careful with that!
```

When we assign the first line of the block, the Python interpreter continues to expect to ... until the end of the block, which is signified by a blank line:

```
ph = float(input("Enter the pH level: "))
Enter the pH level: 8.8
if ph < 7.0:
    ... print(ph, "is acidic.")
    ...
    ... print("You should be careful with that!")
You should be careful with that!
```

If we don't include the extra blank line in the block, the interpreter complains:

```
ph = float(input("Enter the pH level: "))
Enter the pH level: 8.8
if ph < 7.0:
    ... print(ph, "is acidic.")
    ... print("You should be careful with that!")
    Pale yellow - law 3
    print("You should be careful with that!")
```

### **SyntaxError: unindent does not match visual indent**

If the program is in a file, then no blank line is needed. As soon as the interpreter sees ends, Python assumes that the block has ended as well. This is therefore legal:

```
ph = 8.8
if ph < 7.0:
    ... print(ph, "is acidic.")
    ... print("You should be careful with that!")
```

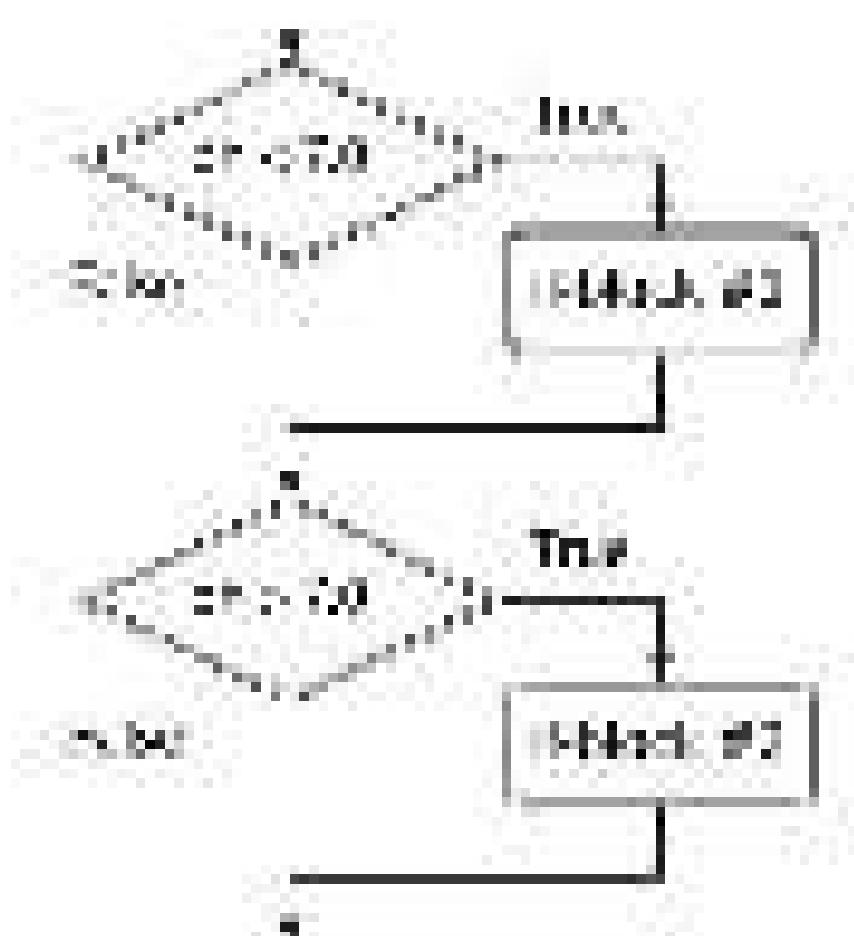
In practice, this slight inconsistency is never a problem, and most people won't even notice it.

Of course, sometimes there are situations where a single decision isn't sufficient. If there are multiple criteria to examine, there are a couple of ways to handle it. One way is to use multiple `if` statements. For example, we might

What different outcomes do you think will happen if we had a pH level of exactly 7, then the neutral case our code won't print anything:

```
>>> ph = float(input('Enter the pH level: '))
Enter the pH level: 8.5
>>> if ph < 7.0:
...     print(ph, "is acidic.")
...
>>> if ph > 7.0:
...     print(ph, "is basic.")
...
8.5 is basic.
```

Here's a flowchart that shows how Python executes the `if` statements. The diamonds are conditions, and the arrows indicate what path to take depending on the result of evaluating those conditions:

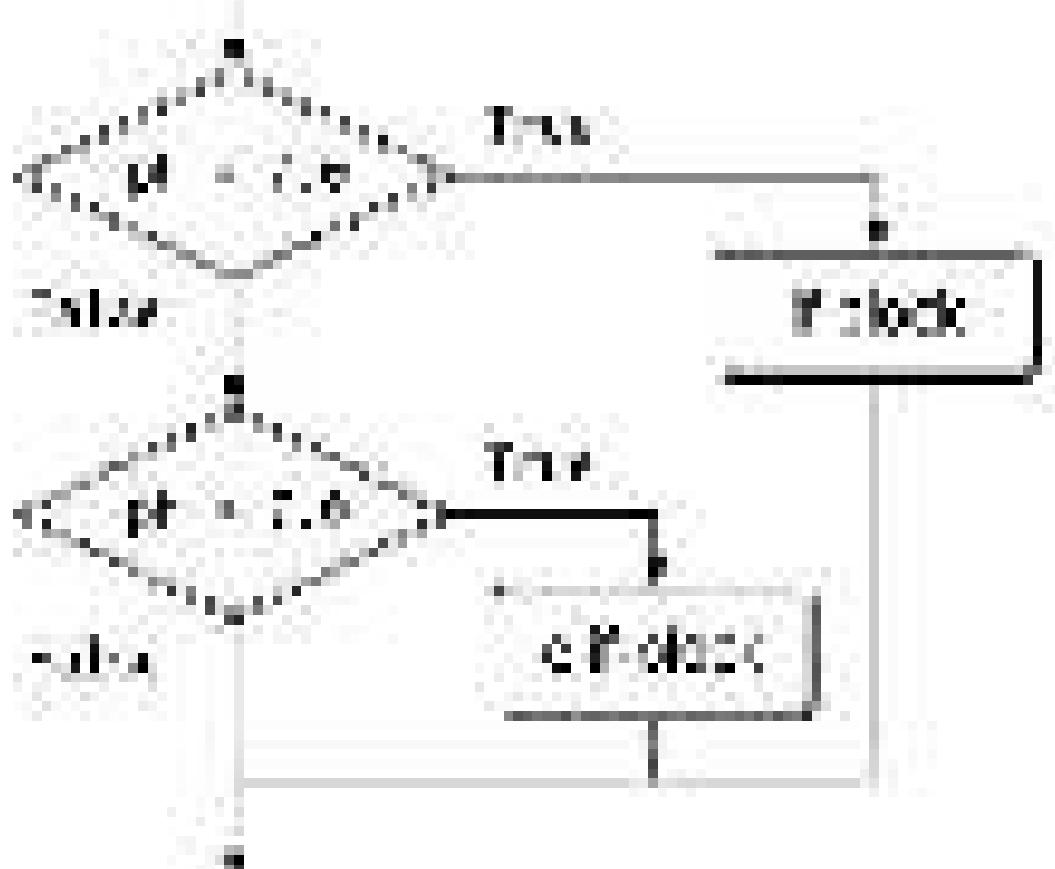


Note that both conditions are always evaluated, even though we know that only one of the blocks can be executed!

We can merge both cases by adding another condition/label pair using the `elif` keyword (which stands for “else if”); each condition/block pair is called a **clause**:

```
>>> ph = float(input('Enter the pH level: '))
Enter the pH level: 6.5
>>> if ph < 7.0:
...     print(ph, "is acidic.")
... elif ph > 7.0:
...     print(ph, "is basic.")
...
6.5 is acidic.
```

The difference between the two is that it is checked only when the condition above it evaluates to true. Here is a flow chart for this code:



This flow chart shows that if the first condition evaluates to true, the second condition is skipped.

If the pH is exactly 7.0, neither cause matches, so nothing is printed:

```

>>> ph = float(input('Enter the pH level: '))
Enter the pH level: 7.0
>>> if ph < 7.0:
...     print(ph, "is acidic.")
... elif ph >= 7.0:
...     print(ph, "is basic.")
...
>>>

```

With the pH example, we accomplished the same thing with two statements as we did with one if block.

This is not always the case; for example, if the body of the `if` changes the value of a variable used in the second condition, they are not equivalent. Here is the version with two `if`'s:

```

>>> ph = float(input('Enter the pH level: '))
Enter the pH level: 6.8
>>> if ph < 7.0:
...     ph = 5.0
...
>>> if ph >= 7.0:
...     print(ph, "is acidic.")

```

`x = 5.0`

And here is the version with an `if/else`:

```

>>> ph = float(input("Enter the pH level: "))
Enter the pH level: 8.0
>>> if ph < 7.0:
...     ph = 8.0
>>> elif ph > 7.0:
...     print(ph, "is acidic.")
...
>>>

```

To a series of `if`s, if two conditions are related, use `elif` instead of two `if`s.

An `I` statement can be followed by multiple `* If clauses`. This longer example translates a chemical formula into English:

```

>>> compound = input("Enter the compound: ")
Enter the compound: CH4
>>> if compound == "H2O":
...     print("Water")
... elif compound == "HCl":
...     print("Hydrochloric acid")
... elif compound == "CH4":
...     print("Methane")
...
>>>

```

**Methane**

>>>

As we saw in the [code on page 89](#), if none of the conditions in a chain of `If`-`else` statements are satisfied, Python does not execute any of the associated blocks. This isn't always what we'd like, though. In our translation example, we probably want our program to print something even if it doesn't recognize the compound.

To do this, we add an `else` clause at the end of the chain:

```

>>> compound = input("Enter the compound: ")
Enter the compound: H2O
>>> if compound == "H2O":
...     print("Water")
... elif compound == "HCl":
...     print("Hydrochloric acid")
... elif compound == "CH4":
...     print("Methane")
...
>>>

```

**Unknown compound**

An `If` statement can have at most one `else` clause, and it has to be the final clause in the statement. Notice here how condition `compound == " "`

```
if condition:
    statement
else:
    statement
```

Equality, just count in the same as this code (except that the condition is evaluated only once in the first form but twice in the second form):

```
if condition:
    statement
if not condition:
    statement
```

## 5.3 Nested If Statements

An if statement can contain any type of Python statement, which implies that it can include other if statements. An if statement inside another is called a *nested if statement*:

```
value = input("Enter the pH level: ")
if testvalue < p:
    ph = float(value)
    if ph < 7.0:
        print(ph, "is acidic")
    elif ph > 7.0:
        print(ph, "is basic")
    else:
        print(ph, "is neutral")
else:
    print("No pH value was given!")
```

In this case, we ask the user to provide a pH value, which will initially appear as a string. The first, or outer, if statement checks whether the user typed something, which determines whether we compare the value of pH with the next if statement. (If the user didn't enter a number, then float will raise an error.)

Nested if statements are sometimes necessary, but they can get complicated and difficult to understand. To describe when a statement is evaluated, we have to mentally evaluate the if lines; for example, the statement `print("car is fast")` executes only if the length of the string that `car` refers to is greater than 5, and `car` is also evaluated to be `car` (meaning the user entered a `car`).

## 5.4 Remembering the Results of a Boolean Expression Evaluation

Take a look at the following line of code and guess what value is stored in `x`:

```
x = 15 > 5
```

If you said this, you were right! It is similar to how the relational operators do, but, and since that's a value like any other, it can be assigned to a variable.

The most common situation in which you would want to do this comes up when translating decision tables into software. For example, suppose you want to calculate someone's risk of heart disease using the following table based on age and body mass index (BMI):

		Age			
		<45	>=45		
BMI	<22.0	Low	Medium		
	=22.0	Moderate	High		

One way to implement this would be to use nested if statements:

```
if age < 45
    if bmi < 22.0
        risk = 'low'
    else
        risk = 'medium'
else
    if bmi < 22.0
        risk = 'medium'
    else
        risk = 'high'
```

The expression for `< 22.0` is used multiple times. To simplify this code, we can evaluate each of the Boolean expressions once, create variables that refer to the values provided by those expressions, and use those variables multiple times:

```
young = age < 45
slim = bmi < 22.0
if young:
    if slim:
        risk = 'low'
    else:
        risk = 'medium'
else:
    if slim:
        risk = 'medium'
    else:
        risk = 'high'
```

We could also write this without nesting as follows:

```

young = age < 45
old = age >= 65
if young and old:
    risk = 'low'
elif young and not old:
    risk = 'medium'
elif not young and old:
    risk = 'medium'
else not young and not old:
    risk = 'high'

```

## 5.5 You Learned About Booleans: True or False?

In this chapter, you learned the following:

- Python uses Boolean values, `True` and `False`, to represent what is true and what isn't. Programs can combine these values using three operators: `and`, `or`, and `not`.
- Relational operators can return a Boolean (a numeric) value. 0, 0.0, the empty string, and `None` are `False`; all other numeric values and strings are treated as `True`. It is best to avoid applying Boolean operators to non-Boolean values.
- Relational operators such as “`equivalent`” and “`less than`” compare values and produce a Boolean result.
- When different operators are combined in an expression, the order of precedence from highest to lowest is `not`, arithmetic, relational, and then `and` or `or`.
- Enclosures control the flow of execution. As with function definitions, the bodies of if statements are indented to set the bodies off from the clauses.

## 5.6 Exercises

Now try some exercises for you to try on your own. Solutions are available at <https://tinyurl.com/peguo5-exercises>.

1. What value does each expression produce? Verify your answers by typing the expressions into Python.
  - `True and not False`
  - `True and not False` (Notice the capitalization.)
  - `True or True and False`
  - `not True or not False`
  - `True and 0.1`

1. `5 < 5.5`
- a. `1.1 < 0.512`
- b. `2 < -4.0`
2. Variables `a` and `b` refer to Boolean values.
- Write an expression that produces `True` if both variables are `True`.
  - Write an expression that produces `False` if both variables are `False`.
  - Write an expression that produces `True` if at least one of the variables is `True`.
3. Variables `t`, `l` and `m` refer to Boolean values. Write an expression that produces `True` if at most one of the variables is `True`.
4. You want an automatic wildlife camera to switch on if the light level is less than 0.01 lux or if the temperature is above freezing, but not if both conditions are true. (You should assume that function `turn_on()` has already been defined.)
- Your first attempt to write this is as follows:
- ```
if (light < 0.01) or (temperature > 0.0):
    print("turn_camera_on")
```
- A friend says that this is not exclusive or and that you could write it more simply as follows:
- ```
if (light < 0.01) == (temperature > 0.0):
    turn_camera_on()
```
- Is your friend right? If so, explain why. If not, give values for `light` and `temperature` that will produce different results for the two fragments of code.
5. In Section 3.1, Exercises That Python Practices, on page 31, we saw built-in function `abs`. Variable `x` refers to a number. Write an expression that evaluates to `True` if `x` and its absolute value are equal and evaluates to `False` if, however, `x` is assigned the resulting value in a variable named `result`.
6. Write a function named `diff(x)` that has two parameters, `a` and `b`. The function should return `True` if `a` and `b` refer to different values and should return `False` otherwise.
7. Variables `population` and `land_sq_kilometre` refer to floats.
- Write an `if` statement that will print the population if it is less than 10,000,000.

- b. Write an if statement that will print the population if it is between 10,000,000 and 20,000,000.
- c. Write an if statement that will print "Densely populated" if the land density (number of people per unit of area) is greater than 100.
- d. Write an if statement that will print "Densely populated" if the land density (number of people per unit of area) is greater than 100, and "Sparse population" otherwise.
8. Function conversion takes from section 3.8, Defining Our Own Functions, on page 85, converts from Fahrenheit to Celsius. Wikipedia, however, discusses eight temperature scales: Kelvin, Celsius, Fahrenheit, Rankine,华氏度, Réamur, Réamur, and Reamur. Visit [http://en.wikipedia.org/wiki/Conversion\\_of\\_temperature\\_scales](http://en.wikipedia.org/wiki/Conversion_of_temperature_scales) to read about them.
- a. Write a `convert_temperature` function to convert temperature from one scale to another. Unlike `kelvin` and `celsius` are each one of "Kelvin", "Celsius", "Fahrenheit", "Rankine", "华氏度", "Réamur", "Réamur", and "Reamur" units.
- Hint:** On the Wikipedia page there are eight tables, each with two columns and seven rows. This translates to an awful lot of statements—at least  $\sim 7 \times 7 = 49$ —each of the eight units can be converted to the seven other units. Possibly even worse, if you decided to add another temperature scale, you would need to add at least sixteen more if statements: eight to convert from your new scale to each of the existing ones and eight to convert from the existing ones to your new scale.
- b. Now if you added a new temperature scale, how many if statements would you need to add?
9. Assume we want to print a warning message if a pH value is below 3.0 and otherwise simply report on the acidity. We try this if statement:
- ```
*** ph = 2
*** ph < 7.0;
    -- print(ph, "is acidic")
    -- else ph > 3.0;
        -- print(ph, "is NOT acidic Be careful!")
    --
```

2. Is it valid?

This prints the wrong message: when a pH of 2 is entered. What is the problem, and how can you fix it?

10. The following code displays a message(s) about the acidity of a solution:

```
an = float(input("Enter the pH level: "))
if an < 7.0:
    print("It's acidic!")
elif an < 4.0:
    print("It's a strong acid!")
```

- a. What message(s) are displayed when the user enters 5.4?
  - b. What message(s) are displayed when the user enters 3.6?
  - c. Make a small change to one line of the code so that both messages are displayed when a value less than 4 is entered.
11. Why does the last example in Section 3.1, Remembering the Results of a Boolean Expression Evaluation on page 89, check to see whether someone is light (0 kg or just past the limit) or less than the threshold value than heavy? If you wanted to write the second assignment statement as `mass = between 200`, what change(s) would you have to make to the code?

СНАРУЖКА

# A Modular Approach to Program Organization

Mathematicians don't prove every theorem from scratch. Instead, they build their proofs on the truth of their predecessors' work: already tested (check) in the same way. It's much harder for someone to write all of a program alone; it's much more common and productive to make use of the millions of lines of code that other programmers have written before.

## What Happens When You Import This?

A module is a collection of variables and functions that are grouped together in a single file. The variables and functions in a module are usually related to one another in some way; for example, module `math` contains the variable `pi` and mathematical functions such as `sin`, `cosine` and `sqrt` (square root). This chapter shows you how to use some of the hundreds of modules that come with Python, as well as how to create your own modules.

## 6.1 Importing Modules

To gain access to the variables and functions from a module, you have to import it. To tell Python that you want to use functions in module `math` for example, you use this `import` statement:

```
>>> import math
```

Importing a module creates a new variable with that name. That variable refers to an object whose type is `module`:

```
>>> type(math)
>>> math
```

Once you have imported a module, you can use module functions to see what it contains. Here is the first part of the help output:

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

**Help on module math:**

**NAME:**

`math`

**URL:** <http://docs.python.org/3/library/math.html>

<http://docs.python.org/3/library/math.html>

The following documentation is automatically generated from the Python source files. It may be incomplete because it reflects features that are considered implementation detail and may vary across different implementations. When in doubt, consult the module reference or the location listed above.

### DESCRIPTION

This module is always available. It provides access to the mathematical functions defined by the C standard.

### FUNCTIONS:

```
acos(...)  
asinh(...)
```

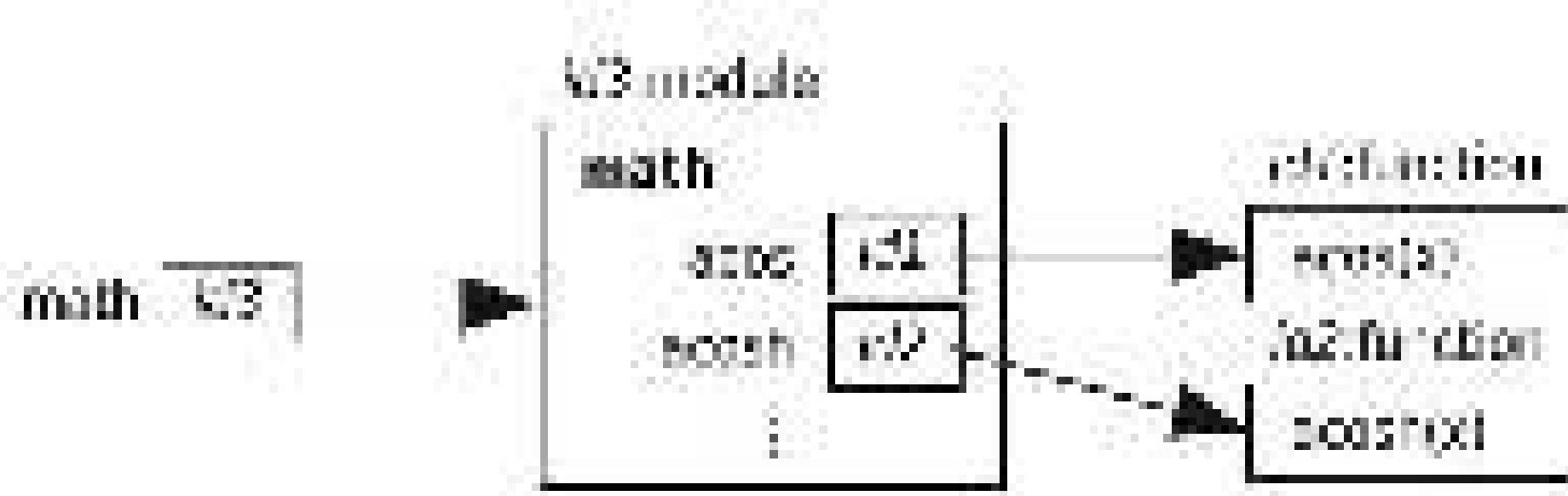
*Return the arc cosine (measured in radians) of x.*

```
>>> import math
>>> math.pi
```

return the hyperbolic arc cosine (measured in radians) of  $x$ .

[Lots of other functions, not shown here.]

The statement `import math` creates a variable called `math` that refers to a module object. In that object are all the names defined in that module. Each of them refers to a function object:



Great—our program can now use all the standard mathematical functions. When we try to calculate a square root, though, we get an error telling us the Python is still unable to find function `sqrt`:

```
**>>> sqrt(9)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'sqrt' is not defined
```

The solution is to tell Python explicitly to look for the function in module `math` by combining the module's name with the function's name using a dot:

```
**>>> math.sqrt(9)
3.0
```

The dot is an operator, just like `+` and `*` are operators. Its meaning is “look up the object that the variable to the left of the dot refers to and, in that object, find the name that occurs to the right of the dot.” In `math.sqrt()`, Python finds `sqrt` in the current namespace, looks up the module object that `math` refers to, finds function `sqrt` in that module, and then executes the function call following the standard rules described in [Section 8.3, “Defining Functions: Calls to the Memory Model”](#) on page 40.

Modules can contain more than just functions. Module `math`, for example, also defines some variables like `pi`. Once the module has been imported, you can use these variables like any others:

```
>>> import math
>>> math.pi
3.141592653589793
>>> radius = 5
>>> print('area is', math.pi * radius ** 2)
area is 78.54074346060835
```

You can even modify the variables imported from modules:

```
>>> import math
>>> math.pi = 3
>>> radius = 5
>>> print('area is', math.pi * radius ** 2)
area is 75
```

**Don't do this!** Changing the value of a `const` is a bad idea. In fact, this is such a bad idea that many languages allow programmers to define `const`s as immutable constants as well as variables. As the name suggests, the value of a `const` cannot be changed after it has been defined—it is always 3.14159 and a little bit more (around 3.141592653589793). This kind of discipline doesn't allow programmers to "freeze" values like this because the language is few significant digits.

Combining the `math`'s name with the names of the things it contains is safe, but it isn't always convenient. For this reason, Python lets you specify exactly what you want to import from a module, like this:

```
>>> from math import pi, sqrt
>>> sqrt(9)
3.0
>>> radius = 5
>>> print('circumference is', 2 * pi * radius)
circumference is 31.41592653589793
```

This doesn't introduce a variable called `math`. Instead, it creates function and variable `pi` in the current namespace, as if you had typed the function definition and variable assignment yourself. Run your shell and try this:

```
>>> from math import sqrt, pi
>>> math.sqrt(9)
Traceback (most recent call last):
  File <ipython-input-12>, line 1, in <module>
    math.sqrt(9)
NameError: name 'math' is not defined
>>> sqrt(9)
3.0
```

Here, we don't have a variable called `math`. Instead, we imported variables `pi` and `sqrt` directly into the current namespace, as shown in this diagram:

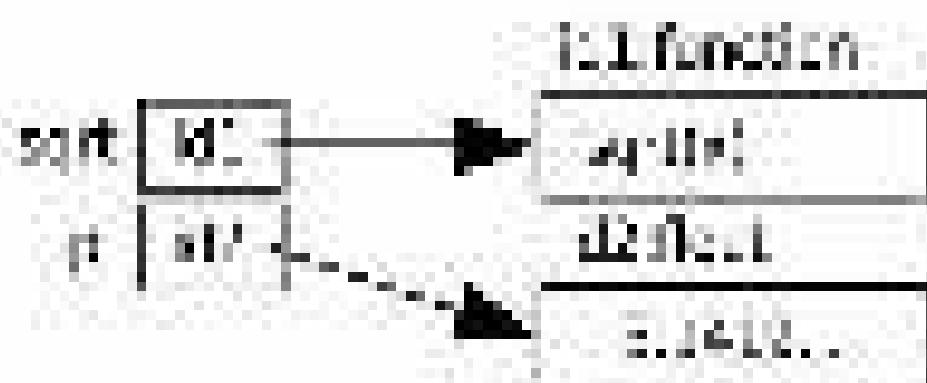
## Restoring a Module

If you change the value of a variable or function from an imported module, you can never find and reimport the module to recover its original value. In this, you can restore an object by closing the shell and restarting.

Whether having to restart the shell, you can remove the module with the `del` keyword or `delattr` method:

```
>>> import math
>>> math.pi = 3
>>> math.pi
3
>>> del math.pi
>>> math.pi
math.pi: name error
>>> math.pi
3.141592653589793
>>>
```

Don't forget to save before exiting the module!



This can lead to problems when different modules provide functions that have the same name. If you import a function called `cos` from a module called `math` and then you import another function called `cos` from the `operator` module, the second replaces the first. It's exactly like assigning one value to a variable and then assigning another value: the most recent assignment overwrites.

This is why it's usually not a good idea to just import whole strings by everything from the module at once:

```
>>> from math import *
>>> print(sqrt(16))
2.88675134746068
```

Although most of us are aware of this, you run the risk of your program overwriting the intended function and not working properly.

The standard Python library contains several hundred modules to do everything from figuring out what day of the week it is to retrieving data from a website. The full list is available at <http://www.python.org/releasenotes/2.7/library.html>; although it's far too much to absorb in one sitting (or even one course),

knowledge base to meet the theory with respect of the family that communication  
should be used in a systematic manner.

## Module Culture

Python's built-in `locals()` object is a dictionary formed at the top level under whose keys and values will be all the global variables defined and used in the same script. As part of Python, you'll see this conversion now play in other modules as well. You can now check on the variable using `locals()` or `sys.modules`, as several modules and variables are available, however the `locals()` which you've either imported or used.

As at Putter 222, 43 of the 143 items in Table 2 are used to signal errors or particular kinds of problems. In fact, 22 and 24 of these 23 words, meanings, and expressions types like *failure*, *error*, *error*. Test markers follow a number, punctuation in which the first letter of each word is uppercase.

Methodology of the study's quantitative research

## 6.2 Defining Your Own Modules

Section 3.7, Poetry and Rhythm of Prophets on page 78, explained that In  
order to have such fine literature, you can put it in a file with a .txt extension.

and it demonstrated how to run that code. Chapter 8, More Variables and Functions, on page 34, also included this function definition:

```
def convert_to_celsius(fahrenheit):
    """ Convert fahrenheit to Celsius.

    Return the number of fahrenheit degrees converted to celsius degrees.

    """
    # convert to celsius
    celsius = (fahrenheit - 32.0) * 5.0 / 9.0
    return celsius
```

The function definition for `convert_to_celsius` from Section 8.2, Writing Our Own Functions, on page 35, is in a file named `temp.py`. You can save this file anywhere you like, although most programs create a separate directory for each set of related files that they write.

Now add another function to `temp.py`, called `above_freeze`, that receives the `f` and only the parameter `celsius` (as shown, including:

```
def convert_to_celsius(fahrenheit):
    """ Convert fahrenheit to Celsius.

    Return the number of fahrenheit degrees converted to celsius degrees.

    """
    # convert to celsius
    celsius = (fahrenheit - 32.0) * 5.0 / 9.0
    return celsius

def above_freeze(celsius):
    """ Convert celsius to Fahrenheit.

    Return the fahrenheit temperature calculated degrees in above freezing.

    """
    fahrenheit = celsius * 9.0 / 5.0 + 32.0
    print(f'{celsius} degrees is above freezing.')
    print(f'{fahrenheit} degrees Farenheit')
    print(f'{celsius} degrees C')
    print(f'{fahrenheit} degrees F')

    return celsius + 0
```

In [1]: [0]

Congratulations—you have created a module called `temperature`. Now that you've created this file, you can run it and import it like any other module:

```
import temperature
celsius = temperature.convert_to_celsius(32.0)
temperature.above_freezing(celsius)
print(celsius)
```

## What Happens During Import

Let's try another experiment. Create a file called `experiment.py` with this one statement; block it:

```
print("The pandas' scientific name is 'Malleus malleus'")
```

Run `experiment.py` and then import it:

```
>>> import experiment
The pandas' scientific name is 'Malleus malleus'
```

What this shows is that Python executes modules as if imports them. You can do anything you would do in any other program, because as far as Python is concerned, it's just another bunch of statements to be run.

Do another similar experiment. Start a fresh Python session, run `experiment.py`, and try importing module `experiment` twice now:

```
>>> import experiment
The pandas' scientific name is 'Malleus malleus'
>>> import experiment
>>>
```

Notice that the message won't print the second time. This is because Python loads modules only the first time they're imported. Internally, Python keeps track of the modules it has already seen; when it is asked to load one that's already in that list, it just skips over it. This saves time and will be particularly important when you start writing modules that import other modules, which in turn import other modules. If Python didn't keep track of what was already in memory, it could end up loading commonly used modules like `math` dozens of times.

Even if you import a module, edit that module's file, and then reimport the module, it will be reloaded. There will be some effect until you restart the shell or call `reload`. For example, after two imported `experiment`, will change the file contents to this:

```
print("The pandas' scientific name is 'Malleus malleus'")
```

We'll now call `reload` to reload module `experiment`:

```
>>> import experiment
The pandas' scientific name is 'Malleus malleus'
>>> import experiment
>>> import imp
>>> imp.reload(experiment)
The pandas' scientific name is 'Pandas pandas'
import 'experiment' from '/Users/timothy/Downloads/experiment.py'
```

In the example above, the call on `imp.reload` returns the module that was imported.

## Selecting Which Code Gets Run on Import: `__main__`

As we saw in [Section 3.7, Writing and Running a Program](#), on page 76, every Python module can be run directly from the command line or by running it from an IDE like IDLE; so, as we saw earlier in this section, it can be run indirectly (imported) by another program. If a module is to be imported by another module, then the file containing the two modules should be saved in the same directory (an alternative approach would be to use absolute file paths, which are explained in [Section 10.2, Opening a File](#), on page 173).

Sometimes we want to write code that should only be run when the module is run directly and not when the module is imported. Python defines a special string variable called `__name__` in every module to help us figure this out. Suppose we put the following into a file:

```
print('__name__ is', __name__)
```

If we run this file, its output is as follows:

```
__name__ is __main__
```

As promised, Python has created variable `__name__`. Its value is `__main__`, meaning this module is the main program. But what about if we run this when we import `sys` (instead of running it directly)?

```
sys import echo
__name__ is echo
```

The same thing happens if we write a program that does nothing but import our existing module. Create a file `import_echo.py` with this code: (note the colon)

```
import echo

print('After import, __name__ is', __name__)
and echo __name__ is', echo __name__)
```

When run from the command line, this code produces this:

```
__name__ is echo
After import, __name__ is __main__ and echo __name__ is echo
```

When Python imports a module, it sets the module's `__name__` variable to be the name of the module rather than the special string "`__main__`". This means that a module can tell whether it is the main program. Now create a file named `more_examples.py` with this code: Inside it:

```
if __name__ == '__main__':
    print('I am the main program.')
else:
    print('Another module is importing me.')
```

Try it. See what happens when you run the code by directly and when you import it.

Some of our modules contain not only function definitions but also programs. For example, consider this module `temperature.py` that contains the functions `from_celsius` and a little program:

```
File: temperature.py
# converts Celsius to Fahrenheit
# number -> float
# Celsius -> Fahrenheit
# Fahrenheit -> Celsius
# above_degrees = 32.0 * 5.0 / 9.0

def convert_to_celsius(fahrenheit):
    """ (float) -> float
    Returns the value of Celsius corresponding to fahrenheit degrees.
    Note: does not handle negative
    fahrenheit values.
    """
    return (fahrenheit - 32.0) * 5.0 / 9.0

def convert_fahrenheit(celsius):
    """ (float) -> float
    Returns fahrenheit degrees Celsius converted to Fahrenheit.
    Note: does not handle negative
    celsius values.
    """
    return celsius * 9.0 / 5.0 + 32.0

def main():
    degrees = float(input("Enter the temperature in degrees Fahrenheit:"))
    celsius = convert_to_celsius(degrees)
    print("The above degrees is", celsius)
    print("It is", convert_fahrenheit(celsius), "degrees Celsius.")

if __name__ == "__main__":
    main()
```

File Edit View Insert Cell Help

When the module is run, it prompts the user to enter a value and, depending on the value entered, prints one of two messages:

```
File: temperature.py
# converts Celsius to Fahrenheit
# number -> float
# Celsius -> Fahrenheit
# Fahrenheit -> Celsius
# above_degrees = 32.0 * 5.0 / 9.0

def convert_to_celsius(fahrenheit):
    """ (float) -> float
    Returns the value of Celsius corresponding to fahrenheit degrees.
    Note: does not handle negative
    fahrenheit values.
    """
    return (fahrenheit - 32.0) * 5.0 / 9.0

def convert_fahrenheit(celsius):
    """ (float) -> float
    Returns fahrenheit degrees Celsius converted to Fahrenheit.
    Note: does not handle negative
    celsius values.
    """
    return celsius * 9.0 / 5.0 + 32.0

def main():
    degrees = float(input("Enter the temperature in degrees Fahrenheit:"))
    celsius = convert_to_celsius(degrees)
    print("The above degrees is", celsius)
    print("It is", convert_fahrenheit(celsius), "degrees Celsius.")

if __name__ == "__main__":
    main()
```

File Edit View Insert Cell Help

Let's create another module, `utility`, that uses the conversion function from module `temperature.py` (see the following figure):



```

import temperature_program

def per_pentesting_instructions(freezing):
    """Performs actions for maintaining the oven at below-freezing and
    Celsius degrees.

    Args:
        freezing (float): Freezing point of water in degrees Celsius.

    Returns:
        str: A string containing the temperature in degrees Fahrenheit.
    """
    return f'Freeze the oven to {freezing} degrees C + {freezing} degrees F.'


def test():
    """Tests the per_pentesting_instructions function.

    Checks if the output is correct for a freezing point of 32 degrees Celsius.
    """
    assert per_pentesting_instructions(32) == "Freeze the oven to 32 degrees Celsius + 32 degrees F."

```

In [2]: Oct 6

When calling `test`, it imports `temperature_program`, so the program at the bottom of `temperature_program.py` is executed. (See [Figure 2: Execution of the imported documentation program](#) (which is in [page 110](#)).

Since we don't care whether a temperature is above freezing when calculating our oven's status, importing `temperature_program.py` we can prevent that part of the code from running by putting it in a `if __name__ == "__main__":` block ([Figure 3: The Main Function](#) is shown in [page 110](#)).

Now when calling `test` in `run`, only the code from `temperature_program.py` outside of the `if __name__ == "__main__":` block is executed ([Figure 4: Output of Running test](#) on [page 111](#)).

We will encounter `__name__` in the following sections and in other chapters.

## 6.3 Testing Your Code Semiautomatically

In [Section 3.6, Designing New Functions: A Recipe](#) on [page 47](#), we introduced the `function_design_recipe` (FDR). Following the FDR, the doctests that we write include example function calls.

The last step of the FDR involves testing the function. Up until now, we have been typing the function calls from the doctests to the shell (or copying and pasting them) to run them and then have been comparing the results with what we expect to make sure they match.

`PyTest` has a module called `assert` that allows us to run the tests that we placed in the strings all at once. It prints out whether the function fails or returns what we

Part No: 3-3-2 (Y2) 3-3-24792 Rev 200, May 11, 2013, 12:52:24  
Page 4 - 2.1 Design Doc - Thermal Test | Date: 7/17/2013 10:45:49 AM  
Type: "Thermal Test", "Test Plan" or "Failure Analysis".  
Doc: [View](#)  
Rev:  
Author: Name: John Doe Email: John.Doe@Company.com  
Title: Thermal Test Plan.  
Under the heating temperature in degrees Fahrenheit 200  
should over no 500.0 degrees F (26.7 degrees C).  
Date:

**Figure 2—Execution of the Impact Temperature program Module**

10. **below\_overhead** (Bobtail Switch):  
     $\text{overhead} \rightarrow \text{overhead}$   
  
 Perhaps the most well-known design pattern is the Bobtail Switch.  
  
 **Problem:** **How can we have multiple behaviors without inheritance?**  
  
 **Solution:** **Introducing Objects**  
 This  
 **Object** **Switching** (Bobtail Switch)  
 **Implementation:**  
  
 **return** **overhead** + **overhead** - **overhead** + **overhead**

11. **above\_overhead** (Bobtail Switch):  
     $\text{overhead} \rightarrow \text{overhead}$   
  
 Perhaps the most well-known design pattern is the Bobtail Switch.  
  
 **Problem:** **How can we have multiple behaviors without inheritance?**  
  
 **Solution:** **Introducing Objects**  
 This  
 **Object** **Switching** (Bobtail Switch)  
 **Implementation:**  
  
 **return** **overhead** + **overhead**

12. **overhead** (Bobtail Switch):  
     $\text{overhead} \rightarrow \text{overhead}$   
  
 Perhaps the most well-known design pattern is the Bobtail Switch.  
  
 **Problem:** **How can we have multiple behaviors without inheritance?**  
  
 **Solution:** **Introducing Objects**  
 This  
 **Object** **Switching** (Bobtail Switch)  
 **Implementation:**  
 **return** **overhead** + **overhead** + **overhead**  
 **return**  
 **print** ("it is below freezing.")

#### How to Use the Kudu Temperature Gauge

expenses (see Figure 6). The section should summarize the basic financial information presented. We will now proceed to write the basic financial statement by referring to the following: Selecting Which Costs Get Run on First. (Note - see page 107.)

```

Python 3.5.2 (v3.5.2:42c71e4cd111, May 12 2017, 16:52:44)
[GCC 4.2.1 Apple Inc. built 5656] on darwin
Type "copyright", "credits" or "license()" for more information.
>>> print(bake(500))
500
Bake the baking temperature in degrees Fahrenheit: 500
Print back over to 'F' or degrees F (273.0 degrees C).
>>>

```

**Figure 4—Output for execution of baking.py**

```

Python 3.5.2 (v3.5.2:42c71e4cd111, May 12 2017, 16:52:44)
[GCC 4.2.1 Apple Inc. built 5656] on darwin
Type "copyright", "credits" or "license()" for more information.
>>> print(bake(500))
500
Bake the baking temperature in degrees Fahrenheit: 500
Print back over to 'F' or degrees F (273.0 degrees C).
>>> help(bake)
>>> docstring(bake)
Help on function bake in module module_temperature:

```

**Figure 5—The doctest Module Running the Tests from module\_temperature.py**

That means it tells us that three tests were run and none of them failed. That is, the three function calls in the doctests were run, and they returned the same value that we expected and stated in the doctesting.

Now let's see what happens when there is an error in our calculation. Instead of the calculation we've been using,  $(\text{number} \cdot 32.0) + 32.0$ , let's remove the parentheses:  $\text{number} \cdot 32.0 + 32.0$  [See [Figure 6—Results of Removing the Parentheses](#), on page 112.]

Figure 7, [Failure Message for doctest](#) on page 113 shows the result of running doctest on that module.

The failure message above indicates that function call `bake(5)` was expected to return `23.333333333333332` but it actually returned `51.22222222222222`. The other two tests ran and passed.

When a failure occurs, we need to review our code to identify the problem. We should also check the expected return value listed in the doctesting to make sure that the expected value matches both the type contract and the description of the function.

[View the code for this example](#)

```

def calculate_fuel_efficiency():
    ''' calculate fuel efficiency
    return None
    # calculate the number of gallons required to travel the distance.
    def calculate_gallons(distance):
        ''' calculate the number of gallons required to travel the distance.
        return distance / 20.0 + 1.0 / 1.0
    # calculate fuel efficiency.
    def above_threshold():
        ''' calculate fuel efficiency
        return None
        # calculate the number of gallons required to travel the distance.
        def calculate_gallons(distance):
            ''' calculate the number of gallons required to travel the distance.
            return distance / 20.0 + 1.0 / 1.0
            # calculate fuel efficiency.
            def calculate_gallons(distance):
                ''' calculate the number of gallons required to travel the distance.
                return distance / 20.0 + 1.0 / 1.0
                # calculate fuel efficiency.
                def calculate_gallons(distance):
                    ''' calculate the number of gallons required to travel the distance.
                    return distance / 20.0 + 1.0 / 1.0
                    # calculate fuel efficiency.
                    def calculate_gallons(distance):
                        ''' calculate the number of gallons required to travel the distance.
                        return distance / 20.0 + 1.0 / 1.0
                        # calculate fuel efficiency.
                        def calculate_gallons(distance):
                            ''' calculate the number of gallons required to travel the distance.
                            return distance / 20.0 + 1.0 / 1.0
                            # calculate fuel efficiency.
                            def calculate_gallons(distance):
                                ''' calculate the number of gallons required to travel the distance.
                                return distance / 20.0 + 1.0 / 1.0
                                # calculate fuel efficiency.
                                def calculate_gallons(distance):
                                    ''' calculate the number of gallons required to travel the distance.
                                    return distance / 20.0 + 1.0 / 1.0
                                    # calculate fuel efficiency.
                                    def calculate_gallons(distance):
  ''' calculate the number of gallons required to travel the distance.
  return distance / 20.0 + 1.0 / 1.0
  # calculate fuel efficiency.
  def calculate_gallons(distance):
  ''' calculate the number of gallons required to travel the distance.
  return distance / 20.0 + 1.0 / 1.0
  # calculate fuel efficiency.
  def calculate_gallons(distance):
  ''' calculate the number of gallons required to travel the distance.
  return distance / 20.0 + 1.0 / 1.0
  # calculate fuel efficiency.
  def calculate_gallons(distance):
  ''' calculate the number of gallons required to travel the distance.
  return distance / 20.0 + 1.0 / 1.0
  # calculate fuel efficiency.
  def calculate_gallons(distance):
  ''' calculate the number of gallons required to travel the distance.
  return distance / 20.0 + 1.0 / 1.0
  # calculate fuel efficiency.
  def calculate_gallons(distance):
  ''' calculate the number of gallons required to travel the distance.
  return distance / 20.0 + 1.0 / 1.0
  # calculate fuel efficiency.
  def calculate_gallons(distance):
  ''' calculate the number of gallons required to travel the distance.
  return distance / 20.0 + 1.0 / 1.0
  # calculate fuel efficiency.
  def calculate_gallons(distance):
  ''' calculate the number of gallons required to travel the distance.
  return distance / 20.0 + 1.0 / 1.0
  # calculate fuel efficiency.
  def calculate_gallons(distance):
  ''' calculate the number of gallons required to travel the distance.
  return distance / 20.0 + 1.0 / 1.0
  # calculate fuel efficiency.
  def calculate_gallons(distance):
  ''' calculate the number of gallons required to travel the distance.
  return distance / 20.0 + 1.0 / 1.0
  # calculate fuel efficiency.
  def calculate_gallons(distance):
  ''' calculate the number of gallons required to travel the distance.
  return distance / 20.0 + 1.0 / 1.0
  # calculate fuel efficiency.
  def calculate_gallons(distance):
  ''' calculate the number of gallons required to travel the distance.
  return distance / 20.0 + 1.0 / 1.0
  # calculate fuel efficiency.
  def calculate_gallons(distance):
  ''' calculate the number of gallons required to travel the distance.
  return distance / 20.0 + 1.0 / 1.0
  # calculate fuel efficiency.
  def calculate_gallons(distance):
  ''' calculate the number of gallons required to travel the distance.
  return distance / 20.0 + 1.0 / 1.0
  # calculate fuel efficiency.
  def calculate_gallons(distance):
  ''' calculate the number of gallons required to travel the distance.
  return distance / 20.0 + 1.0 / 1.0
  # calculate fuel efficiency.
  def calculate_gallons(distance):
  ''' calculate the number of gallons required to travel the distance.
  return distance / 20.0 + 1.0 / 1.0
  # calculate fuel efficiency.
  def calculate_gallons(distance):
  ''' calculate the number of gallons required to travel the distance.
  return distance / 20.0 + 1.0 / 1.0
  # calculate fuel efficiency.
  def calculate_gallons(distance):
  ''' calculate the number of gallons required to travel the distance.
  return distance / 20.0 + 1.0 / 1.0
  # calculate fuel efficiency.
  def calculate_gallons(distance):
  ''' calculate the number of gallons required to travel the distance.
  return distance / 20.0 + 1.0 / 1.0
  # calculate fuel efficiency.
  def calculate_gallons(distance):
  ''' calculate the number of gallons required to travel the distance.
  return distance / 20.0 + 1.0 / 1.0
  # calculate fuel efficiency.

```

See in Editor

**Figure 6—Results of Removing the Parentheses**

## 6.4 Tips for Grouping Your Functions

Put functions and variables that logically belong together in the same module. If they don't seem logical together—for example, if one of the functions calculates how much carbon dioxide different kinds of cars produce, while another figures out how strength given the hemispherical angle and density—then you shouldn't put them in one module just because you happen to be the author of both.

Of course, you'll often have different opinions about what's logical and what isn't. Take Python's math module, for example: should sine, cosine, for example, matrices go in there too, or should they go in a separate linear algebra module? What about basic statistical functions? Going back to the previous paragraph, wouldn't functions like calculate\_fuel\_efficiency() fit in the same module as other fuel calculations carbon emissions? You can always find a reason why two pieces of code should not be in the same module. But a thousand modules with one function each are going to be hard for people (including you) to work with.

```

#fizzbuzz.py
# Python 2.7.13 |Anaconda 3.2.0 (64-bit)| (Python 2.7.13, May 12 2015, 11:52:31)
# [GCC 4.2.1 -- [PyPy 2.4.2 |Anaconda 3.2.0 (64-bit)|] on darwin
# Type "copyright", "credits" or "license" for more information.
# http://pypy.org
# http://pypy.org/licenses.html

# Prints the words 'fizz', 'buzz', or 'fizzbuzz' for each number from 1 to n.
# If n is not a positive integer, it prints nothing.
# If n is divisible by 3, it prints 'fizz'.
# If n is divisible by 5, it prints 'buzz'.
# If n is divisible by both 3 and 5, it prints 'fizzbuzz'.
# If n is not divisible by either 3 or 5, it prints nothing.

def fizzbuzz(n):
    """Prints the words 'fizz', 'buzz', or 'fizzbuzz' for each number from 1 to n.
    If n is not a positive integer, it prints nothing.
    If n is divisible by 3, it prints 'fizz'.
    If n is divisible by 5, it prints 'buzz'.
    If n is divisible by both 3 and 5, it prints 'fizzbuzz'.
    If n is not divisible by either 3 or 5, it prints nothing.

    :param int n: The number to print fizzbuzz for.
    """
    for i in range(1, n+1):
        if i % 3 == 0 and i % 5 == 0:
            print("fizzbuzz")
        elif i % 3 == 0:
            print("fizz")
        elif i % 5 == 0:
            print("buzz")
        else:
            pass

```

In [24]:

**Figure 7—FizzBuzz code for desktop**

As a rule of thumb, if a module has less than a handful of things in it, it's probably too small, and if you can't sum up the contents and purpose of a module in a single or two-sentence description, it's probably too big. These are just guidelines, though; in the end, you'll have to decide based on how more experienced programmers have handled modules like the ones in the Python standard library, and eventually on your own sense of style.

## 6.5 Organizing Our Thoughts

In this chapter you learned the following:

- A module is a collection of functions and variables grouped together in a file. To use a module, you must first import it. Using `import sys`, `sys` has been imported, you refer to its contents using `sys.argv` and `sys.exit()`—module variables.
- Variable `_name_` is created by Python and can be used to specify that some code should only run when the module is run directly and not when the module is imported.
- Functions have to do more than just run to be useful; they have to run correctly. One way to ensure that they do so is to `assert`, which you can do in Python using `assert`.

## 6.6 Exercises

Here are some exercises for you to try on your own. Solutions are available at [nostarch.com/python-practice-book](https://nostarch.com/python-practice-book).

1. Import module `math`, and use the functions to complete the following exercises. (You can call `dir(math)` to get a listing of the items in `math`.)
  - a. Write an expression that produces the float of 7.3.
  - b. Write an expression that rounds the value of `42` and then produces the absolute value of that result.
  - c. Write an expression that produces the ceiling of the sum of `3.1` & `5`.
2. In the following exercises, you will work with Python's `calendar` module.
  - a. Visit the Python documentation website at <https://docs.python.org/3/library/calendar.html>, and look at the Documentation on module `calendar`.
  - b. Import module `calendar`.
  - c. Using function `help`, read the description of function `days`.
  - d. Use `days` to determine the last leap year.
  - e. Use `dir` to get a list of what `calendar` contains.
  - f. Print and use a function in module `calendar` to determine how many leap years there will be between the years 2000 and 2050, inclusive.
  - g. Find, and use a function in module `calendar` to determine which day of the week July 26, 2015, will be.
3. Create a file named `zodiac.py` with this code inside it:
 

```
def zodiac(sign, num):
    """(str, int) --> str

    Returns the sign of the year num.

    >>> zodiac('Aries', 1990)
    'Aries'
    >>> zodiac('Aries', 1991)
    'Taurus'
    ...
    ...

    returns sign + num % 12
```

  - a. Run `python3 zodiac.py` and run `doctests[1:10]`
  - b. Run `doctests[11:16]` in `ipython` session describing fail. Fix the code and return `doctests`. Repeat this procedure until there are none.

# Using Methods

So far we've seen lots of functions built-in functions, functions inside modules, and functions that we've defined. A method is another kind of function that is attached to a particular type. There are *str* methods, *int* methods, *list* methods, and more—every type has its own set of methods. In this chapter, we'll explore how to use methods and also how they differ from the rest of the functions that we've seen.

## 7.1 Modules, Classes, and Methods

In [Section 6.1, Importing Modules](#), on page 100, we saw that a module is a kind of object, one that can contain functions and other variables. There is another kind of object that is similar to a module: a class. You've been using classes all along, probably without realizing it: a class is how Python represents a type.

You may have called built-in functions like `str`, `len`, or `list`. We'll do that now with `dir` (not to check the first few keys that it's a class):

```
>>> help(str)
Help on class str in module builtins:

class str(object):
    |   str(object[, encoding[, errors]]) -> str
    |
    |   Create a new string object from the given object. If encoding or
    |   errors is specified, then the object must support a code buffer
    |   that will be decoded using the given encoding and error handler.
    |   Otherwise, return the result of object.__str__(), or oct(self),
    |   or hex(self).
    |   encoding defaults to sys.getdefaultencoding()
    |   errors defaults to 'strict'
    |
    | Methods defined here:
```

|                                                                                    |                                                                                                                                                                                                                                                                        |
|------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>s.add(...)</code>                                                            | <code>s.add('T')</code> adds 'T' to the string.                                                                                                                                                                                                                        |
| <code>s.contains(...)</code>                                                       | <code>s.contains('T')</code> returns <code>True</code> if the string contains a 'T'.                                                                                                                                                                                   |
| <b>[Lots of other names with leading and trailing underscores not shown here.]</b> |                                                                                                                                                                                                                                                                        |
| <code>s.capitalize(...)</code>                                                     | <code>s.capitalize()</code> returns a capitalized version of <code>s</code> , i.e., makes the first character have upper case and the rest lower case.                                                                                                                 |
| <code>s.center(...)</code>                                                         | <code>s.center(10)</code> returns a string of length 10 containing <code>s</code> , done using the specified fill character (vertical ellipsis).                                                                                                                       |
| <code>s.count(...)</code>                                                          | <code>s.count('a')</code> , <code>s.count('a', 0, 10)</code> return the number of non-overlapping occurrences of <code>a</code> starting at <code>s[0]</code> through <code>s[10]</code> ; occurrences after <code>s[10]</code> are not counted as in either notation. |

[There are many more of these as well.]

Now the top of the documentation is this:

- | `str(s[, encoding[, errors]])`
- | Create a new string object from the given object.

That describes how to use `str` as a function: we can call it to create a `String`. For example, `str('12')` creates the `String` '12'.

We can also use `str` to call a method in class `str`, much like we call a function in module `math`. The main difference is that every method in class `str` requires a `String` as its first argument:

```
ss = str.capitalize('Browning')
'Browning'
```

This is how methods are different from functions: the first argument to every string method must be a `String`, and the parameter is not described in the documentation for the method. This is because all `String` methods require a `String` as the first argument, and more generally, all methods in a class require

an object of that class as the first argument. Here are two more examples, this time using the other two string methods from the earlier part of this book. Both of these also require a string as the first argument:

```
>>> s1.rjust(12)('Sonnet 43'), 26)
'          Sonnet 43'
>>> s1.rjust('How do I love thee? Let me count the ways.', 'the')
2
```

The first method call produces a new string that contains 'Sonnet 43' in a string of length 26, padding to the left and right with spaces.

The second method call counts how many times the string in the `s1` variable occurs between the words 'love' in the word 'thee' and 'the' as the last word in the string.

## 7.2 Calling Methods the Object-Oriented Way

Remember, every method in a class always requires `self` as the first argument (and, more generally, because every method in any class requires an object of that class as the first argument). Python provides a shorthand form for calling a method when the object appears first and then the method call:

```
>>> 'Browning'.capitalize()
'Browning'
>>> 'Sonnet 43'.center(26)
'          Sonnet 43'
>>> 'How do I love thee? Let me count the ways.', count('the')
2
```

When Python encounters one of these method calls, it translates it to the more long-winded form. We will use this shorthand form throughout the rest of this book.

The help documentation for methods uses this form. Here is the help for `str.lower` in `class str`. Notice that we can get help for a single method by prefixing it with the class it belongs to:

```
>>> help(str.lower)
Help on method_descriptor:

lower(...)

    S.lower() -> S.lower()

    <-- Return a copy of the string S converted to lowercase.
```

Contrast that documentation with the help for `funcode_type`'s `imethods` method:

```
>>> helpcmath.sqrt
Help on built-in function sqrt in module math:

sqrt(...)

Return the square root of x.
```

The help for `sqrt` shows that you need to prefix the call with the string value `S`, the help for each opt doesn't show any such prefix.

The general form of a method call is as follows:

```
Sexpression.Method_name(Argument1)
```

So far every example we've seen has a single object as the expression, but any expression can be used as long as it evaluates to the correct type. Here's an example:

```
>>> DNA = "GATTACCTGATTCATG"
>>> DNA.count('T')
2
```

The expression `"TAC" + G + "G"` evaluates to the DNA sequence `"TAGGC"` and that is the object that is used in the call on `count` method of `DNA`.

Here are the steps for executing a method call. These steps are quite similar to those for executing a function and in [Section 3.3, Tracing Function Calls to the Memory Model](#), we saw (4).

1. Evaluate expression: This may be something simple, like `Elizabeth Barrett Browning` (a poet from the 1800s), or it may be more complicated, like `TTA + G * 3`. Either way, a single object is produced and then will be the object we are interacting with during the method call.
2. Note that we have an object, evaluate the method arguments left to right. In our DNA example, the argument is `T`.
3. Pass the result of evaluating the initial expression as the first argument, and also pass the argument values from the previous step, into the method. In our DNA example, `count` is equivalent to `count("TAGGC", T)`.
4. Execute the method.

When the method call finishes, it produces a value. In our DNA example, `count("TAGGC", T)` returns the number of times `T` occurs in `"TAGGC"`, which is `2`.

## Why Programming Languages Are Called Object Oriented

The other programming languages I mentioned in chapter 1 are programming, while *Object-oriented* means that we tell objects to do things by calling their methods, an approach to imperative programming. What that means are the primary focus and the power of an object to work with. Python allows a mixture of both styles.

### 7.3 Exploring String Methods

Strings are central to programming; almost every algorithm uses strings in some way. We'll explore some of the ways in which we can manipulate strings and, at the same time, firm up our understanding of methods.

The most commonly used string methods are listed in Table 7, Common String Methods. (You can find the complete list in Python's online documentation, at <https://docs.python.org/3/library/stdtypes.html>.)

| METHOD                            | DESCRIPTION                                                                                                                                                                                                                                                                                                                                   |
|-----------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>s.capitalize()</code>       | Returns a copy of the string with the first letter capitalized and the rest lowercase.                                                                                                                                                                                                                                                        |
| <code>s.count(s)</code>           | Returns the number of non-overlapping occurrences of <code>s</code> in the string.                                                                                                                                                                                                                                                            |
| <code>s.endswith(suffix)</code>   | Returns <code>True</code> if the string ends with the characters in the suffix string—this is case sensitive.                                                                                                                                                                                                                                 |
| <code>s.find(s)</code>            | Returns the index of the first occurrence of <code>s</code> in the string, or <code>-1</code> if <code>s</code> doesn't occur in the string—the first character is at index 0. This is case sensitive.                                                                                                                                        |
| <code>s.find(s, beg)</code>       | Returns the index of the first occurrence of <code>s</code> at or after index <code>beg</code> in the string, or <code>-1</code> if <code>s</code> doesn't occur in the string, or after index <code>beg</code> —the first character is at index 0. This is case sensitive.                                                                   |
| <code>s.find(s, beg, end)</code>  | Returns the index of the first occurrence of <code>s</code> between indices <code>beg</code> (inclusive) and <code>end</code> (exclusive) in the string, or <code>-1</code> if <code>s</code> does not occur in the string between indices <code>beg</code> and <code>end</code> . The first character is at index 0. This is case sensitive. |
| <code>s.format(*expr args)</code> | Creates a string made by substituting the placeholder <code>{}</code> in the string—with what is a pair of braces, {}, and {} with an integer in between. The expression arguments are numbered from left to right starting                                                                                                                   |

| Method                            | Description                                                                                                                                                                                                                                                                                              |
|-----------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>at(i)</code>                | If <code>i</code> is less than <code>s.length()</code> , it is replaced by the value produced by evaluating the expression whose index corresponds with the character in between the braces of the field. If the expression produces a value that isn't a string, that value is converted into a string. |
| <code>endswith(suffix)</code>     | Returns <code>True</code> if all characters in the string are lowercase.                                                                                                                                                                                                                                 |
| <code>startswith(suffix)</code>   | Returns <code>True</code> if all characters in the string are uppercase.                                                                                                                                                                                                                                 |
| <code>s.lower()</code>            | Returns a copy of the string with all letters converted to lowercase.                                                                                                                                                                                                                                    |
| <code>s.lstrip()</code>           | Returns a copy of the string with leading whitespace removed.                                                                                                                                                                                                                                            |
| <code>s.rstrip()</code>           | Returns a copy of the string with trailing whitespace removed.                                                                                                                                                                                                                                           |
| <code>s.replace(old,new)</code>   | Returns a copy of the string with all occurrences of substring <code>old</code> replaced with string <code>new</code> .                                                                                                                                                                                  |
| <code>s.strip()</code>            | Returns a copy of the string with leading whitespace removed.                                                                                                                                                                                                                                            |
| <code>s.split()</code>            | Returns a copy of the string with trailing whitespace removed.                                                                                                                                                                                                                                           |
| <code>s.splitlines()</code>       | Returns the whitespace-separated words in the string as a list (We'll introduce the <code>list</code> type in Section 6.1, <a href="#">String and Accounting Down to Basic</a> on page 129.)                                                                                                             |
| <code>s.startswith(prefix)</code> | Returns <code>True</code> if the string starts with the letters in the string beginning—this is case sensitive.                                                                                                                                                                                          |
| <code>s.lstrip(i)</code>          | Returns a copy of the string with resulting word trailing whitespace removed.                                                                                                                                                                                                                            |
| <code>s.rstrip(i)</code>          | Returns a copy of the string with leading and trailing occurrences of the character in <code>i</code> removed.                                                                                                                                                                                           |
| <code>s.swapcase()</code>         | Returns a copy of the string with all lowercase letters converted and all uppercase letters made lowercase.                                                                                                                                                                                              |
| <code>s.upper()</code>            | Returns a copy of the string with all letters converted to uppercase.                                                                                                                                                                                                                                    |

**Table 7** Common String Methods

Method calls look almost like common function calls, except that in order to call a method we need an object of the type associated with that method. For example, let's call the method `startswith` on the string 'specie'.

```
<code>'specie'.startswith('sp')</code>
False
<code>'specie'.startswith('spe')</code>
True
```

String method `startswith` takes a single argument and returns a boolean indicating whether the string whose method was called—the one to the left of the dot—starts with the string that is given as an argument. There is also an `endswith` method:

```
<code>'specie'.endswith('e')</code>
False
<code>'specie'.endswith('es')</code>
True
```

Sometimes strings have some whitespace at the beginning and the end. The string methods `strip`, `lstrip`, and `rstrip` remove whitespace from the front, from the end, and from both, respectively. This example shows the result of applying these three functions to a string with trailing and leading whitespace:

```
<code>composed = " Mr. Helly! Mr. Internet! "</code>
<code>composed.lstrip()</code>
'Mr. Helly! Mr. Internet! '<code>
<code>composed.rstrip()</code>
' Mr. Helly! Mr. Internet! '<code>
<code>composed.strip()</code>
' Mr. Helly! Mr. Internet! '
```

Note that the other whitespace inside the string is unaffected. These methods only work from the front and end. Here is another example that uses string method `replace` to change lowercase letters to uppercase and uppercase to lowercase:

```
<code>'Computer Science'.swapcase()</code>
'COMPUTER SCIENCE'
```

`swapcase` method formal has a complex description, but a couple of examples should clear up the confusion. Here we show that we can substitute a series of strings into a formal string:

```
<code>'"{}" is derived from "{}", formed by name, in year {}</code>
'none' is derived from 'no one'
<code>'"{}" is derived from the {}-{}{}, format({Etymology}, 'Greek',
'other')</code>
...
'Etymology' is derived from the Greek "ετυμος"
<code>'"{}" is derived from the {}-{}{}, format({Lanchar}, 'Asian', 'Latin')</code>
'December' is derived from the Latin "decem"
```

<https://www.udemy.com/course/python-the-complete-data-scientist-in-python-3/>

We can have any number of fields. The last example shows that we don't have to use the numbers in order.

Next, using writing method `format`, we'll specify the number of decimal places to round a number to. We indicate this by following the field number with a colon and then using `3f` to state that the number should be formatted as a floating-point number with two digits to the right of the decimal point:

```
>>> np.pi -> 3.14159
>>> 'Pi rounded to {} decimal places is {:.3f}'.format(1, np.pi)
'Pi rounded to 1 decimal place is 3.14'
>>> 'Pi rounded to {} decimal places is {:.2f}'.format(1, np.pi)
'Pi rounded to 2 decimal places is 3.14'
>>> 'Pi rounded to {} decimal places is {:.1f}'.format(1, np.pi)
'Pi rounded to 3 decimal places is 3.141'
```

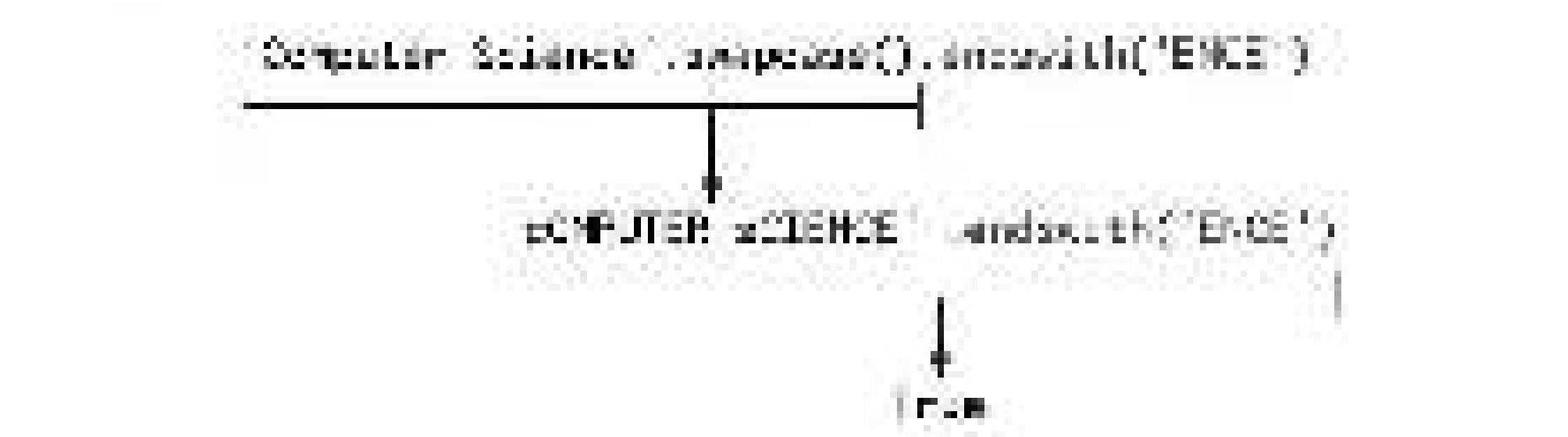
It's possible to omit the position numbers. If that's done, then the arguments passed to `format` replace each placeholder field in order from left to right:

```
>>> 'Pi rounded to {} decimal places is {:.1f}'.format(np.pi)
'Pi rounded to 1 decimal places is 3.142'
```

Remember how a method call starts with an expression? Because `Computer Science` is an expression, we can immediately call method `endswith` on the result of that expression to check whether that result has 'ECE' as its last four characters:

```
>>> 'Computer Science'.endswith('ECE')
True
```

**Figure 8—Chaining Method Calls** gives a picture of what happens when we do this:



**Figure 8—Chaining Method Calls**

The call on method `endswith` produces a new string, and that new string is used for the call on method `endswith`.

`Dash` and `Dot` are classes. It is possible to access the documentation for these either by calling `help()` or by calling `help` on an object of the class:

```
>>> help(str)
Help on int objects:

class int(object):
    |__ init__(self, value=0, base=10)
    |
    | Convert a string or number to an integer. If value is a floating
    | point argument will be truncated towards zero (this does not produce a
    | string representation of a floating point number). When converting a
    | string, use the optional base. It is an error to supply a base when
    | converting a non-string.
    |
    | Methods defined here:
    |
    | __abs__(self) ...
    |     <__abs__(self) at 0x0000000000000000>
    |
    | __add__(self, value, /)
    |     <__add__(self, value) at 0x0000000000000000>
    |
    | __bool__(self) ...
    |     <__bool__(self) at 0x0000000000000000>
```

Most modern programming languages are structured this way: the “things” in the program are objects, and most of the work in the program consists of methods that use the data stored in those objects. Chapter 14, [Object-Oriented Programming, on page 266](#), will show you how to create new kinds of objects; until then, we’ll work with objects in a way that’s built into Python.

## 7.4 What Are Those Underscores?

Any method (or other name) beginning and ending with two underscores is considered special by Python. The `help` documentation for `str` shows three methods among many others:

```
| Methods defined here:
|
| __abs__(self) ...
|     <__abs__(self) at 0x0000000000000000>
|
| __add__(self, value, /)
|     <__add__(self, value) at 0x0000000000000000>
```

These methods are typically connected with some older syntax in Python: one of the syntax will trigger a method call. For example, string method `__add__` is called when anything is added to a string:

```
>>> 'TTA' + 'GG'
'TTAGGG'
>>> 'TTA'.__add__('GG')
'TTAGGG'
```

Programmers almost never call these special methods directly; that is, you’re unlikely to see this and may help you to understand how Python works.

Integers and floating-point numbers have similar features. Here is part of the help documentation for int:

**Help on class int in module builtins:**

class int(object)

```
    |     methods defined here:
    |
    |     abs(...)
    |     ~abs__(self: object) -> object
    |
    |     __add__(...)
    |     ~__add__(self: object, value: object) -> object
    |
    |     __lt__(...)
    |     ~__lt__(self: object, value: object) -> bool
```

The documentation describes when these are called. Here we show both variations of getting the `x` value when `x` is a number:

```
>>> abs(-2)
-2
>>> abs(-2.0)
-2.0
```

We need to put a space after `-` in the second instance (with the underscores) so that Python doesn't think we're making a floating point number `.3` (a number that we've lost all the trailing `0`s).

We could also do integers using this trick:

```
>>> 3 > 5
False
>>> 3 > __gt__(5)
False
>>> 5 > 3
True
>>> 5 > __gt__(3)
True
```

And here we compare two numbers to see whether one is bigger than the other:

```
>>> 3 > 5
False
>>> 3 > __gt__(5)
False
>>> 5 > 3
True
>>> 5 > __gt__(3)
True
```

Again, programmers don't typically do this, but it is worth knowing that Python uses methods to handle all of these operations.

Such functions, like other objects, are mutable-and-writeable variables. For example, the documentation for such function is stored in a variable called `_abs__doc_`:

```
>>> abs.__doc__
'abs(number) --> number\nReturn the absolute value of the argument.'
```

When we use built-in function `print` to print that `_abs__doc_`, here what comes out is looks just like the output from calling built-in function `help`:

```
>>> print(abs.__doc__)
abs(number) --> number
```

Return the absolute value of the argument.

```
>>> help(abs)
```

**Help on built-in function abs in module builtins:**

```
>>> ?abs
```

```
abs(number) --> number
```

Return the absolute value of the argument.

Every function object keeps track of its docstring in a special variable called `_func_doc_`:

## 7.5 A Methodical Review

- **Classes** are like medieval castles that classes contain methods and modules contain functions.
- **Methods** are like functions, except that their arguments must be an object of the class in which the method is defined.
- Method calls in this form `obj.method()` are shortened for this, dropping the prefix `.`.
- Methods beginning and ending with two underscores are considered special by Python, and they are triggered by particular syntax.

## 7.6 Exercises

There are some exercises for you to try on your own. Solutions are available at <https://tinyurl.com/yxqgqzv6> under section `exercises`.

1. In the Python shell, execute the following method calls:

- a. `"Hello world!"`
- b. `"Happy birthday Jason!"`
- c. `"Helloooooo!!!!!!" * 1000000`

1. `'Hello world!'`
  2. `'apple orange'`
  3. `'Hello world!'`
  4. `'Hello [1].format("Python")'`
  5. `'Hello [1]! Hello [1].format("Python", "World")'`
2. Using string method `count`, write an expression that produces the number of `o`s in `"Hello"`.
3. Using string method `lrc`, write an expression that produces the index of the first occurrence of `o` in `"Hello"`.
4. Using `string` method `find`, write a single expression that produces the index of the second occurrence of `o` in `"Hello"`. Hint: Call `find` twice.
5. Using your expression from the previous example, find the second `a` in `"Hello"`. If you don't get the result you expect, modify the expression and try again.
6. Using `string` method `replace`, write an expression that produces a string based on `name` with `name` replaced by `life`.
7. Variable `s` refers to `'sys.'` When a string method is called with `s` as its argument, the string `'s'` is prepended. Which string method was called?
8. Variable `f` refers to `print()`. For the following function calls, in what order are the statements evaluated?
- a. `function(f)(lambda():)`
  - b. `function(lambda():function())`
  - c. `function(function(), lambda():)`
9. Variable `socon` refers to `socket`. Using `string` method `format` and variable `socon`, write an expression that produces `'socat net'`.
10. Variables `size1`, `size2`, and `size3` refer to 3, 4, and 5, respectively. Using `string` method `format` and three `size` variables, write an expression that produces `'The size are: size1, size2, size3'`.
11. Using `string` methods, write expressions that produce the following:
- a. A copy of `text` capitalized
  - b. The last occurrence of `2` in `002 400`
  - c. The second occurrence of `2` in `002 400`
  - d. The 11 and only 11 character beginning `text`

- 1. A copy of 'PMLe' removed to lowercase and then capitalised.
  - 2. A copy of 'Monkey' with the leading whitespace removed.
3. Complete the examples in the doctesting and then write the body of the following function:

```
def total_occurrences(s, c):  
    """ (str, str) -> int  
  
    Return the total number of times that ch occurs in s.  
  
    >>> total_occurrences('color', 'yellow') 10  
    3  
    >>> total_occurrences('red', 'blue') 11  
    >>> total_occurrences('green', 'purple') 0  
    """
```

# Storing Collections of Data Using Lists

Up to this point, we have seen numbers, Boolean values, strings, functions, and a few other types. Once one of these objects has been created, it can't be modified. In this chapter, you will learn how to use a Python type named list. Lists contain zero or more objects and are used to keep track of collections of data. Unlike the other types you've learned so far, lists are mutable.

## 8.1 Storing and Accessing Data in Lists

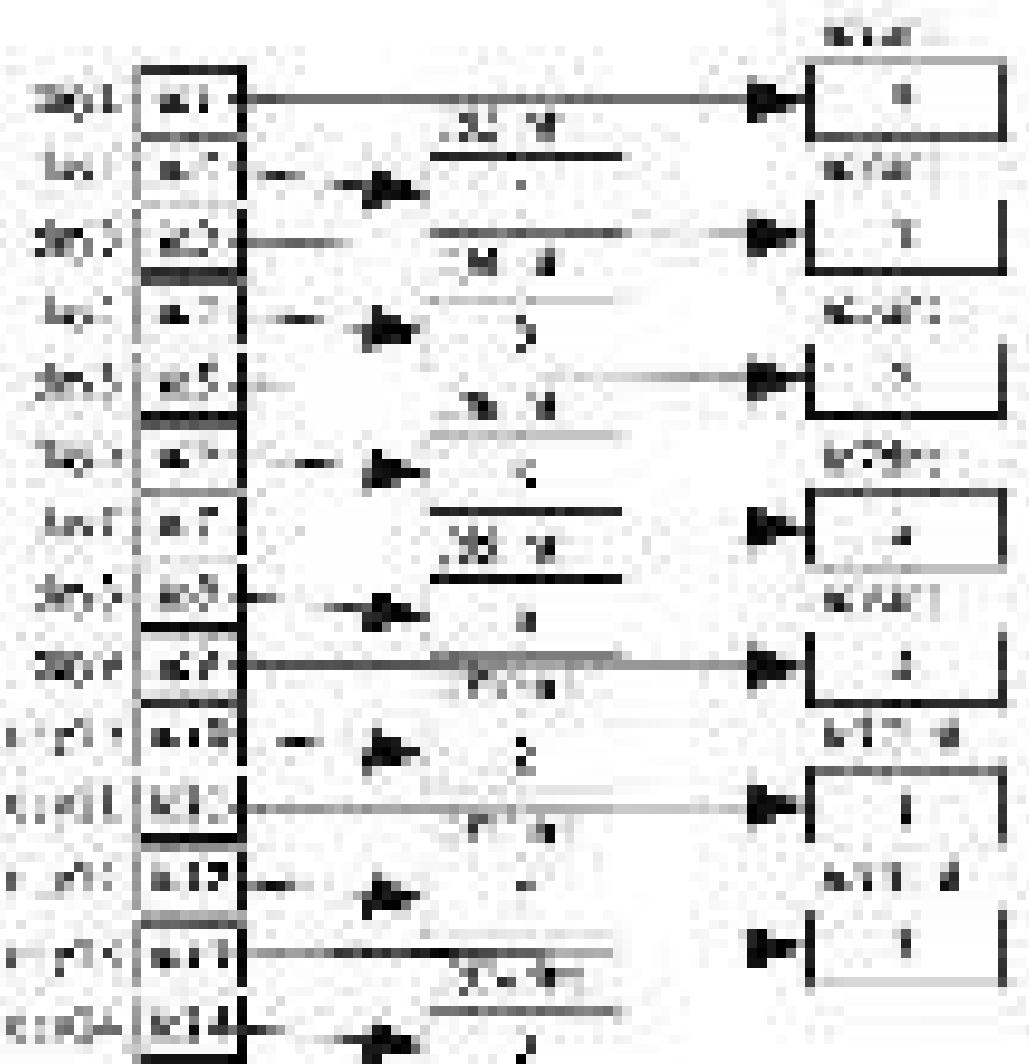
Table 8—Gray Whale Census shows the number of gray whales counted near the Coal Oil Point Natural Reserve in a two-week period starting on February 27, 2018.<sup>1</sup>

| Day | Number of Whales | Day | Number of Whales |
|-----|------------------|-----|------------------|
| 1   | 5                | 8   | 6                |
| 2   | 1                | 9   | 0                |
| 3   | 7                | 10  | 2                |
| 4   | 3                | 11  | 1                |
| 5   | 2                | 12  | 7                |
| 6   | 1                | 13  | 1                |
| 7   | 2                | 14  | 3                |

Table 8—Gray Whale Census

Now that we have seen so far, we haven't been able to create variables to keep track of the number of whales counted each day:

<sup>1</sup> Gray Whales Count Report 2017/18 compilation by Research and Education. <http://www.graywhalecount.org/2017-18/Count-Block-Data.pdf>



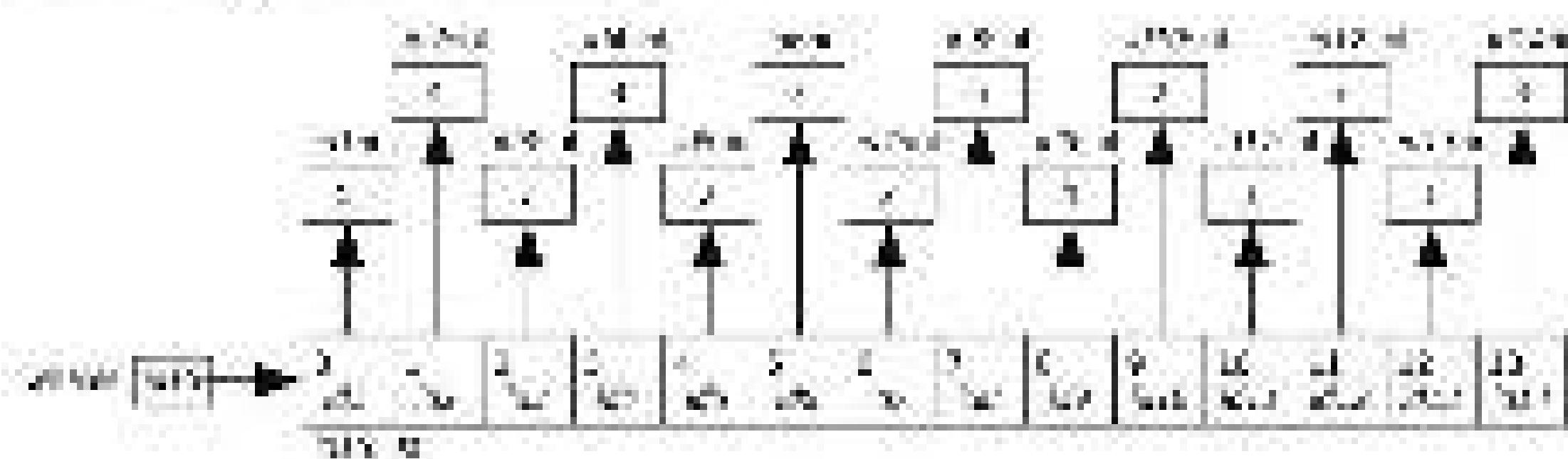
To track an entire year's worth of observations, we would need 365 variables (one for a leap year).

Rather than dealing with this programming nightmare, we can use a list to keep track of the 14 days of whale counts. That is, we can use a list to keep track of the 14 *objects* that concern the counts:

```

>>> whales = [13, 4, 7, 3, 2, 3, 2, 6, 4, 2, 1, 7, 1, 21]
>>> whales
[13, 4, 7, 3, 2, 3, 2, 6, 4, 2, 1, 7, 1, 21]
  
```

A list is an object; like any other object, it can be assigned to a variable. Here is what happens in the memory model:



The general form of a list expression is as follows:

**[*expression1*, *expression2*, ..., *expressionN*]**

The empty list is expressed as [].

In our whale count example, variables *whales* refers to a list with fourteen items, also known as elements. The list itself is an object, but it also contains the

memory addresses of further other objects. The memory model also shows what's after this assignment statement has been executed:

The elements in the list are ordered, and each item has an index corresponding to its position in the list. The first element is at index 0, the second at index 1, and so on. It would be more natural to use 1 as the first index, as human languages do. Python, however, uses the same convention as languages like C and Java and starts counting at zero. To refer to a particular list item, we just use the index in brackets after a reference to the list, just as the name of a variable:

```
>>> shelves = [5, 4, 3, 2, 1, 0, 4, 3, 2, 1, 0, 4]
>>> shelves[0]
5
>>> shelves[1]
4
>>> shelves[2]
1
>>> shelves[3]
0
>>> shelves[4]
```

We can use only those indices that are in the range from zero up to one less than the length of the list, because the list index starts at 0, not at 1. In a fourteen-item list, the valid indices are: 0, 1, 2, and so on, up to 13. Trying to use an out-of-range index results in an error:

```
>>> shelves = [5, 4, 3, 2, 1, 0, 4, 3, 2, 1, 0, 4]
>>> shelves[16]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Unlike most programming languages, Python also has negative index backtracking from the end of a list. The last item is at index -1, the one before it at index -2, and so on. Negative indices provide a way to access the last item, second-to-last item and so on, without having to figure out the size of the list:

```
>>> shelves = [5, 4, 3, 2, 1, 0, 4, 3, 2, 1, 0, 4]
>>> shelves[-1]
0
>>> shelves[-2]
1
>>> shelves[-14]
5
>>> shelves[-15]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Store each item in a list as an object. In this case, the words are stored in other variables:

```
var studies = [5, 4, 7, 3, 2, 5, 8, 6, 2, 1, 7, 1, 9]
var third = studies[2]
var point = third.days();
third.days();
```

In [Section 10.6, Accessing Python's Built-in Methods](#) on page 135, you will learn that an entire list, such as the one that `third` refers to, can be assigned to other variables and discover the effect that has.

## The Empty List

In [Chapter 4, Working with Text](#), on page 63, we saw the empty string, which doesn't contain any characters. There is also an empty list. An empty list is a list with no items in it, so with all index, an empty list is represented using brackets:

```
var studies = []
```

Store an empty list, but no items; trying to index an empty list results in an error:

```
var studies[0]
Traceback (most recent call last):
  File "python", line 1, in <module>
IndexError: list index out of range
var studies[-1]
Traceback (most recent call last):
  File "python", line 1, in <module>
IndexError: list index out of range
```

## Lists Are Heterogeneous

You can contain any type of data, including integers, strings, and even other lists. Here is a list of information about the element krypton, including its name, symbol, melting point (in degrees Celsius), and boiling point (also in degrees Celsius):

```
var krypton = ['Krypton', 'Kr', -157.2, -157.4]
var krypton[1]
'Kr'
var krypton[2]
-157.2
```

A list is usually used to contain items of the same kind, like temperatures or dates or grades in a course. A list can be used to aggregate related information about different items, as we did with `krypton`. By <http://bjjmc.luminum.com/tut/>, we

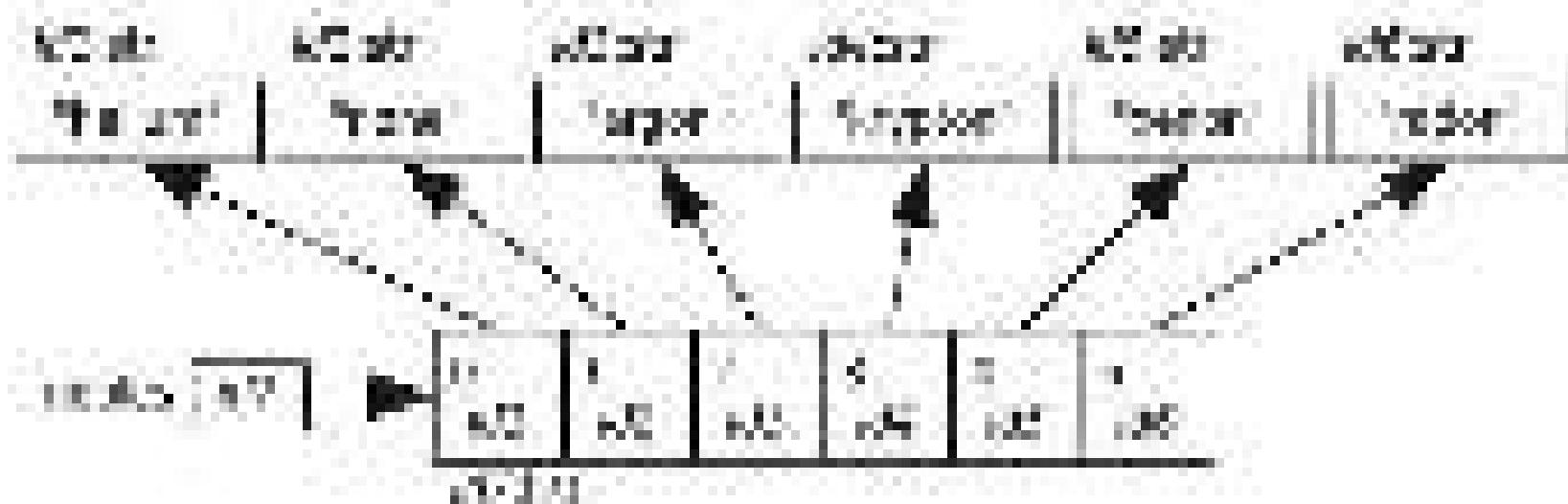
need to remember which temperature comes first and whether the name of the symbol starts the list. Another common source of bugs is when you forget to include a piece of data in your list (or perhaps it was missing in your source of information). For example, would you keep track of similar information for William if you don't know his working plant? What information would you put at index 2? A better, but more advanced, way to do this is described in Chapter 14, *Object-Oriented Programming*, (page 300).

## 0.2 Modifying Lists

Suppose you're trying to a list of the noble gases and your friend says

```
new_nobles = ['Helium', ' neon', ' argon', ' krypton', ' xenon', ' radon']
```

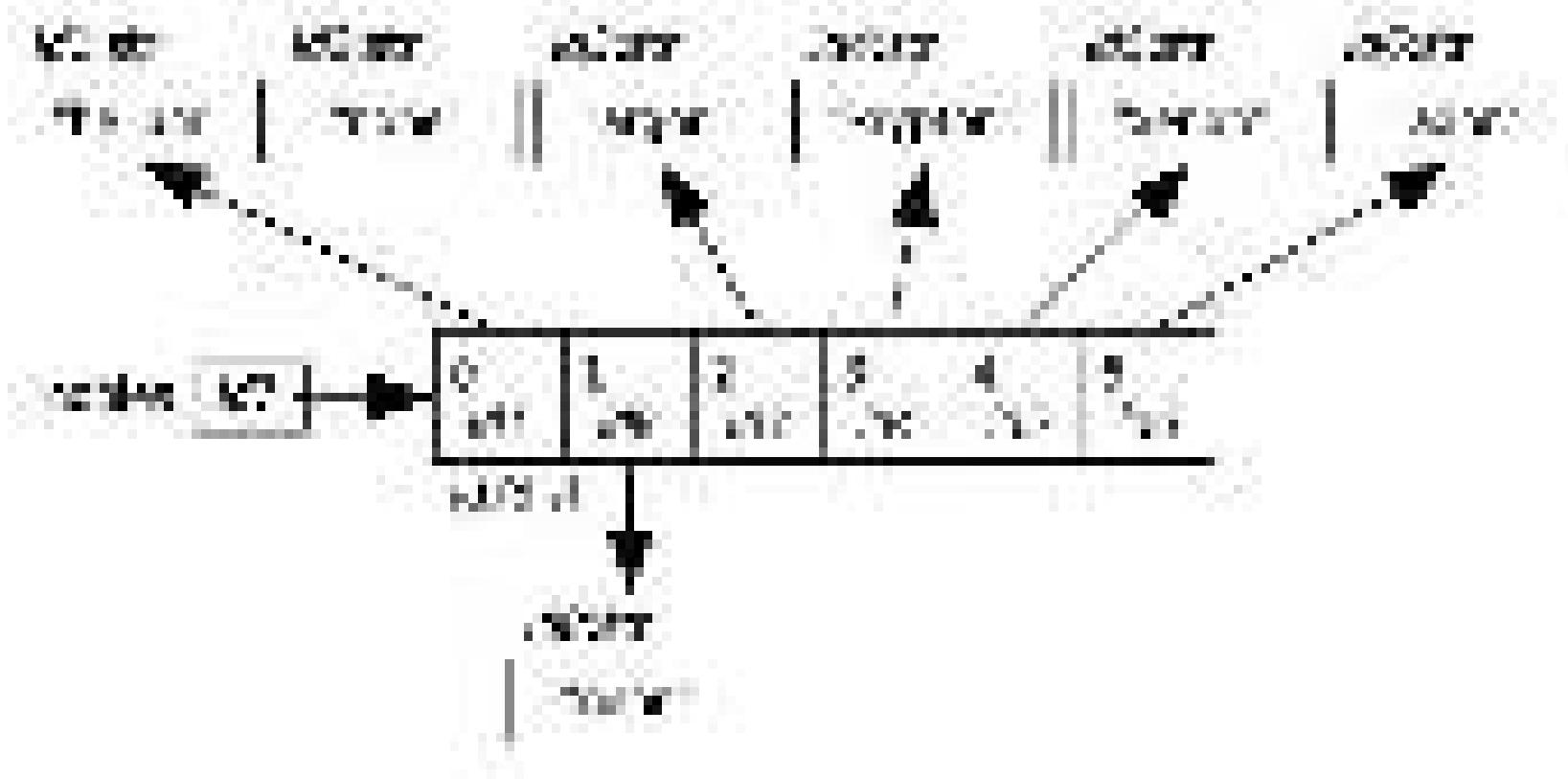
The error here is that you typed `neon` instead of `neon`. Here's the memory model that we would have had assignment statements:



Rather than retyping the whole list, you can assign a new value to a specific element of the list.

```
new_nobles[1] = 'neon'
new_nobles
['Helium', 'neon', 'argon', 'krypton', 'xenon', 'radon']
```

Here's the result after the assignment to `new_nobles[1]`:



That memory model also shows that list objects are, *mutable*. That is, the contents of a list can be modified.

On the other hand, `value[i]` was used on the left side of the assignment operator. It does not have to be used on the right side. In general, in an expression of the form `L[i] = value`, list `L` at index `i` behaves just like a simple variable (see [Section 2.4, Variables and Computer Memory: Remembering Values, on page 128](#)).

If `value` is used in an expression (such as on the right of an assignment statement), it means “Get the value referred to by the memory address at index `i` of list `L”`.

On the other hand, `L[i]` is on the left of an assignment statement (as in `value[i] = value'`), it means “Look up the memory address at index `i` of list `L`, as it can be overwritten.”

In contrast, in lists, numbers and strings are *immutable*. That means, for example, change a letter in a string. Methods that appear to do that, like `upper`, actually create new strings:

```
name = "Peter"
name Capitalized = name.upper()
print(Capitalized)
Peter
print(name)
Peter
```

Because strings are immutable, it is only possible to use an expression of the form `s[i] = value` at index `i` on the right side of the assignment operator.

### 8.3 Operations on Lists

[Section 3.1, Previous This Python Primer: Functions on page 57](#), and [Operations on Strings](#) on page 60, introduced a few of Python’s built-in functions. Some of these, such as `len`, can be applied to lists, as well as others we haven’t seen before. (See the following table.)

| Function               | Description                                                                                                                         |
|------------------------|-------------------------------------------------------------------------------------------------------------------------------------|
| <code>len(L)</code>    | Return the number of items in list <code>L</code> .                                                                                 |
| <code>max(L)</code>    | Return the maximum value in list <code>L</code> .                                                                                   |
| <code>min(L)</code>    | Return the minimum value in list <code>L</code> .                                                                                   |
| <code>sum(L)</code>    | Return the sum of the values in list <code>L</code> .                                                                               |
| <code>sorted(L)</code> | Return a copy of list <code>L</code> where the items are in order from smallest to largest. (This does not mutate <code>L</code> .) |

Table 8—List Functions

Here are some examples. The half-life of a radioactive substance is the time taken for half of it to decay. After twice this time has gone by, three quarters of the material will have decayed; after three times, seven eighths, and so on.

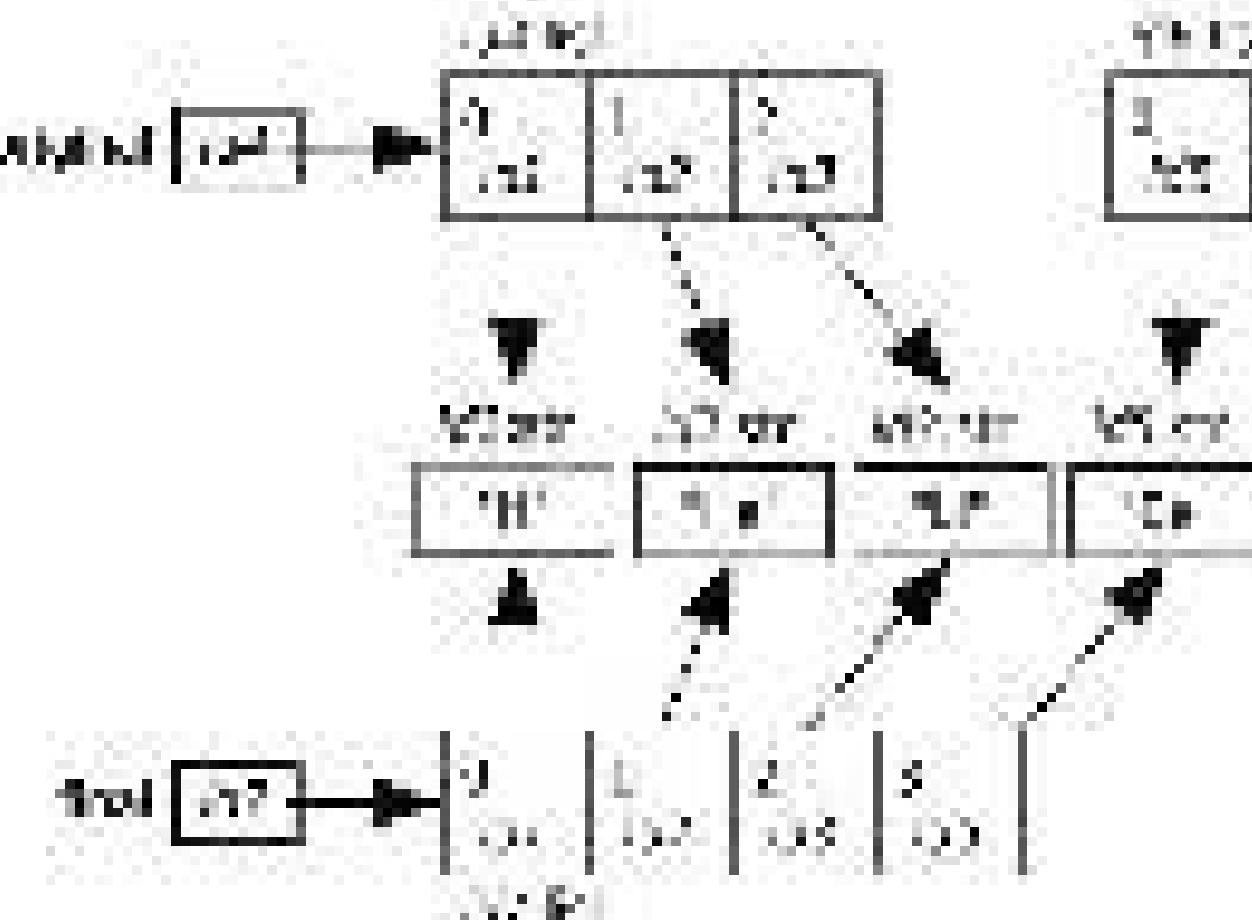
An isotope is a form of an element: plutonium has several isotopes, and each has a different half-life. Here are some of the built-in functions in action working on a list of the half-lives of plutonium isotopes Pu-239, Pu-238, Pu-240, Pu-241, and Pu-242:

```
>>> half_lives = [1587.7, 14166.6, 8863.0, 14, 373366.0]
>>> len(half_lives)
5
>>> max(half_lives)
373366.0
>>> min(half_lives)
14
>>> sum(half_lives)
392366.7
>>> sorted(half_lives)
[14, 1587.7, 8863.0, 14166.6, 373366.0]
>>> half_lives
[373366.0, 14166.6, 8863.0, 14, 1587.7]
```

In addition to built-in functions, some of the operations that we have seen can also be applied to lists. Like strings, lists can be combined using the concatenate (+) operator:

```
>>> original = ['H', 'He', 'Li']
>>> final = original + ['Be']
>>> final
['H', 'He', 'Li', 'Be']
```

This code doesn't create a list of the original list objects. Instead, it creates a new list whose entries refer to the items in the original list.



A list, tuple, dict, and Python compilation. If you tried to add a value of some type in an inappropriate way. For example, an error occurs when the concatenation operator is applied to a list and a string:

```
<in> ['Hi', 'Hi', 'Hi'] + 'Bye'
Traceback (most recent call last):
  File "testfile.py", line 1, in <module>
    TypeError: can only concatenate list (not "str") to list
```

You can also multiply a list by an integer to get a new list containing the elements from the original list repeated that number of times:

```
<in> metals = ['Fe', 'Ni']
<in> metals * 3
['Fe', 'Fe', 'Fe', 'Ni', 'Ni', 'Ni']
```

As with concatenation, the original list isn't modified; instead, a new list is created.

One operator that checks whether a list is in, which stands for delete, can be used to remove an item from a list, as follows:

```
<in> metals = ['Fe', 'Ni']
<in> del metals[0]
<in> metals
['Ni']
```

## The In Operator on Lists

The in operator can be applied to lists to check whether an object is in a list:

```
<in> nobles = ['Buck', 'Even', 'Regis', 'Kappas', 'Sures', 'Tudor']
<in> gas = input('Enter a gas: ')
Enter a gas: nitrogen
<in> if gas in nobles:
...     print('{} is noble.'.format(gas))
...
A gas is noble.
<in> gas = input('Enter a gas: ')
Enter a gas: nitrogen
<in> if gas in nobles:
...     print('{} is noble.'.format(gas))
...

```

Unlike with strings, when used with lists, the in operator checks only for a single item; it does not check for substrates. This code checks whether the list [1, 2] is in item in the list [1, 1, 2, 3]:

```
<in> [1, 2] in [1, 1, 2, 3]
False
```

## 8.4 Slicing Lists

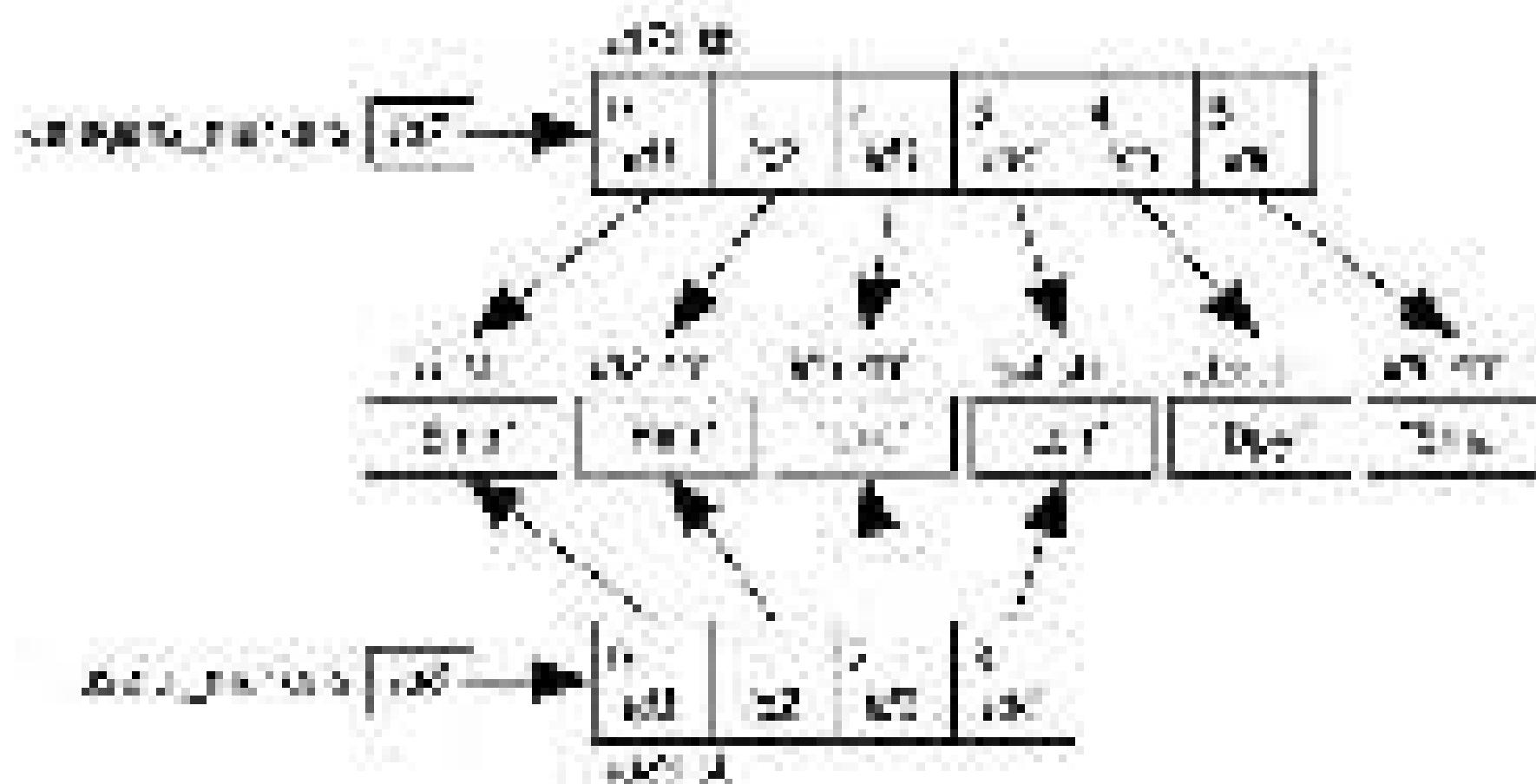
Consider the list `c. elegans_phenotypes` (available as a file of integers.gph), which contains three-letter string markers. Examples include `Dpy` (representing hermaphrodite with high frequency of males), `Unc` (representing both `Unc` (short and fat), `unc` (small), and `unc` (long)). We can have a look:

```
>>> c. elegans_phenotypes = [ 'Cib', 'Hox', 'Unc', 'Lav', 'Dpy', 'Sam' ]
>>> c. elegans_phenotypes
['Cib', 'Hox', 'Unc', 'Lav', 'Dpy', 'Sam']
```

It turns out that `Dpy` contains `unc`. Both worms are difficult to distinguish from each other as they will be nearly indistinguishable in mating status. We can produce a new list based on `c. elegans_phenotypes` but without `Dpy` or `unc` by taking a slice of the list:

```
>>> c. elegans_phenotypes = [ 'Cib', 'Hox', 'Unc', 'Lav', 'Dpy', 'Sam' ]
>>> useful_markers = c. elegans_phenotypes[0:4]
```

This creates a new list consisting of only the interesting markers, which are the first four items from the list. The `useful_markers` looks like:



The first index in the slice is the **starting point**. The second index is one more than the index of the last item we want to include. For example, the last item we wanted to include, `Lav`, had an index of 3, so we use 4 for the second index. Note: interestingly, `[0:4]` is a slice of the original list from index 0 (inclusive) up to, but not including, index 4 (exclusive). Python uses this convention to be consistent with the rule that the legal indices for a list go from 0 up to one less than the length.

The first index can be omitted if we want to slice from the beginning of the list, and the last index can be omitted if we want to slice to the end:

```

>>> catlogue.phenotypes = ['Ebb', 'Mid', 'Low', 'Ext', 'Top', 'Bottom']
>>> catlogue.phenotypes[::]
['Ebb', 'Mid', 'Low', 'Ext', 'Top', 'Bottom']
>>> catlogue.phenotypes[4:]
['Ext', 'Top', 'Bottom']

```

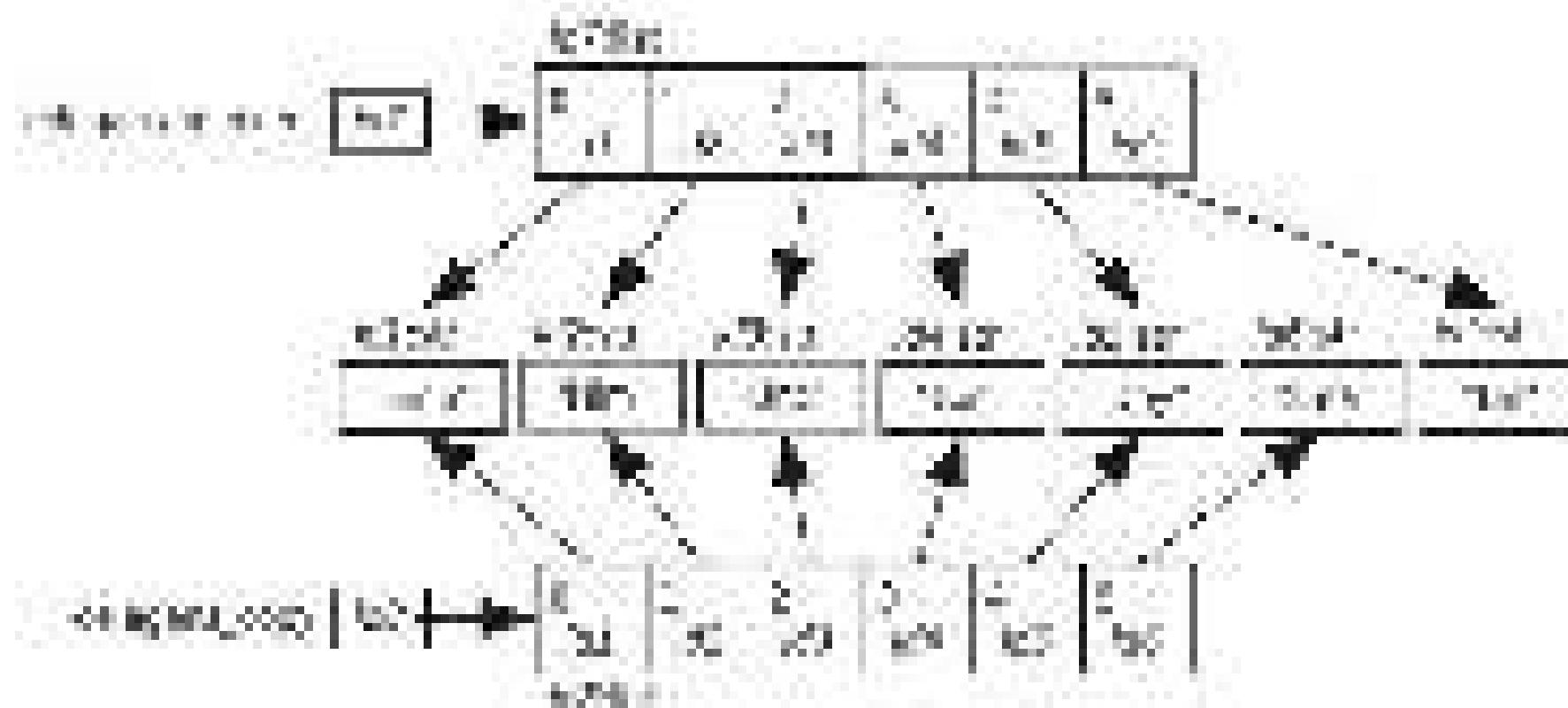
In each iteration of the while loop, `list_index` indicates that the “slice” runs from the start of the list to its end:

```

>>> catlogue.phenotypes = ['Ebb', 'Mid', 'Low', 'Ext', 'Top', 'Bottom']
>>> catlogue_copy = catlogue.phenotypes[::]
>>> catlogue.phenotypes[5] = 'Ext'
>>> catlogue.phenotypes
['Ebb', 'Mid', 'Low', 'Ext', 'Top', 'Ext']
>>> catlogue_copy
['Ebb', 'Mid', 'Low', 'Ext', 'Top', 'Bottom']

```

The list referred to by `catlogue_copy` is a **copy** of the list referred to by `catlogue.phenotypes`. The lists have the same items, but the lists themselves are different objects at different memory addresses:



In Section 8.6, `List Methods` on page 140, you will learn about a list method that can be used to make a copy of a list.

## 8.5 Aliasing: What's In a Name?

An alias is an alternative name for something. In Python, two variables are said to be aliases when they contain the same memory address. For example, the following code creates two variables, `huh` and `huh2`, both of which refer to a single list:



When we modify the last element of the variable, it affects all the other variables since they share the change as well:

```
>>> catlogue_phenotypes = ['Bob', 'Bis', 'Bic', 'Lor', 'Dps', 'Sav']
>>> catlogue_oilies = catlogue_phenotypes
>>> catlogue_phenotypes[5] = 'Sal'
>>> catlogue_phenotypes
['Bob', 'Bis', 'Bic', 'Lor', 'Dps', 'Sal']
>>> catlogue_oilies
['Bob', 'Bis', 'Bic', 'Lor', 'Dps', 'Sal']
```

Absurd! One of the reasons why the notion of mutability is important. For example, if *x* and *y* refer to the same list, then any changes you make to the list through *x* will be "seen" by *y*, and vice versa. This can lead to all sorts of hard-to-debug errors in which a value changes but the code doesn't, even though your program doesn't appear to contain anything weird. This can't happen with immutable values like strings. Since a string can't be changed after it has been created, it's safe to have a variable for it.

## Mutable Parameters

Absurd occurs when you use list parameters as well, since parameters are variables. Here is a simple function that takes a list, removes its last item, and returns the list:

```
>>> def remove_last_item(L):
...     """ (List) --> List
...
...     Return list L with the last item removed.
...
...     Precondition: len(L) >= 0
...
...     See remove_last_item([1, 2, 3, 4])
...     [1, 2, 3]
...
...     def L[1:]
...     return L
...
...     """

```

In the code that follows, a list is created and stored in a variable; then that variable is passed as an argument to `remove_last_item`:

```
>>> catlogue_markans = ['Bob', 'Bis', 'Bic', 'Lor', 'Dps', 'Sal']
>>> remove_last_item(catlogue_markans)
['Bob', 'Bis', 'Bic', 'Lor', 'Dps']
>>> catlogue_markans
['Bob', 'Bis', 'Bic', 'Lor', 'Dps']
```

When the call to `remove` is made, it takes the expected parameter `L` by example and the memory address that `colours` `listType` contains. That makes `colours` `listType`, not `L`. Likewise, when the last item of the list that `L` refers to is removed, the change is “seen” by `colours` as well.

Since `removeLast` modifies the list parameter, the modified list doesn't actually need to be returned. You can remove the return statement:

```
>>> def removeLast(L):
...     """ (List) --> NoneType
...
...     Remove the last item from L.
...
...     Precondition: len(L) >= 0
...
...     >>> removeLast([1, 2, 3])
...     >>>
...     [1, 2]
...
...     >>> colours.pop()
...     'black'
...     >>> colours
...     ['red', 'orange', 'green', 'blue']
```

As we'll see in [Section A.6: List Methods](#) on page 143, several methods modify a list and return None like the second version of `removeLast`.

## 8.6 List Methods

Lists are objects and thus have methods. Table 10: List Methods on page 141 gives some of the most commonly used list methods.

Here is a sample interaction showing how we can use list methods to construct a list of many colors:

```
>>> colors = ['red', 'orange', 'green']
>>> colors.append('black')
>>> colors
['red', 'orange', 'green', 'black', 'blue']
>>> colors.append('purple')
>>> colors
['red', 'orange', 'green', 'black', 'blue', 'purple']
>>> colors.insert(2, 'yellow')
>>> colors
['red', 'orange', 'yellow', 'green', 'black', 'blue', 'purple']
>>> colors.remove('black')
>>> colors
['red', 'orange', 'yellow', 'green', 'blue', 'purple']
```

| Method                        | Description                                                                                                                                                                                                                                                                                                                        |
|-------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>append(v)</code>        | Appends value <code>v</code> to list <code>L</code> .                                                                                                                                                                                                                                                                              |
| <code>clear()</code>          | Removes all items from list <code>L</code> .                                                                                                                                                                                                                                                                                       |
| <code>count(v)</code>         | Returns the number of occurrences of <code>v</code> in list <code>L</code> .                                                                                                                                                                                                                                                       |
| <code>extend(L)</code>        | Appends the items in <code>v</code> to <code>L</code> .                                                                                                                                                                                                                                                                            |
| <code>index(v)</code>         | Returns the index of the first occurrence of <code>v</code> in <code>L</code> . An error is raised if <code>v</code> doesn't occur in <code>L</code> .                                                                                                                                                                             |
| <code>insert(index, v)</code> | Changes the index of the first occurrence of <code>v</code> after index <code>index</code> . An error is raised if <code>v</code> doesn't occur in that part of <code>L</code> .                                                                                                                                                   |
| <code>pop(index)</code>       | Returns the index of the first occurrence of <code>v</code> between indices <code>index</code> (included) and <code>index+1</code> (not included). An error is raised if <code>v</code> doesn't occur in that part of <code>L</code> .                                                                                             |
| <code>remove(v)</code>        | Removes and returns the last item of <code>L</code> , which must be nonempty.                                                                                                                                                                                                                                                      |
| <code>reverse()</code>        | Reverses the order of the values in list <code>L</code> .                                                                                                                                                                                                                                                                          |
| <code>sort()</code>           | Sorts the values in list <code>L</code> . In ascending order for strings with the same letter case, it works in alphabetical order; works for values in list <code>L</code> in descending order (for strings with the same letter case, it sorts in reverse alphabetical order).                                                   |
| <code>sort(key=func)</code>   | Sorts the values in list <code>L</code> according to function <code>func</code> . Function <code>func</code> appends a value <code>v</code> to a list <code>list</code> and returns the item last inserted in <code>list</code> . In fact, the only method that returns a list is <code>copy</code> , which is implemented in [1]. |

Table 10—List Methods

All the methods shown above modify the list instead of creating a new list. This same is true for the methods `sort`, `reverse`, `sort`, and `copy`. All these methods, except `copy`, return a value either `None` or `v` (it retains the item last inserted from the list). In fact, the only method that returns a list is `copy`, which is implemented in [1].

Finally, we will now understand the usage of `copy`. First, append appends a value `v` to the list `L` and returns no value. Second, append modifies the list rather than creating a new one.

## Where Did My List Go?

Any operation, no matter how short, that methods return have either a *returning* and *returning a value* or a *return value* to happen.

Let's take a look at some code showing what happens if you do this:

```
def add_to_list():
    colors = []
    colors.append("green")
    colors.append("purple")
    colors.append("blue")
    colors.append("orange")
    return colors
```

In this example, we want to add four things: the items in the list and the returned `list` value. That's why `variables` take `color` values to them. Similarly, even in the `add_to_list` function, we're still *not* adding the list:

```
def add_to_list():
    colors = []
    colors.append("green")
    colors.append("purple")
    colors.append("blue")
    colors.append("orange")
    return colors
```

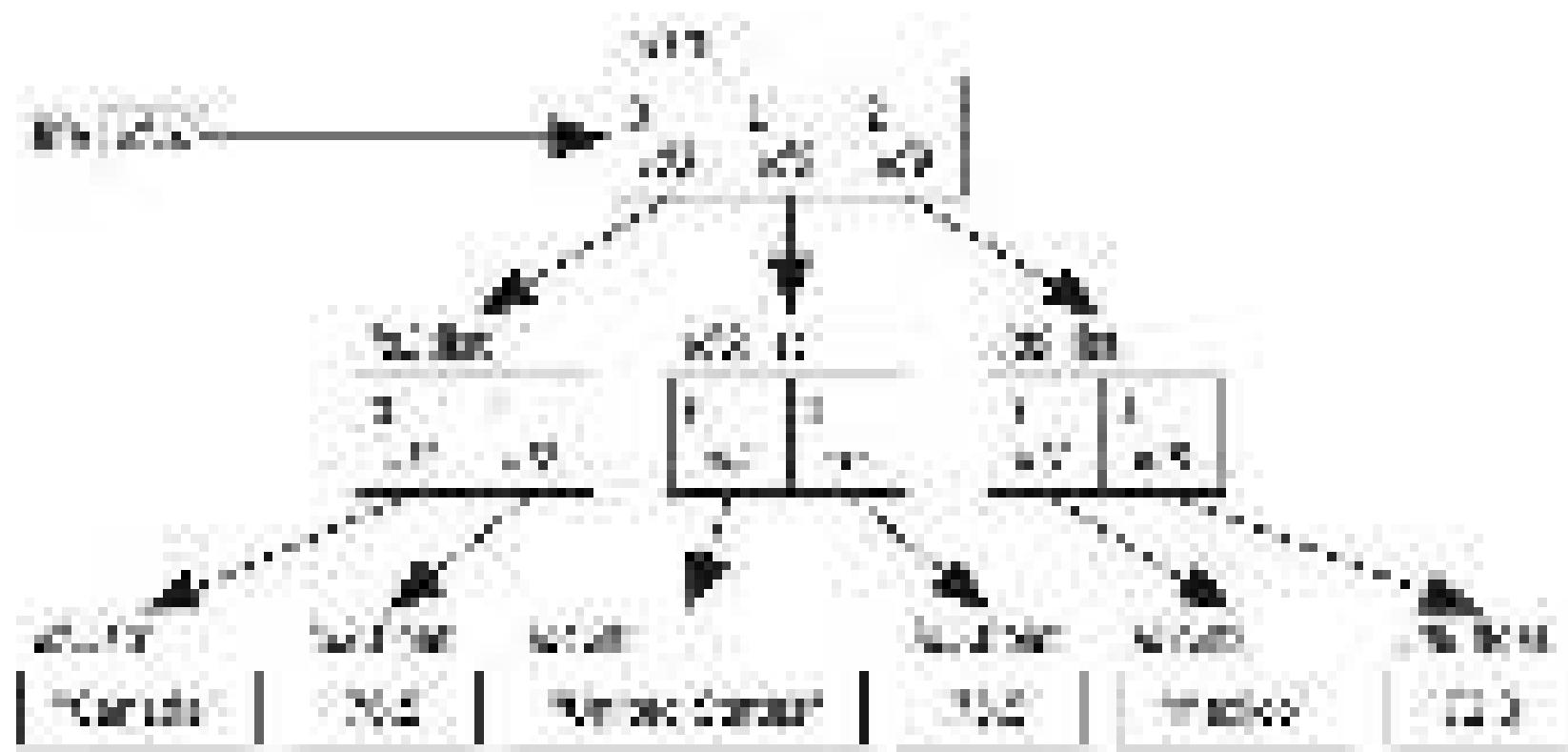
Methods that modify a collection without a *return* statement, it's a common error to expect that they'll return the resulting list. As well, don't assume that `return` records the modification; on page 108, remember that these can be racey (overwriting and so on). See below.

## 8.7 Working with a List of Lists

We saw in [List Comprehensions](#), on page 132, that lists can contain any type of data. That means that they can contain other lists. A list whose items are lists is called a *nested list*. For example, the following nested list describes three express routes in different countries:

```
route_table = [
    ["Canada", 78.51, "United States", 79.51, "Russia", 72.01]
```

Here is the memory model that results from execution of that assignment statement:



Note that each item in the outer list is itself a list of two things. We use the standard indexing notation to access the items in the outer list.

```
>>> life = [['Canada', 76.5], ['United States', 79.5], ['Russia', 72.5]]
>>> life[0]
['Canada', 76.5]
>>> life[0]
['United States', 76.5]
>>> life[0][0]
'Canada'
```

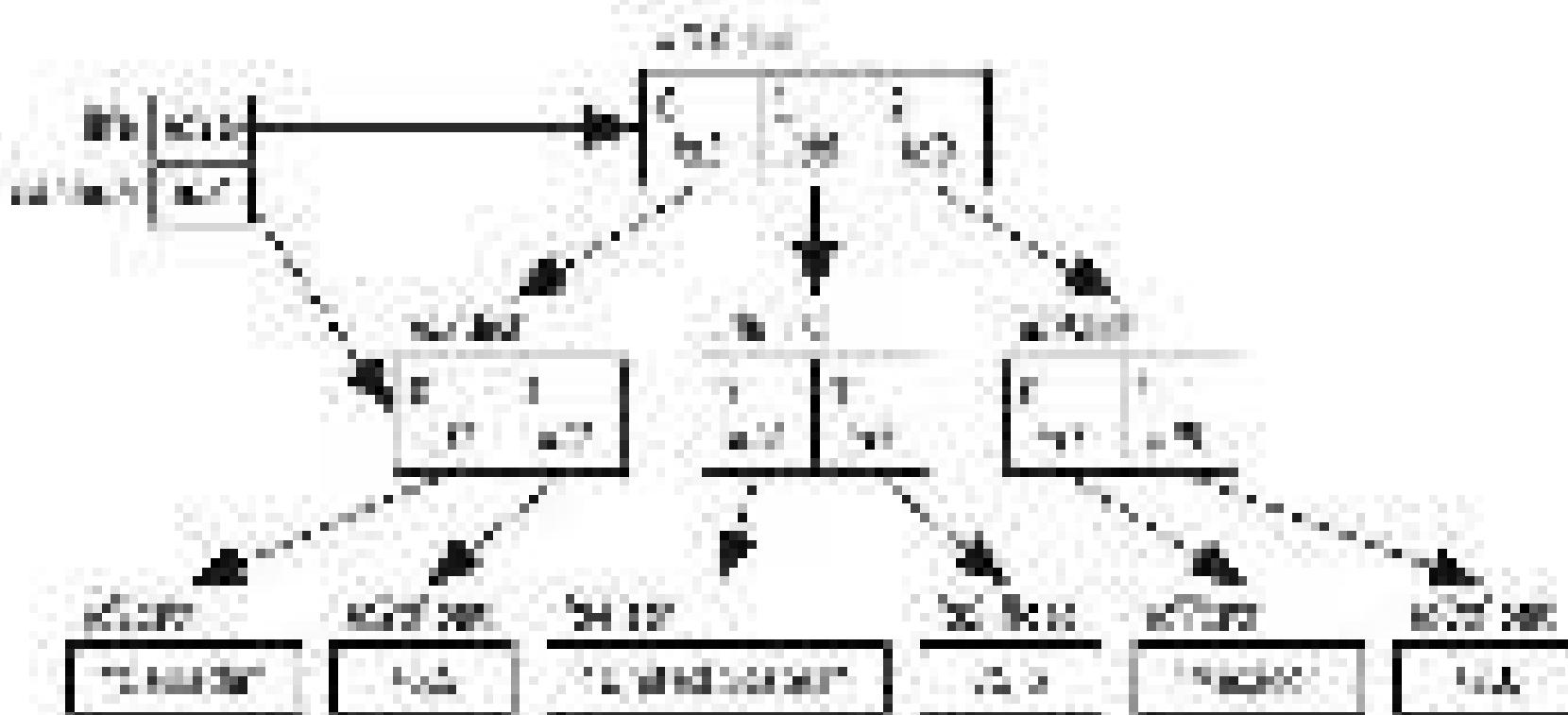
Since each of those items is also a list, we can index it again, just as we can chain together method calls or nest function calls:

```
>>> life = [['Canada', 76.5], ['United States', 79.5], ['Russia', 72.5]]
>>> life[0]
['United States', 76.5]
>>> life[0][0]
'United States'
>>> life[0][0]
76.5
```

We can also **modify** variables by redefinition:

```
>>> life = [['Canada', 76.5], ['United States', 79.5], ['Russia', 72.5]]
>>> canada = life[0]
>>> canada
['Canada', 76.5]
>>> canada[0]
'Canada'
>>> canada[0]
'Russia'
>>> canada[0]
76.5
```

Assigning a symbol to a variable creates no alias for that object:



As before, any change we make through the sublist reference will be seen when we access the main list and vice versa.

```

>>> lista = ['Canada', 'US'], ['United States', 'US'], ['USA', 'US'])
>>> Canada = lista[0]
>>> Canada[1] = 'USA'
>>> Canada
['Canada', 'USA']
>>> lista
[['Canada', 'USA'], ['United States', 'US'], ['USA', 'US']])

```

## 8.8 A Summary List

In this chapter, you learned the following:

- Lists are used to keep track of zero or more objects. The objects in a list are called items or elements. Each item has a position in the list called an index and that position ranges from zero to one less than the length of the list.
- Lists can contain any type of data, including other lists.
- Lists are mutable, which means that their contents can be modified.
- Slicing is used to create new lists that have the same values as a subset of the values of the originals.
- When two variables refer to the same object, they are called aliases.

## 8.9 Exercises

There are some exercises for you to try on your own. Solutions are available at [https://nostarch.com/pythonfordata](#).

1. Variable `idolists` refers to the list [`Barney`, `Fred`, `Chewie`, `Finn`, `Han`]. After creating `idolists` and either **Appending** or **Indenting** with positive indices, write expressions that produce the following:
  - The first item of `idolists`
  - The last item of `idolists`
    - The last [`Barney`, `Fred`, `Chewie`, `Finn`, `Han`]
  - The last [`Chewie`, `Fred`, `Barney`]
  - The last [`Barney`, `Adam`]
  - The empty list
2. Repeat the previous exercise using negative indices.
3. Variable `expenses` refers to the list [`300`, `1000`, `1400`, `1500`, `1530`]. An appointment is scheduled for `1000`, so `1000` needs to be sliced in the list.

6. Once the method appears, add `print` to the end of the last line again. Then execute:
- Instead of using `append`, use the `+` operator to add '100' to the end of the list that `append` refers to.
  - You used two approaches to add '100' to the list. Which approach would add the list and which approach created a new list?
7. Variables are passed to the list [235, 2214, 2355, 230, 2373, 1331] using list methods, do the following:
- Remove 235 from the list.
  - Get the index of 235.
  - Insert 1440 in the list after 235.
  - Extend the list by adding 2346, 1830 to it.
  - Reverse the list.
  - Sort the list.
8. In this exercise, you'll create a list and then answer questions about that list.
- Add each of the following elements—the atomic numbers of the six alkaline earth metals—barium (56), magnesium (12), calcium (20), strontium (38), beryllium (12), and radium (88)—to a variable called `alkaline_earth`.
  - Which index contains calcium's atomic number? Write the answer in two ways, one using a positive index and one using a negative index.
  - Which iteration will you have to use to print all the items? `for i in range(6)`?  
or `for i in range(len(alkaline_earth))`?
  - What's each item in the list? Hint: the highest atomic number in `alkaline_earth` is 88. It has 12 atoms in the same sense from Table 8, Atom Numbers on page 134.)
9. In this exercise, you'll create a list and then answer questions about that list.
- Create a list of temperatures in degrees Celsius with the values 25.2, 35.8, 21.4, 22.9, 30.1, 33.5, and 18.8, and assign it to a variable called `temp`.
  - Using one of the list methods, sort `temp` in ascending order.
  - Using `filter`, create two new lists, `hot` and `not_hot`, which contain the temperatures below and above 30 degrees Celsius respectively.
  - Using list `append`, append `cool` to `not_hot` with `temp` in between, and call it `tempx`.  
`tempx = cool + temp + not_hot`.
10. Complete the examples in the doctests and then write the body of the following function:

```
def sum_first(L):
    """ (List) --> float
    Returns True if and only if the first item of the list is the same as the last.
    """

```

```
>>> sum_first([1, 4, 2, 3, 2])
True
>>> sum_first(['apple', 'banana', 'pear'])
False
>>> sum_first([4.2, -4.5])
False
>>>
```

6. Complete the examples in the doctests and then write the body of the following functions:

```
def is_longer(L1, L2):
    """ (List, List) --> bool
    Returns True if and only if the length of L1 is longer than the length of L2.
    """
    >>> is_longer([2, 2, 3], [4, 5])
    False
    >>> is_longer(['apple', 'pear', 'peach'])
    True
    >>> is_longer([1, 2, 3])
    False
    >>>
```

7. Draw a memory model showing the effect of the following statements:

```
values = [2, 3, 5]
values[1] = values
```

- (a) Variable `values` refers to the nested list [`2`, `3`, `5`], [`apple`], or [`pear`, `peach`]. Using `values` and either slicing or indexing with positive indices, make experiments that produce the following:

- The first item of `values` is the last item of `list`
- The last item of `values` is the last item of `list`
- The string `apple`
- The string `peach`
- The last index of `apple`
- The last big `peach`

- (b) Repeat the previous exercise using negative indices.

# Repeating Code Using Loops

This chapter introduces another fundamental kind of control flow: repetition. Up to now, to calculate an IRS return two hundred times, you would need to write that instruction two hundred times. Now you'll see how to write the instruction once and use loops to repeat that code the desired number of times.

## 9.1 Processing Items in a List

With what you've learned so far, to print the items from a list of velocities of falling objects in metric and Imperial units, you would need to write a call on function `print` for each item in the list:

```
velocities = [0.0, 0.61, 10.61, 20.43]
for print('Metric:', velocities[0], 'm/sec',
... 'Imperial:', velocities[0] * 3.28, 'ft/sec')
Metric: 0.0 m/sec Imperial: 0.0 ft/sec
for print('Metric:', velocities[1], 'm/sec',
... 'Imperial:', velocities[1] * 3.28, 'ft/sec')
Metric: 0.61 m/sec Imperial: 2.0 m/sec
for print('Metric:', velocities[2], 'm/sec',
... 'Imperial:', velocities[2] * 3.28, 'ft/sec')
Metric: 10.61 m/sec Imperial: 34.3558 ft/sec
for print('Metric:', velocities[3], 'm/sec',
... 'Imperial:', velocities[3] * 3.28, 'ft/sec')
Metric: 20.43 m/sec Imperial: 66.7344 ft/sec
```

The code above is messy. It processes a list with just four values. Imagine processing a list with a thousand values. Lists are essential so that you wouldn't have to create a thousand variables to store a thousand values. For the same reason, Python has a `for` loop that lets you process each element in a list in turn without having to write one statement per element. You can use it in `for` to print the velocities.

```

>>> velocities = [0.0, 9.81, 19.62, 29.43]
>>> for velocity in velocities:
...     print(velocity, 'm/sec')
...     'Impulse', velocity * 0.01, 'Nsec')
...
Hectic: 0.0 m/sec. Impulse: 0.0 Nsec
Hectic: 9.81 m/sec. Impulse: 99.09 Nsec
Hectic: 19.62 m/sec. Impulse: 198.18 Nsec
Hectic: 29.43 m/sec. Impulse: 297.27 Nsec

```

The general form of a for loop over a list is as follows:

```

for <var> in <list>:
    <body>

```

A for loop is executed as follows:

- The loop variable is assigned the first item in the list and the loop block—the body of the for loop—is executed.
- The loop variable is then assigned the second item in the list and the loop body is executed again.
- ...
- Finally, the loop variable is assigned the last item of the list and the loop body is executed one last time.

As we saw in [Section 3.3, “Defining Our Own Functions on page 118](#), a `for` loop is just a sequence of one or more statements. Each pass through the block is called an iteration, and at the start of each iteration, Python assigns the next item in the list to the loop variable. As with function definitions and if statements, the statements in the loop block are indented.

In the code above, before the first iteration, variable `velocity` is assigned `velocities[0]` and then the loop body is executed. Before the second iteration it is assigned `velocities[1]`, and then the loop body is executed and so on. In this way, the program can do something with each item in turn. [Table 3.1, “Looping Over the Elements on page 118](#), summarizes the value of `velocity` at the start of each iteration, as well as what is printed during that iteration.

In the previous example, we created a new variable, `velocity`, to refer to the current item in the list inside the loop. We could equally well have used an existing variable:

If we use an existing variable, the loop will share with the variable referring to the new element of the list. The content of the variable before the loop is lost, exactly as if we had used an assignment statement to give a new value to that variable:

[View solution](#)

| Iteration | List Item Selected by Index of Iteration | What Is Printed During This Iteration      |
|-----------|------------------------------------------|--------------------------------------------|
| 0st       | velocity[0]                              | Meter: 10 miles: imperial: 0.0 furl.       |
| 1nd       | velocity[1]                              | Meter: 3.61 miles: imperial: 12.125 furl.  |
| 2nd       | velocity[2]                              | Meter: 10.62 miles: imperial: 34.755 furl. |
| 3rd       | velocity[3]                              | Meter: 29.87 miles: imperial: 95.575 furl. |

**Table 9.1—Looping Over List Velocities**

```

>>> speed = 2
>>> velocities = [10.0, 3.61, 10.62, 29.87]
>>> for speed in velocities:
...     print('Meter: ', speed, 'miles')
...
Meter: 10.0 miles
Meter: 3.61 miles
Meter: 10.62 miles
Meter: 29.87 miles
>>> print('Final: ', speed)
Final: 29.87

```

The variable is left holding its last value when the loop finishes. Notice that the last print statement isn't indented, so it is not part of the for loop. It is executed only once, after the for loop execution has finished.

## 9.2 Processing Characters in Strings

It is also possible to loop over the characters of a string. The general form of a for loop over a string is as follows:

```

for character in string:
    character

```

As with a for loop over a list, the loop variable gets assigned a new value at the beginning of each iteration. In the case of a for loop over a string, the variable is assigned a single character.

For example, we can loop over every character in a string, printing the uppercase letters:

```

>>> country = 'United States of America'
>>> for ch in country:
...     if ch.isupper():
...         print(ch)
...
U
S
A

```

In the code above, variable `i` is assigned to `list[0]` before the first iteration, causing it to be the second, and so on. The loop iterates twenty-four times (four per character) and the `for` statement block is repeated three times (over your uppercase letters).

### 9.3 Looping Over a Range of Numbers

We can also loop over a range of values. This allows us to perform tasks a certain number of times and to do more sophisticated processing of lists and strings. To begin, we must first determine the range of numbers over which to iterate.

#### Generating Ranges of Numbers

Python's built-in function `range` produces an object that will generate a sequence of integers. When passed a single argument, as in `range(10)`, the argument starts at 0 and continues to the integer before step.

```
>>> range(10)
range(0, 10)
```

This is the first time that you've seen Python's `range` type. You can use a loop to access each number in the sequence one at a time:

```
>>> for num in range(10):
...     print(num)
...
0
1
2
3
4
5
6
7
8
9
```

To get the numbers from the sequence all at once, we can use built-in function `list` to create a list of those numbers:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8]
```

Here are some more examples:

```
>>> list(range(13))
[0, 1, 2]
>>> list(range(1))
[0]
>>> list(range(0))
[]
```

The generator produced includes the start value and excludes the stop value, which is technically consistent with how sequence indexing works: the expression `[1:5]` takes a slice of up to, but not including, the value at index 5.

Notice that in the code above, we call `list()` on the value produced by the call on `range()`. Function `range()` returns a range object, and we create a list based on the values it generates, working with it using the set of the operations and methods we are already familiar with.

Function `range()` can also be passed two arguments, where the first is the start value and the second is the stop value:

```
>>> list(range(1, 5))
[1, 2, 3, 4]
>>> list(range(1, 10))
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(5, 10))
[5, 6, 7, 8, 9]
```

By default, function `range()` generates numbers that successively increase by one—the `step` is called the step size. You can specify a different step size for ranges with an optional third parameter:

Here we generate a list of leap years in the first half of this century:

```
>>> list(range(2000, 2050, 4))
[2000, 2004, 2008, 2012, 2016, 2020, 2024, 2028, 2032, 2036, 2040, 2044, 2048]
```

The step size can also be negative, which produces a decreasing sequence. When the step size is negative, the starting index should be longer than the ending index:

```
>>> list(range(2050, 2000, -4))
[2050, 2046, 2042, 2038, 2034, 2030, 2026, 2022, 2018, 2014, 2010, 2006, 2002]
```

Otherwise, `range()` result will be empty:

```
>>> list(range(2000, 2050, -4))
[]
>>> list(range(2050, 2000, 4))
[]
```

We’re still no longer over the sequence produced by a call on `range()`. For example, the following program calculates the sum of the integers from 1 to 100:

```
>>> total = 0
>>> for i in range(1, 101):
...     total = total + i
...
>>> total
5050
```

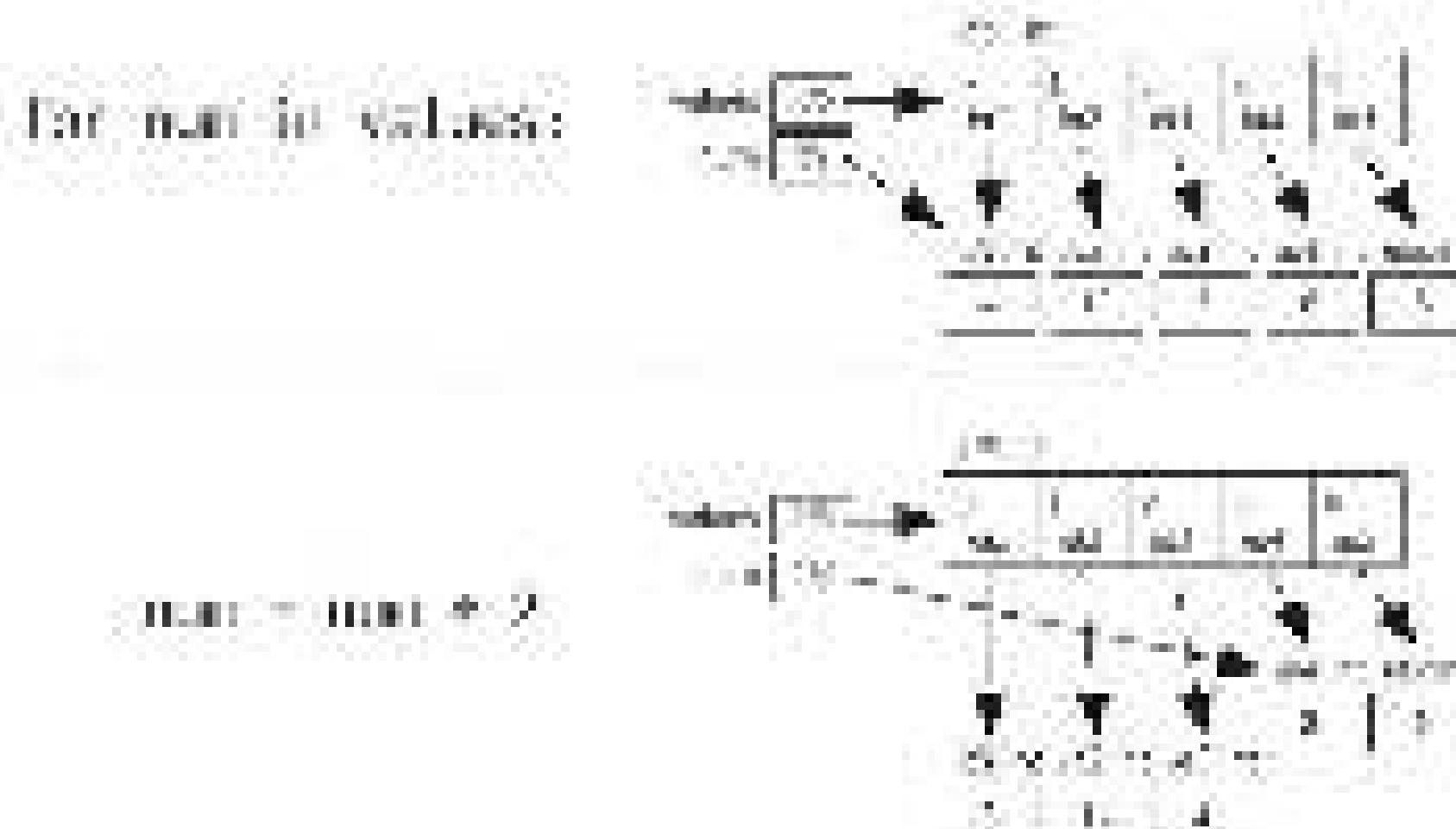
Note that the upper bound passed to `range` is 10.1. It is one more than the greatest integer we actually want.

## 9.4 Processing Lists Using Indices

The loops that lists show how we have written for loops that must access individual items. But what if we want to change the items in a list? For example, suppose we want to double all of the values in a list. The following doesn't work:

```
list values = [6, 10, 8, 8, -6]
for num in values:
    ...
    num = num * 2
...
list values
[6, 10, 8, 8, -6]
```

Each loop iteration assigns the current list value to variable `num`. Doubling that value inside the loop changes what `num` refers to, but it doesn't update the list object.



To work, we can use another point to show how the value that `num` refers to changes during each iteration.

```
list values = [6, 10, 8, 8, -6]
for num in values:
    ...
    num = num * 2
    print(num)

6
10
8
8
-6
list print(values)
[6, 10, 8, 8, -6]
```

The natural approach is to loop over the indices of the list. If variable `index` refers to a list, then `index[0]` is the number of items it contains, and the expression `range(len(index))` generates a sequence containing exactly the indices for values:

```
var values = [4, 10, 3, 8, -6]
var lenvalues
3
var list1 = range(5)
10, 1, 2, 3, 4
var list1 = range(lenvalues))
10, 1, 2, 3, 4
```

The list `list1` value refers to has five items, so the indices are 0, 1, 2, 3, and 4. Rather than looping over `values`, you can iterate over the indices, which are provided by `range(len(values))`:

```
var values = [4, 10, 3, 8, -6]
var for i in range(len(values)):
    print(i)

0
1
2
3
4
```

Note that we called the variable `i`, which is a shorter name. You can then use each index to access the items in the list:

```
var values = [4, 10, 3, 8, -6]
var for i in range(len(values)):
    ...
    print(i, values[i])

0 4
1 10
2 3
3 8
4 -6
```

You can also use them directly as names:

```
var values = [4, 10, 3, 8, -6]
var for i in range(len(values)):
    ...
    values[i] = values[i] + 2
    ...
var values
10, 12, 5, 10, -4
```

Evaluation of the expression on the right side of the assignment looks up the value at index 1 and multiplies it by two. Python then assigns that value to the item at index 1 in the list. When I run this in Jupyter, for example, `weights[1]` is 10, which is multiplied by 2 to produce 20. The list item `weights[1]` is then assigned 20.

## Processing Parallel Lists Using Indices

Sometimes the data from one list corresponds to data from another. For example, consider the following lists:

```
metals = ['Li', 'Be', 'B']  
weights = [6.941, 19.3000000, 10.811]
```

The item at index 0 of `metals` has its atomic weight at index 0 of `weights`. The same is true for the items at index 1 in the two lists, and so on. These lists are parallel lists, because the item at index *i* of one list corresponds to the item at index *i* of the other list.

We would like to print each metal and its weight. To do this, we can loop through each index of the lists, accessing the items in such:

```
metals = ['Li', 'Be', 'B']  
weights = [6.941, 19.3000000, 10.811]  
for i in range(len(metals)):  
    print(metals[i], weights[i])  
  
Li 6.941  
Be 19.3000000  
B 10.811
```

This code does very well when the `metals` and `weights` lists have the same length, as in this case. If the length of `weights` is less than the length of `metals`, then an error would occur when trying to access an index of `weights` that doesn't exist. For example, if `metals` has three items and `weights` only has two, the first two print statements will be executed, but during the third function call, an error would occur when evaluating the second argument.

## 9.5 Nesting Loops in Loops

The block of statements in one loop can contain another loop. In this task, the inner loop is executed once for each item of list `values`:

```
outer = ['Li', 'Be', 'B']  
inner = ['P', 'Cl', 'Br']  
for outer in outer:  
    for inner in inner:  
        print(outer + inner)
```

```
L1F
L1G
L1H
L2F
L2G
L2H
L3F
L3G
L3H
```

The number of times that function `print` is called is called its **nesting level**. In Table 12—**Nested Loops: Over Inner and Outer Loops**, we show that for each iteration of the outer loop (it's `i`), we can determine which iteration of the inner loop (we call it `j`) has been performed.

| Iteration of<br>Outer Loop | What inner loop<br>Refers To | Iteration of<br>Inner Loop | What halogen<br>Refers To | What's Printed |
|----------------------------|------------------------------|----------------------------|---------------------------|----------------|
| 1st                        | A[1][1]                      | 1st                        | A[1][1]                   | L1             |
|                            |                              | 2nd                        | A[1][2]                   | L1C            |
|                            |                              | 3rd                        | A[1][3]                   | L1F            |
| 2nd                        | A[2][1]                      | 1st                        | A[2][1]                   | HeF            |
|                            |                              | 2nd                        | A[2][2]                   | HeCl           |
|                            |                              | 3rd                        | A[2][3]                   | HeBr           |
| 3rd                        | A[3][1]                      | 1st                        | A[3][1]                   | F              |
|                            |                              | 2nd                        | A[3][2]                   | C              |
|                            |                              | 3rd                        | A[3][3]                   | Br             |

Table 12—Nested Loops: Over Inner and Outer Loops

Sometimes, an inner loop uses the same loop as the outer loop. An example of this is shown in the `sumEven` code. In general, a multi-dimensional table. After printing the header row, we can a nested loop to print each row of the table in turn, using tabs (see Table 6, **Escape Sequences**, on page 56) to make the columns line up.

```
def print_table():
    for row in some_type:
```

Print the multiplication table for numbers 1 through 9 (inclusive).

```
(cc) print_table()
```

|   |    |    |    |    |
|---|----|----|----|----|
| 1 | 2  | 3  | 4  | 5  |
| 1 | 2  | 3  | 4  | 5  |
| 2 | 4  | 6  | 8  | 10 |
| 3 | 6  | 9  | 12 | 15 |
| 4 | 8  | 12 | 16 | 20 |
| 5 | 10 | 15 | 20 | 25 |

```

❸ The numbers to include in the table:
numbers = [100, 200, 300, 400, 500]

❹ Print the header row:
for i in numbers:
    print(i**2 + str(i), end=' ')
print()

❺ Print each row number and the contents of each row:
for i in numbers:
    print(i, end=' ')
    for j in numbers:
        print(j**2 + str(j), end=' ')
    print()

```

Each iteration of the outer loop prints a row. Each row consists of a row number, a tab-number pair, and a newline. It's the inner loop's job to print the tabs and numbers' part of the row. For `[[1, 2, 3], [4, 5, 6]]`, let's take a closer look at what happens during the third iteration of the outer loop:

- ❶ It executes `i`, the third item of `numbers`.
- ❷ The row number, 3, is printed.
- ❸ This line of code is the inner loop header, and it will be executed three times. Before the first iteration of the inner loop, `j` is assigned 1; before the second iteration, `j` is assigned 2; and so on; until it is assigned 3 before its last iteration.
- ❹ Five times this line is executed, right after the previous line using whatever value `j` just assigned. The first time, it prints a tab followed by 1, then a tab followed by 4, and so on until it prints a tab followed by 16.
- ❺ Now that a row has been printed, the program prints a newline. This line of code occurs outside of the inner loop so that it's only executed once per row.

## Looping Over Nested Lists

In addition to looping over lists of numbers, strings, and Booleans, we can also loop over lists of lists. Here is an example of a loop over an outer list. The `langs` variable, which we've named `langs`, is assigned an item of nested lists at the beginning of each iteration:

<https://repl.it/@benjiboo/LoopingOverNestedLists#main.py>

```
<-- elements = [['Li', 'Na', 'K'], ['F', 'Cl', 'Br']]  
for inner_list in elements:  
...     print(inner_list)  
  
[['Li', 'Na', 'K'],  
 ['F', 'Cl', 'Br']]
```

To access each string in the nested lists, you can loop over the outer list and then over each inner list using a nested loop. Here, we print every string in every inner list:

```
<-- elements = [['Li', 'Na', 'K'], ['F', 'Cl', 'Br']]  
for inner_list in elements:  
...     for item in inner_list:  
...         print(item)  
  
Li  
Na  
K  
F  
Cl  
Br
```

In the code above, the outer loop variable, `inner_list`, refers to a list of strings, and the inner-loop variable, `item`, refers back working from that list.

When you have a nested list and you want to do something with every item in the inner lists, you need to use a nested loop.

## Looping Over Ragged Lists

Remember: most lists have to be the same length:

```
<-- info = [['Thomas Edison', 1847, 1880],  
...           ['Charles Darwin', 1809, 1882],  
...           ['Alan Turing', 1912, 1954, 'alanturing.net']]  
for item in info:  
...     print(item[0])  
  
Thomas Edison  
Charles Darwin  
Alan Turing
```

Nested lists with inner lists of varying lengths are called **ragged lists**. Ragged lists can be tricky to process in the data being uniform; the example trying to associate a list of email addresses to for data where some subjects have no email requires careful thought.

Repeating data does occur normally. For example, if a record is made each day of the time at which a person has a drink of water, each day will have a different number of entries:

```
>>> drinking_times_by_day = [[{"day": "2016-02-01", "time": "06:02"}, {"day": "2016-02-01", "time": "06:17"}, {"day": "2016-02-01", "time": "13:52"}, {"day": "2016-02-01", "time": "18:23"}, {"day": "2016-02-01", "time": "21:31"}, {"day": "2016-02-02", "time": "06:45"}, {"day": "2016-02-02", "time": "12:44"}, {"day": "2016-02-02", "time": "14:52"}, {"day": "2016-02-02", "time": "22:17"}, {"day": "2016-02-03", "time": "06:55"}, {"day": "2016-02-03", "time": "11:11"}, {"day": "2016-02-03", "time": "12:34"}, {"day": "2016-02-03", "time": "13:48"}, {"day": "2016-02-03", "time": "15:02"}, {"day": "2016-02-03", "time": "18:06"}, {"day": "2016-02-03", "time": "21:15"}, {"day": "2016-02-04", "time": "06:15"}, {"day": "2016-02-04", "time": "11:44"}, {"day": "2016-02-04", "time": "13:23"}, {"day": "2016-02-04", "time": "18:38"}, {"day": "2016-02-04", "time": "21:51"}, {"day": "2016-02-05", "time": "06:01"}, {"day": "2016-02-05", "time": "13:34"}, {"day": "2016-02-05", "time": "18:51"}, {"day": "2016-02-05", "time": "21:22"}]
```

```
>>> for day in drinking_times_by_day:
...     for drinking_time in day:
...         print(drinking_time, end=" ")
...     print()

0:02 16:17 13:52 18:23 21:31
0:45 12:44 14:52 22:17
0:55 11:11 12:34 13:48 15:02 18:06 21:15
0:15 11:44 16:38
18:38 18:58 18:48 19:08
0:02 11:16 13:02 18:48
0:01 13:34 18:51 21:22
```

The inner loop iterates over the items of `day`, and the length of that list varies.

## 9.6 Looping Until a Condition Is Reached

`for` loops are useful only if you know how many iterations of the loop you need. In some situations, it is not known in advance how many loop iterations to execute. In a game program, for example, you don't know whether a player is going to want to play again or quit. In these situations, we use a `while` loop. The general form of a `while` loop is as follows:

```
while expression:
    #Body
```

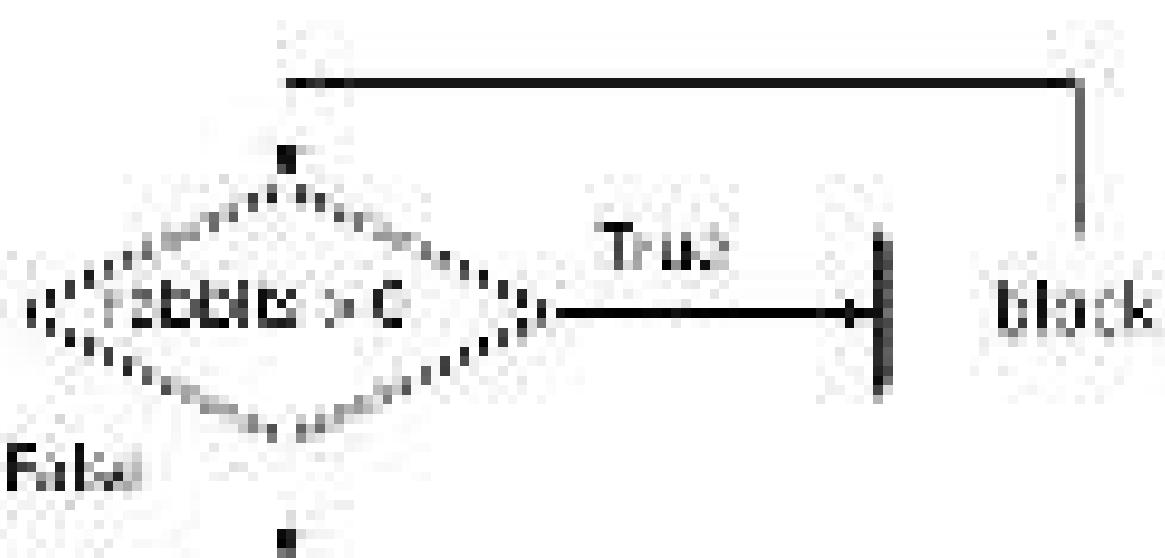
The `while` loop expression is sometimes called the loop condition, just like the condition of an `if` statement. When Python executes a `while` loop, it evaluates the expression. If that expression evaluates to `False`, that is the end of the execution of the loop. If the expression evaluates to `True`, on the other hand, Python executes the loop body once and then goes back to the top of the loop and reevaluates the expression. If it still evaluates to `True`, the loop body is executed again. This is repeated—expression, body—expression, body—until the expression evaluates to `False`, at which point Python stops executing the loop.

Here's an example:

```

for rabbit := 3
    while rabbit > 0:
        print(rabbit)
        rabbit = rabbit - 1
    ...
3
2
1
```

Note that this loop did not print 0. When the number of rabbits reaches zero, the loop condition evaluates to false, so the body isn't executed. Here's a flow chart for this code:



As a more useful example, we can calculate the growth of a bacterial colony using a simple exponential growth model, which is essentially a calculation of compound interest:

$$P(t+1) = P(t) \cdot (1+r)$$

In this formula,  $P(t)$  is the population size at time  $t$ , and  $r$  is the growth rate. Using this program, it's easy to see how long it takes the bacteria to double their population:

```

time = 0
population = 1000 # initial conditions
growth_rate = 0.21 # 21% growth per minute
while population < 2000:
    population = population + growth_rate * population
    print("at time", time, "population is", population)
    time = time + 1
```

`print("at time", time, "population is", population)` gives the answer we're looking for: 2000.

The variable `time` was updated in the loop body. Its value after the loop was the time of the last iteration, which is exactly what we want. Running this program gives us the answer we were looking for:

```

1210
1464
1772
2144
It took 4 iterations for the bacteria to double.
The final population was 2144 bacteria.

```

### Infinite Loops

The preceding example used population < 2000 as a loop condition so that the loop stopped when the population reached double its initial size or more. What would happen if we stopped only when the population was exactly double its initial size?

# use multi-line assignment to set up constants

```
base_population = 2000
growth_rate = 0.02
```

# don't stop until we're exactly double the original value

```
while population != base_population:
```

```
    population = population + growth_rate * population
```

```
    print("population = ", population)
```

```
    time = time + 1
```

```
print("It took", time, "time units for the bacteria to double.")
```

Here's this program's output:

```

1210
1464
1772
2144
...
...5,000 lines or so later...
inf
inf
inf
inf
...
...and so on forever...

```

Whoops—since the population is never exactly two thousand bacteria, the loop never stops! The first several iterations are more than three thousand values, each 21 percent larger than the one before. Eventually, these values are too large for the computer to represent, so it displays “inf” on some computers (Ubuntu), which is to say it’s saying “indefinitely infinity.”

A loop like this one is called an **infinite loop**, because the computer will execute it forever (or until you kill your program, whichever comes first). In IDLE, you kill your program by selecting Restart Shell from the Shell menu. From the command-line shell, you can kill it by pressing Ctrl-C. Infinite loops are a common kind of bug; the usual symptoms include printing the same value over and over again or hanging (doing nothing at all).

Source: https://www.mathsisfun.com/algebra/exponential-equations.html

## 9.7 Repetition Based on User Input

We can use function repetition in a loop to make the chemical formula translation example from Section 5.2, [Choosing Which Statements to Execute](#), on page 146, interactive. We will ask the user to enter a chemical formula, and our program, which is saved in a file named `translate.py`, will print its name. This should continue until the user types `quit`:

```
text = ''
while text != 'quit':
    text = input('Please enter a chemical formula (or "quit" to exit): ')
    if text == 'quit':
        print('Exiting program')
    elif text == 'H2O':
        print('Water')
    elif text == 'CH4':
        print('Methane')
    elif text == 'CO2':
        print('Carbon dioxide')
    else:
        print('Unknown compound')
```

Since the `loop` condition checks the value of `text`, we have to assign it a value before the loop begins. Now we can run the program in `Terminal` and it will continue whenever the user types `quit`:

```
Please enter a chemical formula (or "quit" to exit): H2
Methane
Please enter a chemical formula (or "quit" to exit): CO2
Carbon dioxide
Please enter a chemical formula (or "quit" to exit): quit
Exiting program
```

The number of times that this loop executes will vary depending on user input, but it will execute at least once.

## 9.8 Controlling Loops Using Break and Continue

As a rule, for and while loops evaluate all the statements in their body on each iteration. However, sometimes it is handy to be able to break that rule. Python provides two ways of interrupting the iteration of a loop: `break`, which terminates execution of the loop immediately, and `continue`, which skips ahead to the next iteration.

### The Break Statement

In Section 8.7, [Repetition Based on User Input](#), on page 161, we showed a program that continuously reads input from a user until the user types `quit`. Here

is a program that accomplishes the same task, but this time uses break to terminate execution of the loop when the user types quit:

```
while True:
    text = input("Please enter a word or press Ctrl-D to quit: ")
    if text == "quit":
        print("Exiting program")
        break
    elif text == "hi":
        print("Hello!")
    elif text == "oh":
        print("Wow!")
    elif text == "oh":
        print("Huh?")
    else:
        print("Unknown command")
```

This implementation is stronger. It evaluates to true so this looks like an infinite loop. However, when the user types out the first condition, `text == "quit"`, eval returns to True. This `print()` and `break` statement is executed and prints the last statement, which causes the loop to terminate.

As a side point, we are somewhat allergic to loops that are written like this. We find this style loop with an explicit condition to be harder to understand.

Remember a loop's task is to check before the first iteration. Using what you have seen so far, though, the loop still has to finish iterating. For example, let's write some code to find the index of the first digit in string `PIR`. (The digit 3 is at index 1 in this string). Using a for loop, we would need to write something like this:

```
var s = "PIR"
var digit_index = -1 # True until an integer is found in the string.
for i in range(len(s)):
    ... # If we haven't found a digit, and find one in s[i]
    ... if digit_index == -1 and s[i].isdigit():
        ...     digit_index = i
...
var digit_index
1
```

Here we use variable `digit_index` to represent the index of the first digit in the string. It initially refers to -1, but when a digit is found, the `digit_index` is assigned to `digit_index`. If the string doesn't contain any digits, then `digit_index` remains -1 throughout execution of the loop.

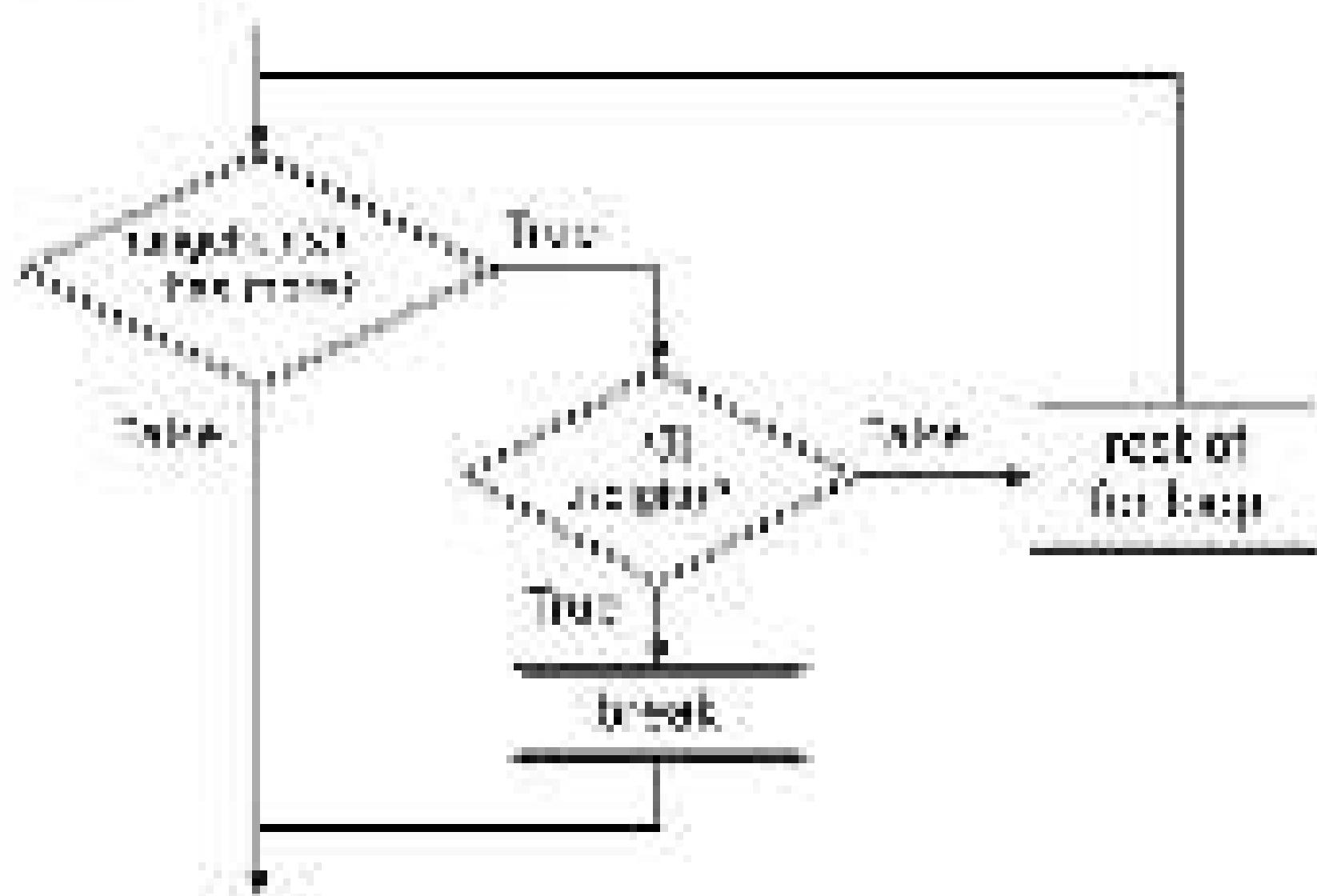
Once `digit_index` has been assigned a value, it is never again equal to -1, so the P condition will not evaluate to True. Even though the job of the loop is done, the loop continues to iterate until the end of the string is reached.

To fix this, you can terminate the loop early using a `break` statement, which jumps out of the loop body immediately:

```
num = '12345'
sum_dig_index = 0 # sum of digits in index 0
for i in range(len(num)):
    ... # of num digit > digit
    if s[i].isdigit():
        ... # digit_index + 1
        break # Total adds the zeros.
    ...
    sum_dig_index
1
```

Note that because the loop terminates early, we don't need to shrink the `if` statement condition. As soon as `digit_index` is assigned a new value, the loop terminates, so it isn't necessary to check whether `digit_index` equals `len`. This check only related to prevent `digit_index` from being assigned to `len` by a subsequent digit in the string.

Let's draw a flowchart for this code:



One more thing about `break`: it terminates only the innermost loop in which it is contained. This means that in a nested loop, a `break` statement inside the inner loop will terminate only the inner loop, not both loops:

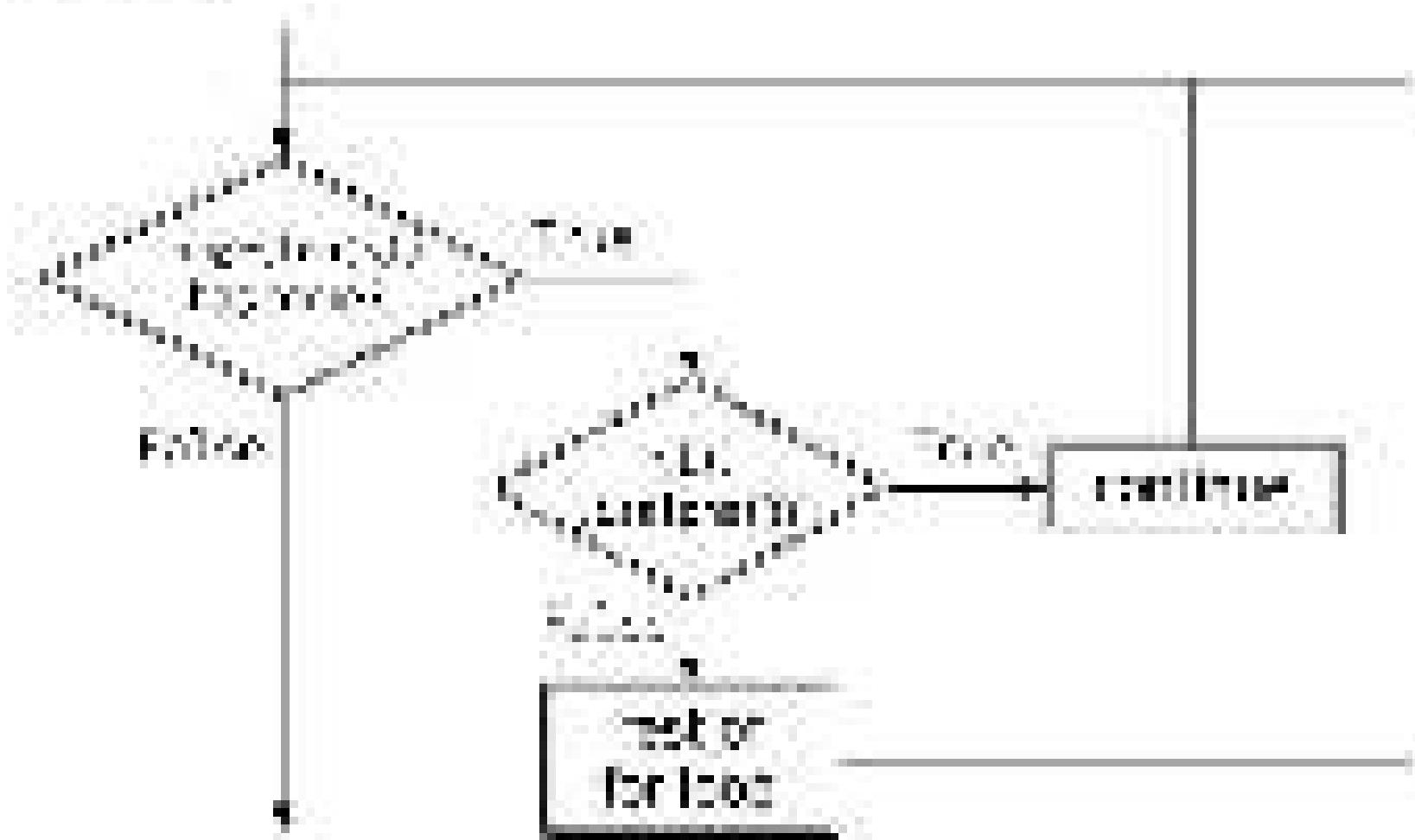
### The `Continue` Statement

Another way to bend the rules for iteration is to use the `continue` statement, which causes Python to skip immediately ahead to the next iteration of a loop. Here, we add up all the digits in a string, and we also count how many

digits there are. Whenever a result is encountered, we use `break` to stop the rest of the loop body and go back to the top of the loop in order to start the next iteration.

```
src> s = 'C9H'
src> total = 0 # The sum of the digits user so far.
src> count = 0 # The number of digits user so far.
src> for i in range(len(s)):
...     if s[i].isalpha():
...         continue
...     total = total + int(s[i])
...     count = count + 1
...
src> total
10
src> count
2
```

When `continue` is executed, it immediately begins the next iteration of the loop. All statements in the loop body that appear after it are skipped, as we only calculate the `total` and `count` when `s[i]` isn't a letter. Here is a flowchart for this code:



Using `continue` is one way to skip a alphabetic characters, but this can also be accomplished by using `if` statements. In the previous code, we can prevent the variables from being modified. In other words, if the character isn't alphabetic, it should be processed.

The form of the previous semantics matches that of an `if` statement, and the updated code is as follows:

```
src> s = 'C9H'
src> total = 0
```

```

>>> count = 0
>>> for i in range(len(s)):
...     if not s[i].isalpha():
...         total = total + int(s[i])
...         count = count + 1
...
>>> total
10
>>> count
2

```

This new version is easier to read than the first one. Most of the time, it is better to rewrite the code to avoid `continue`, since always, the code ends up being more readable.

### A Warning About Break and Continue

`break` and `continue` have their place, but they should be used sparingly since they can make programs harder to understand. When people see while and for loops in programs, their *new assumption* is that the whole body will be executed every time—their mind’s best guess is that the body can be treated as a single “super statement” when trying to understand the program. If the loop contains `break` or `continue`, though, that assumption is false. Sometimes only part of the statements in the body will be executed, which means the reader has to keep two minds in mind:

There are always alternatives well-chosen loop conditions [see [Section 9.7, Repetition Based on User Input \(or just: ISL\)](#)] can replace `break`, and `else` clauses can be used to skip statements instead of `continue`. It is up to the programmer to decide which option makes the program clearer and which makes it more complicated. As we said in [Section 9.7, Repetition Cycles](#) on page 113, programs are written for human beings. Taking a few moments to make your code as clear as possible, or to make clarity a habit, will pay dividends for the lifetime of the program.

Note: But code is getting pretty complicated. It's even more important to write comments describing the purpose of each tricky block of statements.

## 9.9 Repeating What You've Learned

In this chapter, you learned the following:

- Repeating a block is a fundamental way to control a program’s behavior. A for loop can be used to iterate over the items of a list, over the characters of a string, and over expressions of integers, floats, or built-in function names.

- The most general kind of repetition is the `while` loop, which continues executing its body as long as some specified Boolean condition is true. However, the condition is tested only at the beginning of each iteration. If that condition is never false, the loop will be executed forever.
- The `break` and `continue` statements can be used to change the way loops execute.
- Control structures like loops and conditionals can be nested inside one another to any desired depth.

## 9.10 Exercises

Here are some exercises for you to try on your own. Solutions are available at [http://openpyr.com/python\\_exercises\\_solutions.html](http://openpyr.com/python_exercises_solutions.html).

- Write a `for` loop to print all the values in the `ideاصن_پروپریتیز` list from Section 8.4, Sorting Lists, on page 137 (one per line; `category_group_type` refers to 27th, 1st, 3rd, 1st, 7th, 5th).
- Write a `for` loop to print all the values in the `half_beat` list from Section 8.8, Operations on Lists, on page 134, all on a single line. (`half_beat` refers to [37.74, 24.517, 65.526, 12.3, 37.996]).
- Write a `for` loop to add 1 to all the values from `water` from Section 8.1, Sharing and Accessing Data in Lists, on page 129, and store the converted values in a new list called `new_water`. The code below shows an example that adds 1, first to [3, 4, 7, 3, 2, 5, 3, 6, 4, 2, 3, 1, 2, 4].
- In this exercise, you'll create a nested list and then write code that performs operations on that list.
  - Create a nested `list` where each element of the outer list contains the atomic number and atomic weight for an alkali or earth metal. The values are: lithium (3 and 9.012), magnesium (12 and 24.302), calcium (20 and 40.078), strontium (38 and 87.62), barium (56 and 137.3187), and radium (88 and 226). Assign the list to variable `alkali_and_earth`.
  - Write a `for` loop to print all the values in `alkali_and_earth`, with the atomic number and atomic weight for each alkali or earth metal on a different line.
  - Write a `for` loop to create a new list called `numbers_and_weight` that contains the elements of `alkali_and_earth` in the same order but just two lines.

- Ex. The following function does it have a docstring or comments? Write enough code to make it easy for another programmer to understand what the function does and how, and then compare your solution with those of at least two other people. How similar are they? Why do they differ?

```
def mystery_function(values):
    result = []
    for i in range(len(values)):
        result.append(mystery(i))
    return result
```

- Ex. In Section 8.7, *Repetition: Break and Continue*, on page 181, you saw a loop that prompted users until they typed `exit`. This code won't work if users type `quit`, or `Q`, or any other version that isn't exactly `quit`. Modify that loop so that it terminates if a user types that word with any capitalization.
- Ex. Consider the following statement, which creates a list of populations of countries in eastern Asia: `Korea, DPR Korea, Hong Kong, Mongolia, Republic of Korea, and Taiwan` to millions: `country_populations = [1295, 23, 7, 1, 47, 51]`. Write a `for` loop that adds up all the values and stores them in variable `total`. Hint: Give `total` an initial value of zero, and inside the loop body, add the population of the current country to `total`.
- Ex. You are given two lists, `rat1` and `rat2`, that contain the daily weights of two rats over a period of ten days. Assume the rats never have exactly the same weight. Write statements to do the following:
- If the weight of `rat1` is greater than that of `rat2` on day 1, print "Rat 1 weighed more than Rat 2 on day 1". Otherwise, print "Rat 1 weighed less than Rat 2 on day 1".
  - If `rat1` weighed more than `rat2` on day 1 and if `rat1` weighs more than `rat2` on the last day, print "Rat 1 weighed more than Rat 2". Otherwise, print "Rat 2 weighed heavier than Rat 1".
  - If your solution to the previous exercise used nested `if` statements, then do it without nesting, or vice versa.
- Ex. Print the numbers in the range 38 to 49 (inclusive).
- Ex. Print the numbers from 1 to 10 (inclusive) in descending order, no new line.

11. Using a loop, sum the numbers in the range 2 to 22 (inclusive), and then calculate the average.

12. Consider this code:

```
def remove_neg(num_list):
    """ (List of numbers) --> ListType

    Removes the negative numbers from the list num_list.

    Examples:
    remove_neg([1, 5, -2, -3, 2]) == [1, 5, 2]
    remove_neg([]) == []
    remove_neg([-1, -2, -3]) == []
    remove_neg([1, 2, 3]) == [1, 2, 3]
```

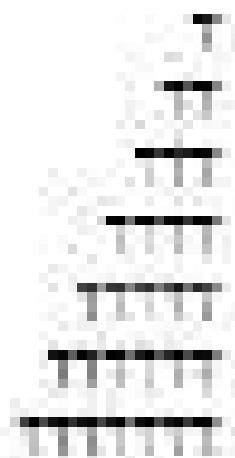
for item in num\_list:  
 if item < 0:  
 num\_list.remove(item)

What `remove_neg([1, 2, 5, -1, 6, -1, -1, 1])` is expected, the function `[1, 2, 5, 6, 1]`. The for loop traverses the elements of the list, and when a negative value (like `-1`) at position `i` is reached, it is removed, shifting the subsequent values one position earlier in the list (so it moves into position `N`). The loop just continues on to process the next item, skipping over the value that moved into the removed item's position. If there are two negative numbers in a row (like `-1` and `-1`), then the second one won't be removed. Rewrite this code to avoid this behavior.

13. Using nested for loops, print a right triangle of the character `T` to the screen where the triangle is one character wide at its narrowest point and seven characters wide at its widest width:

```
T
TT
TTT
TTTT
TTTTT
TTTTTT
TTTTTTT
```

14. Using nested for loops, print the triangle described in the previous exercise with the tripes mixed on the left side:



15. Rewrite the previous two exercises using while loops instead of for loops.
16. Variables `rat1_weight` and `rat2_weight` contain the weights of two rats at the beginning of an experiment. Variables `rat1_grow` and `rat2_grow` assume the rate that the rats' weights are expected to increase each week (for example, 4 percent per week).
- Using a while loop, calculate how many weeks it would take for the weight of the first rat to become 25 percent heavier than it was originally.
  - Assume that the two rats have the same initial weight, but rat 1 is expected to gain weight at a faster rate than rat 2. Using a while loop, calculate how many weeks it would take for rat 1 to be 10 percent heavier than rat 2.

# Reading and Writing Files

Data is often stored in plain text files, which can be organized in several different ways. The most straightforward data organization is one piece of data per line; for example, the rainfall amount in Oregon for each separate day in a study period might be stored on a line of its own. Alternatively, each line might store the values for an entire week or month, with a delimiter such as a comma, tab, or colon used to separate values to make the data easier for humans to read.

Often, data organization is more complex. For example, a study might keep track of the heights, weights, and ages of the participants. Each person would appear on a line by itself with the pieces of data in each record separated by commas. Some records might even span multiple lines, in which case each record will usually have some kind of a separator (such as a blank line) or use special symbols to mark the start or end of each record.

In this chapter, you'll learn about different file formats, common ways to organize data, and how to read and write this data using Python. You'll first learn how to open and read information from files. After that, you'll learn about the different techniques for reading files, and then you'll see several code examples that use the various techniques.

## 10.1 What Kinds of Files Are There?

There are many kinds of files. Text files, movie files, videos, and various word processing and presentation documents are common. Text files only contain characters; all the other file formats include formalized information that is specific to that particular file format, and in order to use a file in a particular format you need a special program that understands that format.

Try opening a Microsoft PowerPoint (.pptx) file in a text editor such as Apple TextEdit, Microsoft Notepad, or one of the many Linux text editors such as `vi`,

marks, and so on. Scroll through it, and you'll see what looks like gibberish text. That is because those files contain a lot of information: what's a title, what are the headings, which words are bold, which are italic, what the line height should be, what the margins are, what the links to embedded images are, and a lot more. Without a program such as Microsoft PowerPoint, .ppt files are unusable.

Text files, on the other hand, don't contain any style information. They contain only readable characters. You can open a text file in any text editor and read it. You won't include style information in text files, but you gain a lot in portability.

Plain-text files take up very little disk space. Compare the size of an empty text file to “empty” OpenOffice, Apple Pages, and Microsoft Word documents:

|                                                                                                                 |              |                                   |
|-----------------------------------------------------------------------------------------------------------------|--------------|-----------------------------------|
|  <a href="#">empty.docx</a>  | 21 kB        | Microsoft Word document           |
|  <a href="#">empty.odt</a>   | 5 kB         | OpenDocument Text (text) Document |
|  <a href="#">empty.pages</a> | 25 kB        | Pages document                    |
|  <a href="#">empty.txt</a>   | 0 kB (bytes) | Plain Text File                   |

The empty.txt file is truly empty; there is no styling information or metadata such as author information, number of pages, or anything else in the file. This makes text files much faster to process than other kinds of documents, and any existing program can read an empty text file.

They take up little disk space and are easy to process. The power comes from applications that can process text files that are written with a particular syntax. The Python programs we have been writing are text files, and by convention they are only written in a file that contains it with a Python interpreter, these Python text files are called. You can execute a powerful algorithm, and the interpreter will follow your instructions.

Similarly, web browsers read and process HTML files, spreadsheets read and process comma-separated value files, calendar programs read and process calendar data files, and other programming language applications read and process files written with a particular programming language's syntax. A database, which you'll learn about in [Chapter 17: Databases](#), on page 802, is another way to store and manage data.

In the next section, you'll learn how to write programs that open and print the contents of a text file.

## 10.2 Opening a File

When you want to write a program that opens and reads a file, that program needs to tell Python where that file is. By default, Python assumes that the file you want to read is in the same directory as the program that is doing the reading. If you’re working in IDLE, as you read this book, there’s a little action you should do:

1. Make a directory, perhaps called `file_exercises`.
2. In IDLE, select `File→New Window` and type (or copy and paste) the following:

```
# first line of text  
second line of text  
third line of text
```

3. Save this file in your `file_exercises` directory under the name `file_data.txt`.
4. In IDLE, select `File→New Window` and type (or copy and paste) this program:

```
file = open('file_data.txt', 'r')  
contents = file.read()  
print(contents)  
file.close()
```

5. Save this as `file_reader.py` in your `file_exercises` directory.

When you run this program, this is what gets printed:

```
First line of text  
Second line of text  
Third line of text
```

It’s important that we save the two files in the same directory, as you’ll see in the next section. Also, this won’t work if you try these same commands from the Python shell.

Built-in function `open` opens a file much like you open a book when you want to read it and return an *object* that knows how to get information from the file: how much you’ve read, and which part of the file you’re about to read next. The marker that keeps track of the current location in the file is called a *file cursor* and acts much like a bookmark. The file cursor is initially at the beginning of the file, but as you read or write data it moves to the end of what you just read or wrote.

The `file` argument in the example call on function `open`, “`file_data.txt`”, is the name of the file to open and the `second` argument, `'r'`, tells Python the file

want to read the file this is called the file mode. Other options for this mode include `w` for writing and `a` for appending, which you'll see later in this chapter. If you call `read` with only the name of the file (omitting the mode), then the default is `r`.

The second statement, `contents = file.read`, tells Python that you want to read the contents of the entire file into a string, which we assign to a variable called `contents`. The third statement, `print(contents)`. When you run this program, you'll see that newline characters are treated just like every other character: a newline character is just another character in the file.

The last statement, `file.close()`, releases all resources associated with the open file object.

### The `with` Statement

Because every call on function `open` should have a corresponding call on method `close`, Python provides a `with` statement that automatically closes a file when the end of the block is reached. Here is the same example using a `with` statement:

```
with open('text_file.txt') as file:
    contents = file.read()
```

`print(contents)`

The general form of a `with` statement is as follows:

```
with open('filename', 'mode') as Variable:
    block
```

### How Files Are Organized on Your Computer

A file path specifies a location in your computer's file system. A file path contains the sequence of directories to a file, starting at the root directory at the top of the file system, and optimally including the name of a file.

Here is an example of the file path for file `script.py`:

`Users/jacob/Desktop/script.py`

This file path is on a computer running Apple OS X. A file path to Linux would look similar. Path separating systems use a forward slash as the directory separator.

In Microsoft Windows, the path usually begins with a drive letter, such as C. There is one colon letter per disk partition. Also, Microsoft Windows uses a backslash as the directory separator. When working with backslashes in

different separators, you might want to review [Section 4.2, “More Advanced Characters and Strings”](#) on page 122.)

How to a path in Windows:

 \

If you always use \ between slashes, Python's file-handling operations will automatically translate them to backslashes in Windows.

## Specifying Which File You Want

Python keeps track of the current working directory. This is the directory in which it looks for files. When you run a Python program, the current working directory is the directory where that program is saved. For example, perhaps this is the path of the file that you have open in IDLE:

 C:\Users\David\Documents\pyBook\chapter01\openfile.py

Then this is the current working directory:

 C:\Users\David\Documents\pyBook\chapter01\openfile.py

When you call function `open`, it looks for the specified file in the current working directory.

The default current working directory for the Python shell is operating-system dependent. You can find out the current working directory using function `getcwd` from module `os`:

```
os.getcwd()
#>>> os.getcwd()
'/home/patrick'
```

If you want to open a file in a different directory, you need to tell where that file is. You can do that with either an absolute path or with a relative path. An absolute path (like all the examples above) is one that starts at the root of the file system, and a relative path is relative to the current working directory. Alternatively, you can change Python's current working directory to a different directory using function `chdir` (short for “change directory”):

```
os.chdir('/home/patrick/Documents/pyBook')
#>>> os.getcwd()
'/home/patrick/Documents/pyBook'
```

Let's say that you have a program called `makeup` and a directory called `data` in the same directory as `makeup`. Inside `data` you might have files called `makeup.html` and `makeup.txt`. This is how you would open `data.html`:

```
open('data/makeup.html', 'r')
```

Here, `datafile.txt` is a relative path.<sup>1</sup>

To look to the directory above the current working directory, you can use two dots:

```
open('data/dots.txt', 'r')
```

You can chain them to go up multiple directories. Here, Python looks for `dots` three directories above the current working directory and then down into a `dots` directory:

```
open('../..//data/dots.txt', 'r')
```

## 10.3 Techniques for Reading Files

As we mentioned at the beginning of the chapter, Python provides several techniques for reading files. You'll learn about them in this section.

All of these techniques work starting at the current file cursor. This allows us to consider the techniques in sequence.

### The Read Technique

Use this technique when you want to read the contents of a file into a single string, or when you want to specify exactly how many characters to read. This technique was introduced in [Section 10.1. Opening a File \(cont'd\)](#). Here is the same example:

```
with open('example.txt', 'r') as file:  
    contents = file.read()
```

```
print(contents)
```

When called with no arguments, it reads everything from the current file cursor all the way to the end of the file and moves the file cursor to the end of the file. When called with one integer argument, it reads that many characters and moves the file cursor after the characters that were just read. Here is a version of the same program in a file called `example.py`: It reads ten characters and then the rest of the file:

```
with open('example.txt', 'r') as example_file:  
    first_ten_chars = example_file.read(10)  
    the_rest = example_file.read()
```

```
print(first_ten_chars) # first ten chars
```

```
print(the_rest) # the rest
```

---

<sup>1</sup> Python still makes it hard to handle paths well. Try looking at the [absolute](#) [os.path module](#).

Method `readline()` moves the file cursor to the next cell, so example `lines[1]` reads everything from character 11 to the end of the file.

### Reading at the End of a File

When the file cursor is at the end of the file, functions `read()`, `readline()`, and `readlines()` return empty strings. In order to read the contents of a file forward, then, you'll need to close and reopen the file.

### The `readlines` Technique

Use this technique when you want to get a Python list of strings containing the individual lines from a file. `readlines` works much like function `read()`, except that it splits up the file into a list of strings. As with `read()`, the file cursor is moved to the end of the file.

This example reads the contents of a file into a list of strings and then prints the list:

```
with open('file_example.txt', 'r') as example_file:  
    lines = example_file.readlines()  
  
print(lines)
```

Here is the output:

```
[<str: 'Line of text.\nSecond line of text.\nThird line of text.'>]
```

Take a closer look at that list; you'll see that each line ends in a character. Python does not remove any characters from what is read; it only splits them into separate strings.

The last line of a file may be empty and will contain a newline character, as summarized in Section 1.8, *Reading Multiple Lines*, on page 119.

Assume the planet file contains the following text:

```
Mercury  
Venus  
Earth  
Mars
```

This example prints the lines in reverse backward, from the last line to the first line. We use built-in function `reversed`, which returns the items in the list in reversed order:

```

>>> with open('planets.txt', 'r') as planets_file:
...     planets = planets_file.readlines()
...
>>> planets
['Mercury\n', 'Venus\n', 'Earth\n', 'Mars\n']
>>> for planet in reversed(planets):
...     print(planet.strip())
...
Mars
Earth
Venus
Mercury

```

We can use the `readlines` technique to read the file *line by line*, and print the planets alphabetically (here, we use built-in function `reversed`, which returns the items in the list *in order from smallest to largest*):

```

>>> with open('planets.txt', 'r') as planets_file:
...     planets = planets_file.readlines()
...
>>> planets
['Mercury\n', 'Venus\n', 'Earth\n', 'Mars\n']
>>> for planet in sorted(planets):
...     print(planet.strip())
...
Earth
Mars
Mercury
Venus

```

## The “For Line in File” Technique

Use this technique when you want to do the same thing to every line from the file *except* to the end of a file. On each iteration, the file cursor is moved to the beginning of the next line.

This code opens file `data.txt` and prints the length of each line. In fact, the code reads open file `data.txt` and prints the length of each line. In fact, the code reads open file `data.txt` and prints the length of each line.

```

>>> with open('data.txt', 'r') as data_file:
...     for line in data_file:
...         print(len(line))
...
8
5
5
5

```

Take a close look at the last line of output. There are only four characters in the word “Hura”, but our program is reporting that the line is five characters long. The reason for this is the `len` or `len()` function reads *each* of the lines

wanted from the file leaves nothing written at the end. We can get rid of it using string method `strip()`, which returns a copy of a string that has leading and trailing whitespace characters (spaces, tabs, and newlines stripped away).

```
... with open('planets.txt', 'r') as data_file:
```

```
...     for line in data_file:
...         print(line.strip())
...
7
8
9
10
```

## The Readline Technique

This technique reads one line at a time, unlike the readline() technique. Use this technique when you want to read only part of a file.

For example, you might want to treat lines differently depending on context; perhaps you want to process a file that has a header section followed by a series of records, either one record per line or with multiple records.

The following data, taken from the [Pant-Sarac Data Library](#), [Hurst](#), describes the number of pelts of fox fur produced in Hopedale, Labrador, in the years 1874-1875. (The full data set has values for the years 1874-1875.)

Collected for fur production, HOPEDALE, Labrador, 1874-1875.

**Source:** C. Elton (1942) "Notes, Nits and Laundry", *Edgars Journals*, Press

**Table 17.** p. 265-266

```
22
20
2
16
12
33
0
00
166
```

The first line contains a description of the data. The next two lines contain comments about the data, both of which begin with a \* character. Each piece of actual data appears on a single line.

We'll use the Readline technique to skip the header, and then we'll use the `for line in file` technique to process the data in the file, counting how many fox fur pelts were produced.

```

with open('hopedale.txt', 'r') as hopedale_file:

    # Read the description line.
    hopedale_file.readline()

    # Keep reading content lines until we read the first piece of data.
    data = hopedale_file.readline().strip()
    while data.startswith('#'):
        data = hopedale_file.readline().strip()

    # Now we have the first piece of data. Associate the total number of
    # polts.
    total_pelts = int(data)

    # Read the rest of the file.
    for data in hopedale_file:
        total_pelts = total_pelts + int(data.strip())

print(total_pelts)

```

And here is the output:

```
Total number of pelts: 372
```

Each call on `functools.readline` moves the file cursor to the beginning of the next line.

Sometimes trailing whitespace is important and you'll want to preserve it. In the Hopkirk data, for example, the integers are right justified to make them line up nicely. In order to preserve this, you can use `strip` instead of `rstrip` to remove the trailing newline; here's a program that prints the data from that file, preserving the whitespace:

```

with open('hopedale.txt', 'r') as hopedale_file:

    # Read the description line.
    hopedale_file.readline()

    # Keep reading content lines until we read the first piece of data.
    data = hopedale_file.readline().strip()
    while data.startswith('#'):
        data = hopedale_file.readline().strip()

    # Now we have the first piece of data.
    print(data)

    # Read the rest of the data.
    for data in hopedale_file:
        print(data.strip())

```

And here is the output:

```
22
29
2
16
12
35
8
33
165
```

## 10.4 Files over the Internet

These days, of course, the file containing the data we want could be on a machine half a world away. Provided the file is accessible over the Internet, though, we can read it just as we do a local file. For example, the DopeData data not only exists in our computer, but it's also on a web page. As the code is writing, the URL for the file is <http://www.pythontutorial.net/tutorials/dope-data/> (you can look at it online).

Note that the examples in this section will work only if your computer is actually connected to the Internet.

Module `urllib.request` contains a function called `urlopen` that opens a web page for reading. `urlopen` returns a file-like object that you can use much as if you were reading a local file.

There's a hitch, however: there are many kinds of files (images, music, videos, text, and more). The file-like objects read with `urlopen` methods both return a type you haven't yet encountered: `bytes`.

### What's a Byte?

Two bytes of information tell us nothing: it lets us know that's about a byte. All data, however, has some specific representation, such as a sequence of bits. Computers represent these bits in groups of eight. Each group of eight bits is called a byte. Programming languages have had this byte as an available unit of data since the very first computers, such as the UNIVAC I, the Bell 101, the IBM 650, vintage, and others, and documents.

When dealing with type `bytes`, such as a piece of information returned by a call to `urlopen` from `urllib.request`, we need to decode it. In order to decode it, we need to know how it was encoded.

Common encoding schemes are described in the online Python documentation: <https://pythonprogramming.net/text-and-binary-data-encoding/>

The Republic data on the Web is crawled using LWP::Simple module that web::page and user-agent methods from in order to decode the text obtained and take repart.

```
#!/usr/bin/python
# crawl_republic.py
# with replit the script is imported to an webpage
for line in webpage:
    line = line.strip()
    line = line.decode('utf-8')
    print(line)
```

## 10.5 Writing Files

This program opens a file called topics.txt, writes the words Computer Science to the file, and then closes the file.

```
with open('topics.txt', 'w') as output_file:
    output_file.write('Computer Science')
```

In addition to writing characters to a file, `write` returns the number of characters written. For example, `len('Computer Science')` returns 16.

To create a new file or to replace the contents of an existing file, we use `write` mode (`'w'`). If the file name doesn't exist already, then a new file is created; otherwise the file contents are erased and replaced. This is useful for writing, you can use method `file.readline` to write a string to the file.

Another alternative replacing the file contents, you can also add to a file using the append mode (`'a'`). When we write to a file that is opened in append mode, the data we write is added to the end of the file and the current file contents are not overwritten. For example, In add to our previous file `topics.txt`, we can append the word `Software Engineering`.

```
with open('topics.txt', 'a') as output_file:
    output_file.write('Software Engineering')
```

At this point, if we print the contents of `topics.txt`, we'll see the following:

`Computer Science Software Engineering`

Unlike `print` function with `file.close()` automatically start a new line. If you want a string to end in a newline, you have to include it manually using `\n`. In each of the previous examples, we called `write` only once, but you'll typically call it multiple times.

The next example, a file called `total.py`, is more complex, and it involves both reading from and writing to a file. Our input file contains ten numbers per line separated by a space. The output file will contain those numbers that we sum from the input file and their sum (all separated by a space):

```
#!/usr/bin/python
# total.py
# calculate the sum of all the values in a file
```

```

def sum_number_pairs(input_file, output_file):
    """Read the data from input file, add two floats per line
    separated by a space. Open output file and write, for each line in
    input file, a line to the output file that contains the two floats
    from the corresponding line of input file plus a space and the sum of the
    two floats.

    :param str input_file: name of input file
    :param str output_file: name of output file
    """
    with open(input_file, 'r') as input_file:
        for number_pair in input_file:
            number_pair = number_pair.strip()
            operator = number_pair.split()[1]
            total = float(number_pair[0]) + float(number_pair[2])
            new_line = f'{operator} {total}\n'
            output_file.write(new_line)

```

Assume that `number_pairs.txt` contains the following contents:

```

1.2 2.4
2 -1.2
-1.1

```

Then `sum_number_pairs(number_pairs.txt, 'output.txt')` creates this file:

```

1.2 3.4 4.7
2 -2 0.2
-1 1 0.0

```

## 10.6 Writing Algorithms That Use the File-Reading Techniques

There are several common ways to organize information in files. The rest of this chapter will show how to apply the various file-reading techniques to those situations and how to develop your algorithms to help with this.

### Skipping the Header

Many data files begin with a header, as described in [The Readline Techniques](#) on page 178. TSV files begin with a row for the description followed by comments in lines beginning with a `#`, and the `readline` technique can be useful to skip the header. The technique only works if you read the first real row of data, which will be the row immediately after the description that doesn't start with a `#`.

In English, we might try this algorithm to process the kind of file:

**Skip the first line in the file**

**Skip over the comment lines in the file**

**For each of the remaining lines in the file**

**Process the data on that line**

The problem with this approach is that we can't tell whether a line is a comment line until we've read it, but we can read a line from a file only once – there's no simple way to "pick up" in the file. An alternative approach is to read the file, skip it if it's a comment, and process it if it's not. Once we've processed the first line of data, we process the remaining lines.

```
Skip the first line in the file
Read and process the first line of non-comment data
For each of the remaining lines
    Process the data on the line
```

The thing to notice about this algorithm is that it processes lines in two passes: one which it reads the first "interesting" line in the file and one which it handles all of the following lines.

```
def skip_header(f):
    """File open for reading"""
    # skip the header in reader and return the first real piece of data
    line = reader.readline()
    while line.startswith('#'):
        line = reader.readline()

    # now line contains the first real piece of data
    return line

def process_file(input):
    """File open for reading"""
    # read and print the first piece of data
    line = skip_header(reader)
    print(line)

    # read the rest of the data
    for line in reader:
        line = line.rstrip()
        print(line)

    if reader == None:
        with open('output.txt', 'w') as output_file:
            process_file(input, output_file)
```

In this program, we return the first line of input data. Because this is wider than it, we can't read it again (we can do forward but not backward). We'll want to use `skip_header` in all of the file-processing functions in this section. Rather than copying the code each time we want to use it, we can put the function in a file called `utils.py` (in `Tim's Slick Data Library`) and use it in other programs using `import`, as shown in the next example. This allows us to move the `skip_header` out, and it means it is modified from there to only one copy of the function to edit.

This program processes the `hopperfile` data set to find the smallest number of hives per year in any year. As we progress through the file, we keep the smallest value seen so far in a variable called `smallest`. That variable is initially set to the value on the first line. Whenever the smaller (and only) value seen so far:

```
import time_per_line

def smallest_value_reader():
    """A file open for reading, >= 3 lines.
    Read and process reader and return the smallest value after the
    first two lines.
    """
    line = time_per_line.skip_header(reader, 2)

    # The first two lines are the header, which we don't want to read.
    # Just read the rest of it as though it is the input.
    smallest = int(line)

    for line in reader:
        value = int(line.strip())

        # If we find a smaller value, replace it.
        if value < smallest:
            smallest = value

    return smallest

if __name__ == '__main__':
    with open('hopperfile.txt', 'r') as input_file:
        print(smallest_value(input_file))
```

As with any algorithm, there are other ways to write this. For example, we can replace the `for` statement with this single line:

```
smallest = min(smallest, value)
```

## Dealing with missing values in data

We also have data for colored fox production in Nipigon, Labrador:

Coloured fox fur production, Nipigon, Labrador, 1824-1838

**Source:** C. Elton (1942) "Notes, Mice and Landrige", Oxford Univ. Press

**#Table 17.** p.203-204

**Remark:** missing value for 1836

156

262

.

102

170

227

The hyphen indicates that data for the year 1836 is missing. Unfortunately, calling `read_csv()` on the Nipigon does generates this error:

```
200> import pandas as pd
201> pd.read_csv('labrador_value.csv') #open('labrador.txt', 'r'))
Traceback (most recent call last):
  File <ipython-input-1>, line 1, in <module>
    File <ipython-input-1>, line 1, in read_csv
      value = int(line.strip())
ValueError: invalid literal for int() with base 10: '-'
```

The problem is that `int()` expects an integer or calling `int('')`. This isn't an isolated problem. In general, we will often need to skip blank lines, documents, or lines containing other "impurities" in our data. And this is a common situation in science and engineering, dealing with them is just a fact of scientific life.

In this section, we'll add a check inside the `skip_header()` function to make sure it contains a real value. In the TBLH data sets, missing entries are always marked with hyphens, so we just need to check for that before trying to convert the string we have read to an integer:

```
import time as tm
```

```
def read_csv(filename, skip_header=0):
    """(file open for reading) --> DataFrame
    Read and process reader, under and start with a skip_header header.
    Return the unfiltered value after the header. Skip missing values, which
    are indicated with a hyphen.
    """

```

```
    line = time.perf_counter()
    if line == 0:
        print("start reading file")
    else:
        print("start reading file", line)
    reader = csv.reader(open(filename))
    for i in range(skip_header):
        next(reader)
    for row in reader:
        if row[0] != '-':
            yield row
    print("end reading file", time.perf_counter() - line)
```

```

for line in reader:
    line = line.rstrip()
    if line[0] == '#':
        value = int(line)
        maxYear = min(maxYear, value)

return maxYear

if __name__ == '__main__':
    with open('lynx.txt', 'r') as input_file:
        print(maxYear_value(input_file))

```

Notice that the update to `maxYear` is nested inside the check for hyphens.

## Processing Whitespace-Delimited Data

The file at <http://www.oceanographer.org/lynx.txt> ([The Lynx Data Library](#), 2004) contains information about lynx counts in the years 1721–1934. All data values are integers, but since there are many values, the values are separated by whitespace, and for reasons best known to the file's author, each value ends with a period.

```

annual number of lynx trapped, Mackenzie River, 1721-1934
original source: Elton, S. and MacLean, H. (1942)
# The ten year cycle in numbers of Canadian lynx*.
# J. Animal Ecology, Vol. 11, 215-244
#This is the famous data set which has been 'data' before in
#various publications:
#Carrick, H.J. and Walker, A.H. (1977) "A survey of statistical work on
#the Mackenzie River series of annual Canadian lynx trappings for the years
#1721-1934 with a new analysis" J. Roy. Statistical Soc. Ser. C 14(3), 420-431.
  209. 321. 563. 871. 1473. 2621. 3508. 5042. 4090. 2877. 929. 35.
  154. 279. 460. 2288. 2883. 3485. 3824. 422. 151. 45. 23. 213.
  345. 1803. 2120. 2538. 557. 381. 377. 125. 880. 282. 1222. 1713.
  2871. 2119. 824. 299. 226. 248. 582. 3828. 5532. 3322. 4245. 357.
  256. 473. 398. 684. 1584. 1818. 2250. 3428. 358. 255. 222. 223.
  444. 706. 2602. 3671. 4493. 2611. 509. 15. 25. 45. 54. 243.
  507. 1190. 2601. 3646. 581. 184. 145. 25. 158. 1421. 4485. 5447.
  1019. 2192. 3100. 3601. 502. 1032. 1498. 2114. 4550. 4391. 2685. 4743.
  164. 81. 104. 108. 228. 386. 1162. 2442. 4574. 2943. 547. 324.
  451. 867. 1869. 3166. 7066

```

To process this, we will break each line into pieces and strip off the periods. Our algorithm is the same as it was for the fox pellet data: find and process the first line of data in the file, and then process each of the subsequent lines. However, the notion of “processing a line” needs to be extended further because there are many values per line. Our refined algorithm, shown in `x1`, uses nested loops to handle the notion of “for each line and for each value on that line”:

**Find the first line containing real data after the header.**  
 For each place of data in the current line:  
   Process that place.

**For each of the remaining lines of data,**  
   **For each place of data in the current line,**  
     **Process that place.**

Once again we are processing lines in two different places. That is a situation that we should write a helper function to avoid duplicates code. Refactoring our algorithm and making it specific to the problem of finding the largest value makes this cleaner:

**Find the last line of real data above the header.**  
**Find the target value in that line.**

**For each of the remaining lines of data,**  
**Find the target value in that line.**  
 If that value is larger than the previous target, remember it.

The helper function required is one that takes the largest value in a line, and it must split up the line. String method `split()` will split around the whitespace, but we still have to remove the periods at the ends of the values.

We can also simplify our code by initializing `target = -1`, because that value is guaranteed to be smaller than any of the possible values in the file. That way, no matter what the first real value is, it will be larger than the "previous" value (`-1`) and replace it.

Import this earlier:

```
def find_largest(lines):
    """(str) --> int

    Return the largest value in lines, which is a whitespace-delimited string
    of integers that start and end with a '-'.

    Examples:
    find_largest("-1 2 3 4 5")
    # The largest positive number
    target = 1
    for value in lines.split():
        # Remove the leading minus
        v = int(value[1:])
        # If we found a very small number...
        if v > target:
            target = v

    return target
```

We now face the same choice as with our books. We can put the largest in a module (possibly so), or we can include it in the same file as the rest of the code. We choose the latter this time because the code is specific to this particular dataset and problem:

```
import time_series

def find_largest_line():
    """str -> str

    Return the largest value in line, which is a whitespace-delimited string
    of integers (not separated with a ',').

    ex: find_largest('1 2 3 5 3 11')
    """
    # Given: largest value seen so far
    largest = -1
    for value in line.split():
        # Remove the trailing period.
        v = int(value[:-1])
        # If we find a larger value, remember it.
        if v > largest:
            largest = v

    return largest

def process_file(header):
    """(file open for reading) -> str

    Read and process header, then read data with a time_series header.
    Return the largest value after the reader. There may be multiple pieces
    of data on each line.

    line = raw_input(header).strip()
    # The largest value so far is the largest in this piece of data.
    largest = find_largest(line)

    # Given: the rest of the lines for larger values.
    for line in reader:
        large = find_largest(line)
        if large > largest:
            largest = large

    return largest

if __name__ == '__main__':
    with open('lines.txt', 'r') as input_file:
        print(process_file(input_file))
```

Not to have sample the task, he process the looked! This happened only because we decided to write helper functions. To show you how much clearer this is, here is the same code without using `fd1_header` and `fd1_parser` as helper modules.

```
def process_file(reader):
    """file open for reading) --> int

    Read and process reader, which must start with a class_header.
    Return the largest value after the reader. There may be multiple pieces
    of data on each line.

    # Read the header line
    line = reader.readline()

    # read the first non-empty line
    line = reader.readline()
    while line.strip() == '':
        line = reader.readline()

    # the line contains the first real piece of data
    # The largest value seen so far is the current max
    largest = -1

    for value in line.split():

        # Remove the trailing period
        v = int(value[:-1])

        # If we find a larger value, remember it
        if v > largest:
            largest = v

    # Given line will not have the last period for larger values
    for line in reader:

        # The largest value seen so far is the current max
        largest_in_line = -1

        for value in line.split():

            # Remove the trailing period
            v = int(value[:-1])

            # If we find a larger value, remember it
            if v > largest_in_line:
                largest_in_line = v

    return largest
```

```

if target_in_file == target:
    target = target_in_file

    print("Target")
    if name == "molecule":
        with open("lyne.txt", "r") as input_file:
            print(next(input_file))

```

## 10.7 Multiline Records

Not every data record will fit onto a single line. Here is a file in simplified Protein Data Bank (PDB) format that describes the arrangement of atoms in ammonia:

|               |                |   |       |        |        |
|---------------|----------------|---|-------|--------|--------|
| <b>COMPND</b> | <b>AMMONIA</b> |   |       |        |        |
| ATOM          | 1              | H | 0.257 | -0.363 | 0.000  |
| ATOM          | 2              | H | 0.257 | 0.727  | 0.000  |
| ATOM          | 3              | H | 0.771 | -0.727 | 0.000  |
| ATOM          | 4              | H | 0.771 | -0.727 | -0.000 |
| <b>COD</b>    |                |   |       |        |        |

The first line is the name of the molecule. All subsequent lines down to the one containing COD specify the ID-type, and X/Y/Z coordinates of each of the atoms in the molecule.

Reading this file is straightforward using the techniques we have built up in this chapter. But what if the file contained two or more molecules: ammonia, methane:

|               |                |   |        |        |        |
|---------------|----------------|---|--------|--------|--------|
| <b>COMPND</b> | <b>AMMONIA</b> |   |        |        |        |
| ATOM          | 1              | H | 0.257  | -0.363 | 0.000  |
| ATOM          | 2              | H | 0.257  | 0.727  | 0.000  |
| ATOM          | 3              | H | 0.771  | -0.727 | 0.000  |
| ATOM          | 4              | H | 0.771  | -0.727 | -0.000 |
| <b>COD</b>    |                |   |        |        |        |
| <b>COMPND</b> | <b>METHANE</b> |   |        |        |        |
| ATOM          | 1              | C | -0.746 | -0.015 | 0.034  |
| ATOM          | 2              | H | 0.586  | 0.426  | -0.228 |
| ATOM          | 3              | H | -1.263 | -0.262 | -0.581 |
| ATOM          | 4              | H | -1.263 | 0.754  | 0.888  |
| ATOM          | 5              | H | 0.896  | 0.594  | 0.888  |
| ATOM          | 6              | H | 0.716  | -1.484 | 0.157  |
| <b>COD</b>    |                |   |        |        |        |

As always we tackle this problem by reading two smaller files and joining each of those together. Our first algorithm is as follows:

While there are more molecules in the file

Read a molecule from the file

Append it to the list of molecules read so far

Simply, extract the only way to tell whether there is another molecule left in the file is to try to read it. Our modified algorithm is as follows:

```
reading = True
while reading:
    try:
        molecule = read_a_molecule_from_file('file')
        if there is one:
            append it to the list of molecules
            read next line
            reading = False
        else:
            reading = False
```

In Python, this is as follows:

```
def read_a_molecule_from_file():
    """file open for reading → read as SMILES

    Read a single molecule from reader and return it, or return None if signal
    end-of-file. The first atom in the molecule is the name of the compound,
    each line contains an atom type and the X, Y, and Z coordinates of that
    atom.

    Returns:
        a list of the molecule's atoms, None if end-of-file
    """
    line = reader.readline()
    if not line:
        return None
    else:
        # name of the molecule, remove '\n'
        key, name = line.split()
        # other lines are either 'XYZ' or 'H' for 2D
        molecule = [name]
        line = reader.readline()

    # Parse all the atoms in the molecule.
    while not line.startswith('END'):
        key, num, atom_type, x, y, z = line.split()
        molecule.append(atom_type, x, y, z)
        line = reader.readline()

    return molecule

def read_all_molecules_from_file():
    """file open for reading → read as SMILES

    Read many molecules from reader, returning a list of all molecule
    information.

    Returns:
        a list of all molecules returned
    """
    result = []
    while reading:
        molecule = read_a_molecule_from_file()
        if molecule:
            result.append(molecule)
```

```

reading = True
while reading:
    molecule_file = open(molecule_file_name)
    if molecule_file == None:
        print("Error: file %s does not exist." % molecule_file_name)
        break
    reading = False
    return result

if __name__ == '__main__':
    molecule_file = open('molecule.txt', 'r')
    molecule = read_mol(molecule_file.readline())
    print(molecule)

```

The `while` actually reading a single molecule has been put in a function of its own that must return some value (so it can't find another molecule in the file). The function checks the first line it finds to see whether there is actually any data for it. If no, it returns immediately. In fact `all_molecules` that the end of the file has been reached. Otherwise, it pulls the name of the molecule out of the first line and then reads the molecule's atoms one at a time, shown in the `for` loop:

```

def read_molecule(molecule):
    molecule_file = open(molecule_file_name)
    if not molecule_file:
        print("Error: molecule file %s does not exist." % molecule_file_name)
        exit(1)
    else:
        lines = molecule_file.readlines()
        if len(lines) > 1:
            line = lines[0]
            if line[0] == '#':
                print("Warning: first line of molecule file %s is a comment." % molecule_file_name)
            else:
                molecule = read_mol(line)
                if molecule == None:
                    print("Warning: molecule file %s is empty." % molecule_file_name)
                    exit(1)
                else:
                    reading = True
                    for line in lines[1:]:
                        if line[0] == '#':
                            continue
                        else:
                            molecule.append(read_mol(line))
                    reading = False
                    return molecule
    molecule_file.close()

```

If there isn't another line, it will exit the end of the file.

```

    line = molecule.readline()
    if not line:
        return None

```

If none of the molecules' names match `name`,

```

    if name != line.split()[0]:
        continue

```

it either ends with `XYZ` or `CART` (or `ATOMS(x,y,z)`)

```

    molecule = lines[1]
    reading = True

```

While `reading`:

```

    line = molecule.readline()
    if line[0:4] == "ATOMS":
        reading = False
    else:
        key, name, atom_type, x, y, z = line.split()
        molecule.append((atom_type, x, y, z))

```

### return molecule

Notice that this function uses exactly the same trick to spot the EBD: just consider the end of a molecule as the first molecule used to spot the end of the file.

## 10.8 Looking Ahead

Let's add one final complication. Suppose that molecules didn't have EBD markers but instead just a COMPOD line followed by one or more ATOM lines. How would we read multiple molecules from a single file? In that case?

At first glance (Figure 9, A PDB file Without EBD markers), it doesn't seem much different from the problem we just solved: `read_molecule` could extract the molecule's name from the COMPOD line and then read ATOM lines until it finds either an empty string signifying the end of the file or another COMPOD line signifying the start of the next molecule. But once it has read that COMPOD line, there isn't any table for the next call to `read_molecule`, so how can we get the name of the second molecule (and all the ones following it)?

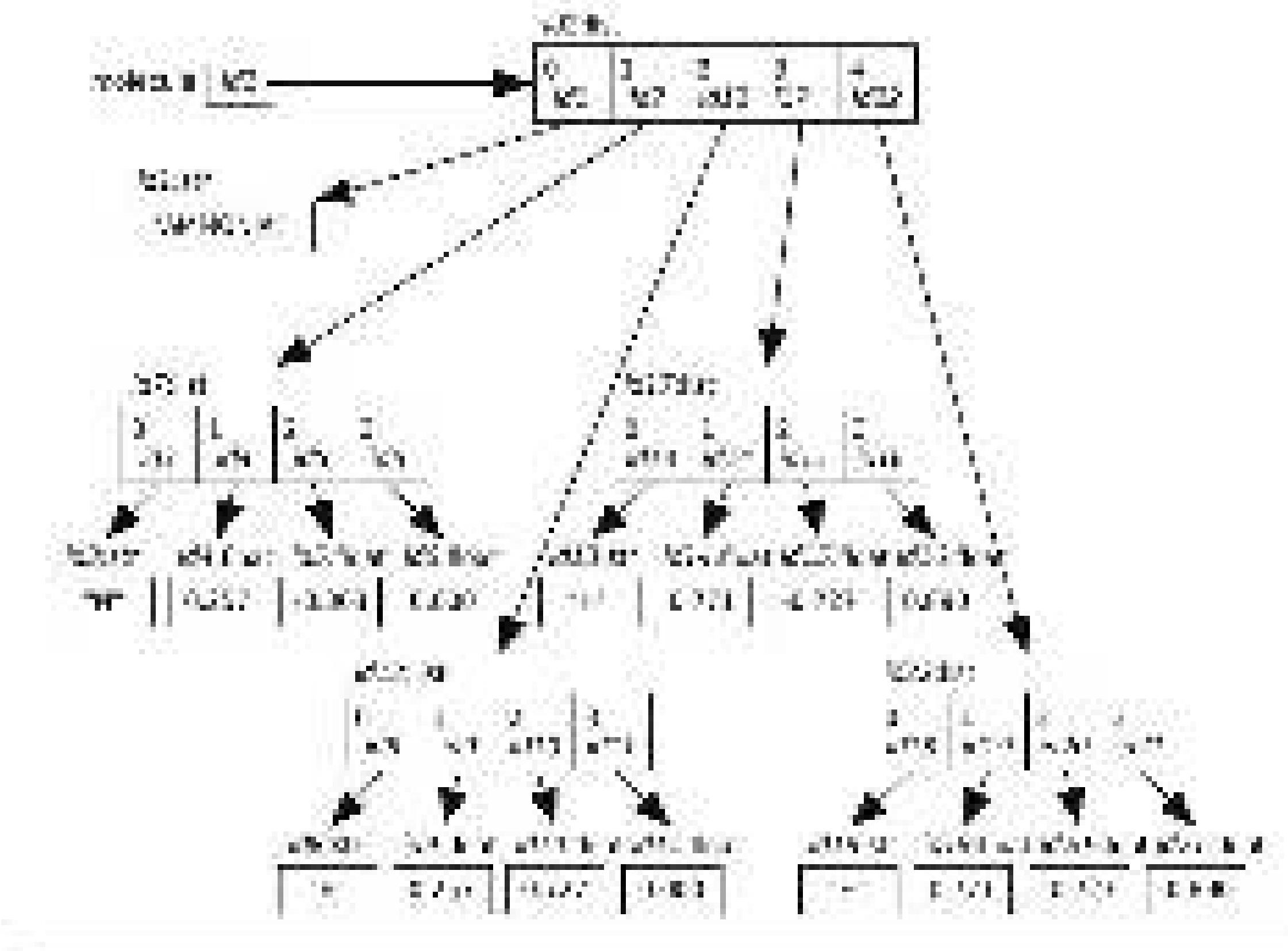


Figure 9—A PDB file Without EBD markers

To solve this problem our functions must return “linked” molecules. Let's start with the function that reads multiple molecules:

```
def read_all_molecules(filename):
    """Read from the file filename lines containing a list of the molecules that
    follow.

    result = []
    line = reader.readline()
    while line:
        molecule, line = read_molecule(reader, line)
        result.append(molecule)

    return result
```

This function begins by reading the first line of the file. Provided that line is not the empty string (that is, the file being read is not empty), it passes both the opened file object and the line to `read_molecule`, which is responsible to return two things: the next molecule in the file and the `line` (immediately after the end of that molecule) or an empty string if the end of the file has been reached.

This simple description is enough to get us started writing the `read_molecule` function. The first thing it has to do is check that `line` is actually the start of a molecule. If there really lines have been read, an `empty` looking string of characters:

- The end of the file, which signals the end of both the current molecule and the file;
- Another CCGTGT line, which signals the end of this molecule and the start of the next one;
- Just a gap, which is to be added to the current molecule.

The most important thing is that when this function returns, it returns both the molecule and the rest line that the caller can keep processing. The result is probably the most complicated function we have seen so far, but understanding the logic behind it will help you understand how it works. Refer to the following code and [Figure 13: Loading ahead on page 143](#):

```
def read_molecule(reader, line):
    """file open for reading and <file>
```

`Read a molecule from reader. When done return to the first line of`

the molecule to be used. Below the  $\alpha$ -carbon and the first three atoms of the  $\beta$ -ketone chain are shown the first two atoms of the  $\gamma$ -ketone chain.

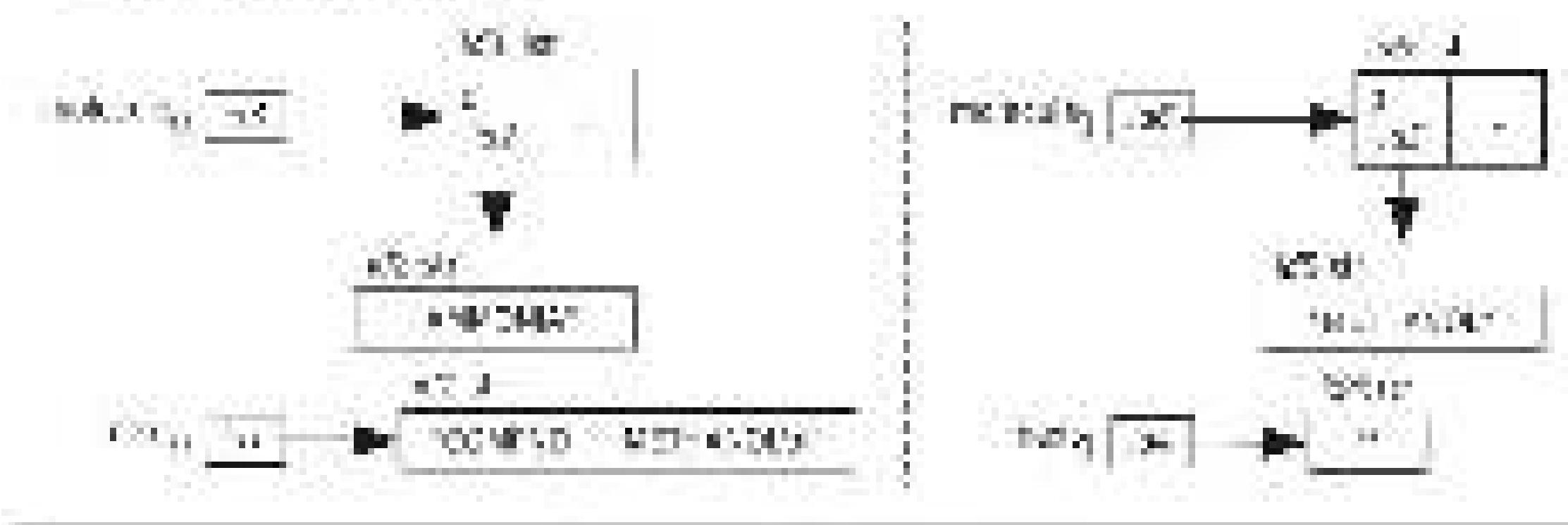
```

    fields = line.split()
    molecule.append(fields)
    line = reader.readline()

while line and not line.startswith('COSAT'):
    fields = line.split()
    if fields[0] == 'STRUCTURE':
        key, num, atom_type = fields[1:4]
        molecule.append([atom_type, num])
    line = reader.readline()

return molecule, line

```



**Figure 2** Double-blind placebo

### 10.9 Notes to File Away

In this chapter, you learned the following:

- When files are opened and read their contents are commonly stored in lists of strings.
  - Data stored as files is usually formatted in one of a small number of ways, from one value per line to multiline records with explicit end-of-record markers. Each format can be processed in a memory-efficient way.
  - Data processing programs should be broken into input, processing and output stages so that each can be tested independently.
  - Files can be read (content read), written (content replaced), and added to (new content appended). When a file is opened as writing mode and it doesn't exist a new file is created.

- These files come in many different formats, we custom want to often re-square but we can reuse as much as possible by writing helper functions.
  - To make the function work for different types of readers, the reader file is wrapped in optional code: the function passed as argument to the function, and then closed outside the function.

## 10.10 Exercises

Here are some exercises for you to try on your own. Solutions are available at <http://www.mathematica-journal.com/resources/exercises.html>.

1. Write a program that makes a backup of a file. Your program should prompt the user for the name of the file to copy and then write a new file with the same contents but with .bak as the file extension.
  2. Suppose the file `data.txt` contains the names, phone numbers, and email address of the following earth mortals:

1996-1997  
1997-1998  
1998-1999  
1999-2000  
2000-2001

With a keypress turned the contents of a buffer into a list and stored it in a list of lists, with each inner list containing the name, phone number, and symbolic weight for an element. (Note: The entry editor)

3. All of the file-handling functions we have seen in this chapter read from and through the file from the first character or line to the last. How could you write a function that would read backward through a file?
  4. In [Processing Shakespeare's Invaluable Data](#), on page 187, we used the “File as List” technique to process data line by line, breaking it into pieces using `split()` method as it. `readline()` function processes file by `skipLine()` function so it is much better to use the “Read” technique to read all the data at once.
  5. Modify the file reader in `read_gutenberg.py` or [Getting the Header](#), on page 193 so that it can handle files with no data after the header.
  6. Modify the file reader in `read_gutenberg.py` or [Skipping the Header](#), on page 193 so that it uses a carriage to end the loop instead of `not 1`. Which form do you find easier to read?
  7. Modify the TSV file reader in [Section 10.7, Matching Records](#), on page 191, so that it ignores blank lines and comment lines in TSV files. A blank line is one

- that contains only space and tab characters (that is, one that looks empty when viewed). A comment is any line beginning with the keyword `/*`.
6. Modify the PDD file reader to check that the serial numbers on atoms start at 1 and increase by 1. What should the method `readAtoms()` do if it finds a file that doesn't obey this rule?

# Storing Data Using Other Collection Types

In Chapter 6, *Storing Collections of Data Using Lists*, on page 129, you learned how to store collections of data using lists. In this chapter, you will learn about three other kinds of collections: sets, tuples, and dictionaries. With four different options for storing your collections of data, you will be able to pick the one that best matches your problem in order to keep your code as simple and efficient as possible.

## 11.1 Storing Data Using Sets

A set is an **unordered** collection of distinct items. Unordered means that items aren't stored in any particular order. Something is either in the set or it's not, but there's no notion of it being the first, second, or last item. Distinct means that any item appears in a set at most once; in other words, there are no duplicates.

Python has a type called `set` that allows us to store mutable collections of unordered, distinct items. (Remember that a `mutable` object is one over which you can modify.) Here we create a set containing three words:

```
my_words = ['a', 'b', 'c', 'd', 'e']
my_words
{'a', 'b', 'c', 'd', 'e'}
```

It looks much like a list, except that `set` has two (that is, 1 and 2) instead of four brackets (that is, [) and (]. Notice that, when displayed in the shell, the set is unordered. Python does some mathematical tricks behind the scenes to make accessing the items very fast, and one of the side effects of this is that the items aren't in any particular order.

Here we show that each item is *distinct*: duplicates are ignored:

```
my_words = ['a', 'b', 'a', 'b', 'c', 'a', 'b', 'c']
my_words
{'a', 'b', 'c'}
```

Even though there were three `a`s and two `b`s when you created the set, only one of each was kept. Python considers the two sets to be equal:

```
>>> ('a', 'b', 'c') == ('a', 'b', 'c')
True
```

The reason they are equal is that they contain the same items. Again, order doesn't matter, and only one of each item is kept.

Variable `names` refers to an object of type `set`:

```
>>> type(names)
<class 'set'>
>>> type({1, 2, 3})
<class 'set'>
```

In [Section 4.1.3, Mutable Data Using Dictionaries](#), you made a small change to a type that uses the `delitem()` method to prevent us from using that method to represent an empty set. Instead, to create an empty set, you need to call function `set` with no arguments:

```
>>> set()
set()
>>> type(set())
<class 'set'>
```

Function `set` expects either no arguments to create an empty set or a single argument that's a collection of values. We can, for example, create a set from a list:

```
>>> set([2, 3, 2, 5])
[2, 3, 5]
```

Because duplicates aren't allowed, only one of the 2s appears in the set.



Function `set` expects at most one argument. You can't pass several values as separate arguments:

```
<-- set(2, 3, 5)
Traceback (most recent call last):
  File "script1.py", line 1, in <module>
    TypeError: set expected at most 1 arguments, got 3
```

In addition to lists, there are a couple of other types that can be used as arguments to function `set`. One is a set:

```
<-- vowels = {'a', 'e', 'i', 'o', 'u', 'A', 'E', 'I', 'O', 'U'}
<-- vowels
{'A', 'E', 'I', 'O', 'U'}
<-- len(vowels)
5
<-- set({2, 3, 4})
{2, 3, 4}
```

Another useful type is range from [Generators, Iterators, and Range](#). In the following code a set is created with the values 3 to 4 inclusive.

```
<-- set(range(5))
{0, 1, 2, 3, 4}
```

In [Section 11.9, Shoring Down Using Tuples](#), on page 204, you will learn about the tuple type, another type of sequence that can also be used as an argument to functions...

## Set Operations

In mathematics, set operations include union, intersection, add, and remove. In Python, these are implemented as methods that a complete list, see [Table 12. Set Operations](#), on page 202. We'll show you these in action.

Sets are mutable, which means you can change what's in a set object. The methods `add`, `remove`, and `clear` all modify what's in a set. The letter `g` is sometimes considered to be a vowel. Here we add it to our set of vowels:

```
<-- vowels = {'a', 'e', 'i', 'o', 'u'}
<-- vowels
{'a', 'e', 'i', 'o', 'u'}
<-- vowels.add('g')
<-- vowels
{'a', 'e', 'i', 'o', 'u', 'g'}
```

Other methods, such as `intersection` and `difference`, return new sets based on their arguments.

In the following code, we show all of these methods in action:

```

<<< len = arr1.range(10)
<<< loevn = {0, 1, 2, 3, 4}
<<< odds = {1, 3, 5, 7, 9}
<<< loevn.add(9)
<<< loevn
{0, 1, 2, 3, 4, 9}
<<< loevn.difference(odds)
{0, 2, 4}
<<< loevn.intersection(odds)
{1, 3, 5}
<<< loevn.isdisjoint(item)
True
<<< loevn.isupper(items)
False
<<< loevn.remove(8)
<<< loevn
{1, 2, 3, 4, 9}
<<< loevn.symmetric_difference(odds)
{0, 2, 4, 5, 7}
<<< loevn.union(odds)
{1, 2, 3, 4, 5, 7, 9}
<<< loevn.clear()
<<< loevn
set()

```

| Method                                            | Description                                                                                                                            |
|---------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------|
| <code>S.add(item)</code>                          | Adds item to a set if—relative to no other items is already in it.                                                                     |
| <code>S.discard()</code>                          | Removes all items from set S.                                                                                                          |
| <code>S.difference(other)</code>                  | Returns a set with items that occur in set S but not in other.                                                                         |
| <code>S.intersection(other)</code>                | Returns a set with items that occur both in sets S and other.                                                                          |
| <code>S.intersection_update(other)</code>         | Returns the final set of all items that occur in both S and other.                                                                     |
| <code>S.isdisjoint(other)</code>                  | Returns True if no elements exist in both sets.                                                                                        |
| <code>S.pop()</code>                              | Removes item's elements.                                                                                                               |
| <code>S.symmetric_difference(other)</code>        | Returns a set with items that occur exactly one of sets S and other. Two items have to be in both sets but not included in the result. |
| <code>S.symmetric_difference_update(other)</code> | Returns a set with items that are either in one or the other but not both.                                                             |

Table 1-2—Set Operations

Many of the tasks performed by methods can also be accomplished using operators. If `s1` and `s2` are two sets, for example, then `s1 | s2` creates a new set containing their union (that is, all the elements from both sets and `base`), while `s1 < s2` tests whether all the values in `s1` are also in `s2`. Some of the operators that help support sets are listed in Table 14, Set Operators.

| Method Call              | Operator                 |
|--------------------------|--------------------------|
| <code>s1 &amp; s2</code> | <code>s1 &amp; s2</code> |
| <code>s1   s2</code>     | <code>s1   s2</code>     |
| <code>s1 ^ s2</code>     | <code>s1 ^ s2</code>     |
| <code>s1 -&gt; s2</code> | <code>s1 == s2</code>    |
| <code>s1 &lt; s2</code>  | <code>s1 &lt; s2</code>  |
| <code>s1 &lt;= s2</code> | <code>s1 ^ s2</code>     |

Table 14. Set Operators

The following code shows the set operations in action:

```
odd := {x | x < 10, x % 2 != 0}
even := {x | x < 10, x % 2 == 0}
even < odd = {x | x < 10, x % 2 == 0} < {x | x < 10, x % 2 != 0}
{0, 1, 2}
even < odd < odd = {x | x < 10, x % 2 == 0} < {x | x < 10, x % 2 != 0} < {x | x < 10, x % 2 != 0}
{1, 2}
even < odd < odd = {x | x < 10, x % 2 == 0} < {x | x < 10, x % 2 != 0} < {x | x < 10, x % 2 == 0}
{0, 1, 2, 3, 4, 5, 6, 7, 8}
even ^ odd = {x | x < 10, x % 2 != 0} ^ {x | x < 10, x % 2 == 0}
{0, 2, 4, 6, 8}
```

## Arctic Birds

In the last section we used `superset` to have a file used to record observations of birds in the Canadian Arctic and you want to know which species have been observed. The observations file, `observations.txt`, has one species per line:

```
canada goose
canada goose
long-tailed jaeger
canada goose
snow goose
canada goose
long-tailed jaeger
canada goose
swallow tailed gull
```

The program looks much like the line of the file, strips off the leading and trailing white space, and adds the species on that line to the set:

```
>>> observations_set = open('observations.txt', 'r')
>>> birds_observed = set()
>>> for line in observations_file:
...     bird = line.strip()
...     birds_observed.add(bird)
...
>>> birds_observed
{'long-tailed jaeger', 'canada geese', 'northern shrike', 'snow geese'}
```

The resulting set contains four species. Since sets don't contain duplicates, calling the `add` method with a species already in the set had no effect.

You can loop over the values from a set. In the code below, `for` loops is used to print each species:

```
>>> for species in birds_observed:
...     print(species)
...
long-tailed jaeger
canada geese
northern shrike
snow geese
```

Looping over a set works exactly like a loop over a list, except that the order in which items are enumerated is arbitrary. This is no guarantee that they will come out in the order in which they were added, as alphabetical order, in order by length, or to any other order.

## 11.2 Storing Data Using Tuples

This section introduces one kind of ordered sequence in Python. You've already learned about one of the others: strings (see Chapter 4, Working with Text, on page 87). Normally, a string is an immutable sequence of characters. The characters in a string are read-only and writing can be tedious and slow, like a list to iterate over elements:

```
>>> rock = "anthracite"
>>> rock[0]
'�'
>>> rock[0:2]
'ant'
>>> rock[-2:]
'actite'
>>> for character in rock[5:]:
...     print(character)
...
t
i
t
e
```

```
a
b
t
h
y
```

Python also has an immutable sequence type called a tuple. Tuples are written using parentheses instead of brackets; like strings and lists, they can be subscripted, sliced, and iterated over:

```
>>> bases = ('A', 'C', 'G', 'T')
>>> for base in bases:
...     print(base)
...
A
C
G
T
```

This is similar to list, although () represents the empty tuple, a tuple with one element is not written as (a) but as (a,) (with a trailing comma). This is done to avoid ambiguity. If the trailing commas weren't required, (5 - 5) could mean either to turn off the normal rules of arithmetic or the tuple containing only the value 5.

```
>>> (5)
5
>>> type((5))
<type 'tuple'>
>>> (5, )
(5, )
>>> type((5, ))
<type 'tuple'>
>>> (5 + 2)
6
>>> (5 + 2, )
(5, )
```

Unlike lists, once a tuple is created, it cannot be mutated:

```
>>> life = [('Canada', 36.5), ('United States', 35.5), ('Russia', 22.5)]
>>> life[0] = life[1]
Traceback (most recent call last):
  File <stdin>, line 1, in <module>
TypeError: object does not support item assignment
```

However, the objects inside it can still be mutated:

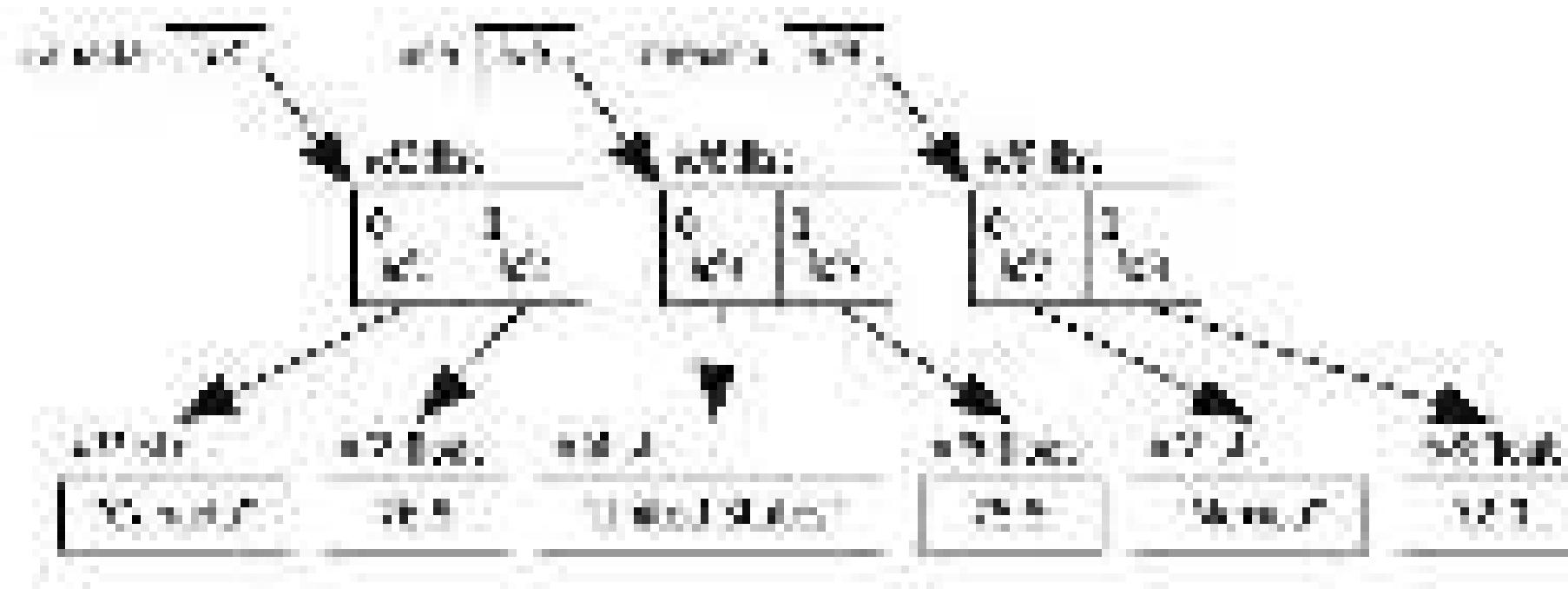
```
>>> life = [('Canada', 36.5), ('United States', 35.5), ('Russia', 22.5)]
>>> life[0][1] = 38.0
>>> life
[('Canada', 38.0), ('United States', 35.5), ('Russia', 22.5)]
```

Rather than saying that a tuple cannot change, it is more accurate to say that the references contained in a tuple cannot be changed after the tuple has been created. Though the objects referred to may themselves be modified.

Here's an example that explores what is mutable and what isn't. We'll build the same tuple as in the previous example, but we'll do it in steps. First let's create three lists:

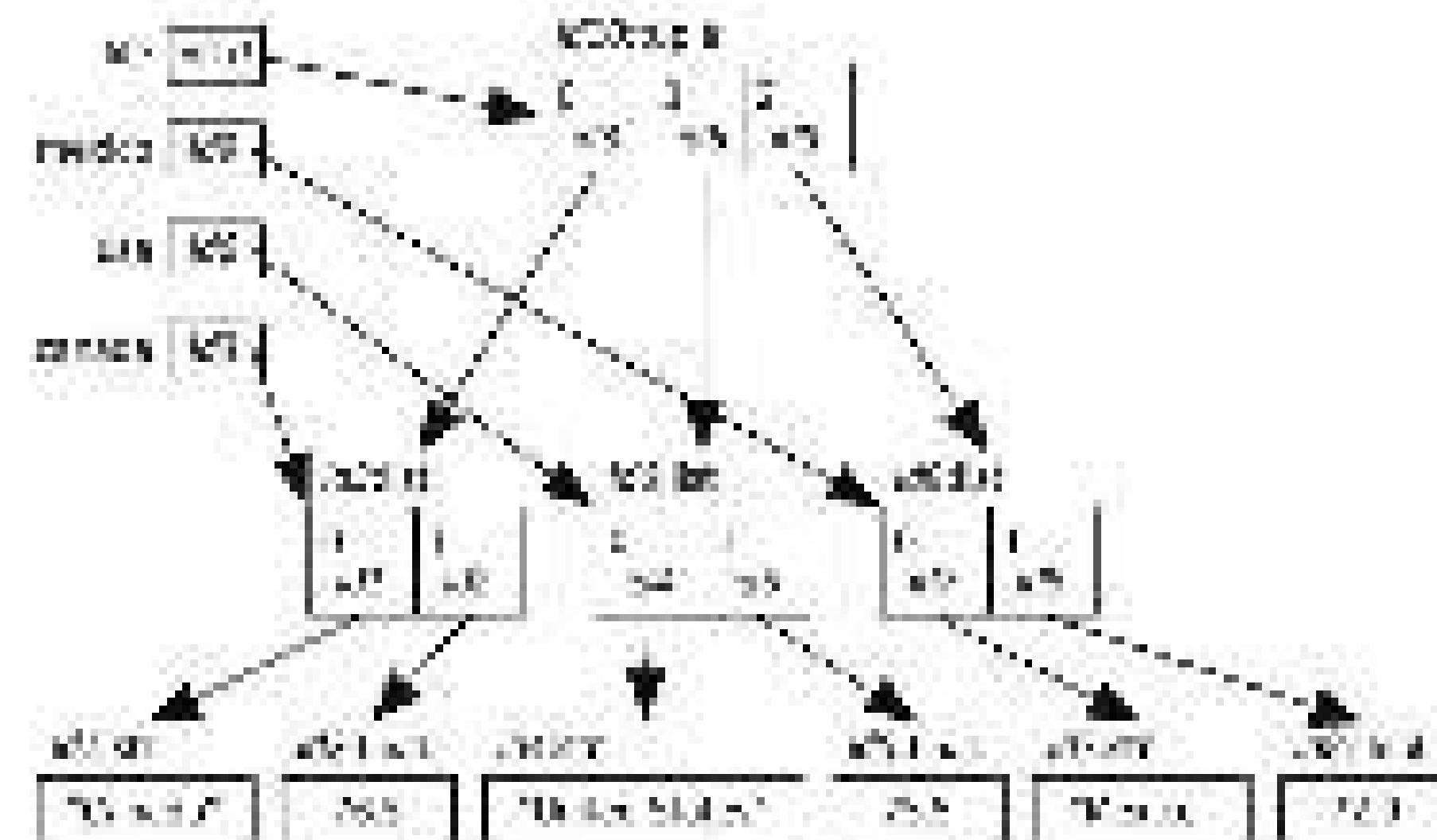
```
ccc_canada = ['Canada', 36.5]
ccc_usa = ['United States', 35.5]
ccc_mexico = ['Mexico', 32.0]
```

That looks like memory initial:



We'll create a tuple using those variables:

```
ccc_life = (ccc_canada, ccc_usa, ccc_mexico)
```

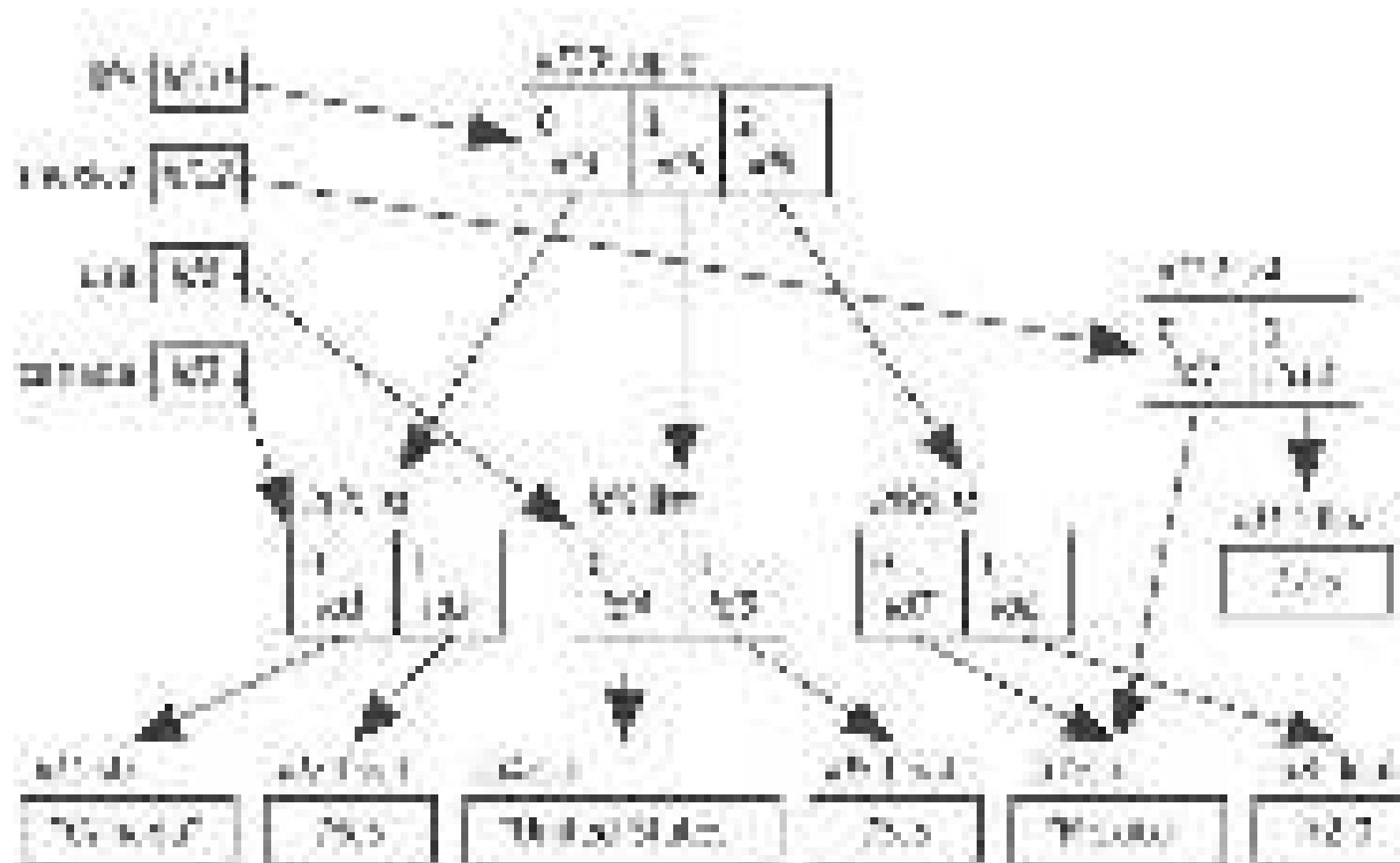


The most important thing to notice is that none of the four variables know about the others. The tuple object contains three references, one for each of the country lists.

Now let's change what variable `canada` refers to:

```
>>> mexico = ('Mexico', 72.5)
>>> life
('Canada', 76.5), ('United States', 78.5), ('Russia', 72.2)
```

Notice that the tuple that variable `life` refers to hasn't changed. Here's the new picture:



`life` will always refer to the same list object. we can't change the memory address stored in `life` but we can mutate that list object; and because variables `canada` also refers to that list, it won't be mutated [see [Mutating the List Object](#), on page 206].

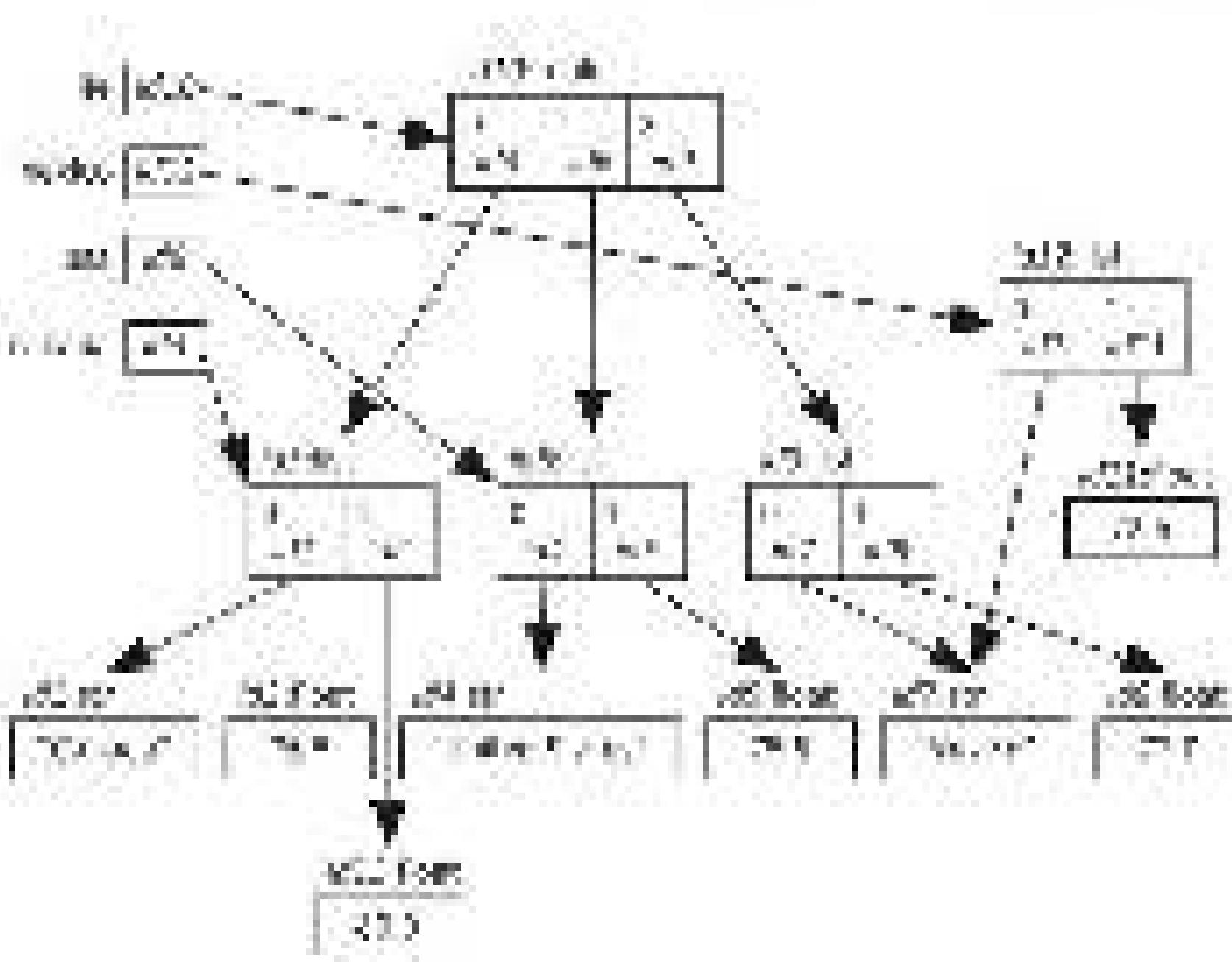
```
>>> life[0][1] = 86.6
>>> canada
('Canada', 86.6)
```

We hope that it is clear how **essential it is to correctly understand variables and references**, and how **mutation of certain references to objects** and not to variables.

## Assigning to Multiple Variables Using Tuples

You can assign to multiple variables at the same time:

```
>>> (x, y) = (10, 20)
>>> x
10
>>> y
20
```

**Figure 11—Mutating the List Object**

As with a normal assignment statement (see [Assignment Statement](#), on page 14), Python first evaluates all expressions on the right side of the `=` symbol, and then it assigns those values to the variable on the left side.

Python uses the comma as a tuple separator, so we can leave off the parentheses:

```
>>> 10, 20
(10, 20)
>>> x, y = 10, 20
>>> x
10
>>> y
20
```

In fact, multiple assignment will work with lists and sets as well. Python will happily pull apart information and put any collection in:

```
>>> [10, x1, y1, z1] = [10, 20, 100, 1, 40]
>>> x
10
>>> y
20
>>> z
100
>>> x1
1
>>> y1
40
```

Note: depth of nesting will work as long as the structure on the right can be translated into the structure on the left.

One of the most common uses of multiple assignment is to swap the values of two variables:

```
one.x1 = "first"
one.x2 = "second"
one.x1, one.x2 = one.x2, one.x1
one.x1
"second"
one.x2
"first"
```

This works because the assignments on the right side of operator = are evaluated before assigning to the variables on the left side.

### 11.3 Storing Data Using Dictionaries

Here is the same bird-watching observation file that we saw in [Arctic Birds](#), on page 203:

```
canada goose
canada goose
long-tailed jaeger
canada goose
snow goose
canada goose
long-tailed jaeger
canada goose
northern fulmar
```

Suppose we want to know how often each species was seen. Our first attempt uses a list of lists, in which each inner list has two items. The item at index 0 of the inner list contains the species, and the item at index 1 contains the number of times it has been seen so far. To build this list, we iterate over the lines of the observations file. For each line, we iterate over the line looking for the species on that line. If we find that the species occurs in the list, we add one to the number of times it has been observed.

```
one observations_file = open("observations.txt", "r")
one bird_counts = []
one for line in observations_file:
...     bird = line.rstrip()
...     found = False
...     # Check if it is in the list of bird counts.
...     for entry in bird_counts:
...         if entry[0] == bird:
...             entry[1] = entry[1] + 1
...             found = True
...     if not found:
...         bird_counts.append([bird, 1])
```

```

... if not found:
    bird_count.append([bird, 1])
...
for observations in birds:
    for entry in bird_count:
        print(entry[0], entry[1])
...
canada goose 5
long-tailed jaeger 2
snow goose 1
northern fulmar 1

```

The code above uses a Boolean variable, `found`. Once a species is read from the file, it is assigned to `bird`. The program then loops over the list, looking for the position at index 0 of one of the items. If the species occurs in an item list, `found` is assigned true. At the end of the loop over the list, if `found` still remains false it means the tree species is not yet present in the list and so it is added, along with the number of observations to it, which is currently 1.

The code above works, but there are two things wrong with it. The first is that it is **inefficient**. The more nested loops our programme contains, the harder they are to understand, fix, and extend. The second is that it is **inflexible**. Suppose we were interested in **leeches** instead of birds and that we had millions of observations of tens of thousands of species. Scanning the list of names each time we want to add one new observation would take a long time, even on a fast computer (a topic we will return to in Chapter 18, [Scanning and Sorting](#) on page 207).

Can you use a set to solve both problems at once? Sets can look up values in a single step. Why not combine each bird's name and the number of times it has been seen into a two-valued tuple and put those tuples in a set?

The problem with this idea is that you can look for values only if you know what those values are. In this case, you won't. You will know only the name of the species, not how many times it has already been seen.

The right approach is to use another data structure called a **dictionary**. Also known as a map, a dictionary is an unordered **mutable** collection of key/value pairs. In plain English, Python's dictionaries are like dictionaries that map names to definitions. They associate a key (like a word) with a value (such as a definition). The keys form a set; any particular key can appear several times in a dictionary; and like the elements in sets, keys must be immutable. Though the values associated with them don't have to be.

Dictionaries are created by putting key/value pairs inside braces (such that `key` is followed by a colon and then by the value):

`{'name': 'canada goose', 'count': 5}`

```
1>>> bird_to_observations = { 'canada goose': 1, 'northern fulmar': 3}
2>>> bird_to_observations
{'northern fulmar': 3, 'canada goose': 1}
```

We can see our dictionary has two observations so far. This example refers to a dict: the keys where each key is a bird and each value is the number of observations of that bird. In other words, the dictionary maps birds to observations. Here is a picture of the resulting dictionary:



The get() function associated with a key returns the key in square brackets, much like indexing into a list.

```
1>>> bird_to_observations['northern fulmar']
3
```

Looking up a dictionary with a key it doesn't contain produces an error, just like an IndexError: occurs for a list slice:

```
1>>> bird_to_observations['canada goose']
3
2>>> bird_to_observations['long-tailed jaeger']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'long-tailed jaeger'
```

The empty dictionary is written {} (that's why we can't use this notation for the empty set). It doesn't contain any key/value pairs, so translating into it always results in an error.

## Updating and Checking Membership

To update the value associated with a key, you use the same notation as for lists, except you use a key instead of an index. If the key is already in the dictionary, this assignment statement changes the value associated with it. If the key isn't present, the key/value pair is added to the dictionary:

```
1>>> bird_to_observations = {}
2>>> bird_to_observations['canada goose'] = 22
3>>> bird_to_observations['canada goose'] = 23
4>>>
```

```
>>> # Add a new key:value pair, 'eagle': 100.
>>> bind_to_observations['eagle'] = 100
>>> bind_to_observations
{'eagle': 100, 'snow geese': 10}
>>>
>>> # Change the value associated with key 'eagle' to 5.
>>> bind_to_observations['eagle'] = 5
>>> bind_to_observations
{'eagle': 5, 'snow geese': 10}
```

To remove an entry from a dictionary, use `del d[k]`, where `d` is the dictionary and `k` is the key being removed. Only entries that are present can be removed; trying to remove one that isn't there results in an error:

```
>>> bind_to_observations = {'snow geese': 10, 'eagle': 5}
>>> del bind_to_observations['snow geese']
>>> bind_to_observations
{'eagle': 5}
>>> del bind_to_observations['goose'] # This will raise an exception!
Traceback (most recent call last):
  File "c:\users\jordan\documents\scripting\scripting\exercises\01\scripting.py", line 10, in <module>
    print(bind_to_observations['goose'])
```

To test whether a key is in a dictionary, we can use the `in` operator:

```
>>> bind_to_observations = {'eagle': 500, 'snow geese': 40}
>>> if 'eagle' in bind_to_observations:
...     print('eagles have been seen')
...
eagles have been seen
>>> del bind_to_observations['eagle']
>>> if 'eagle' in bind_to_observations:
...     print('eagles have been seen')
...
>>>
```

The `in` operator only checks the keys of a dictionary. In the example above, `'eagle' in bind_to_observations` is false, since 500 is a value, not a key.

## Looping Over Dictionaries

Like the other collections you've seen, you can loop over dictionaries. The general form of a for loop over a dictionary is as follows:

```
for <variable> in <dictionary>:
    <block>
```

For dictionaries, the loop variable is assigned each key from the dictionary in turn:

```
>>>
```

```

>>> bird_to_observations = {'canada goose': 100, 'long-tailed jaeger': 21,
...                           'snow goose': 63, 'northern fulmar': 1}
>>> for bird in bird_to_observations:
...     print(bird, bird_to_observations[bird])
...
long-tailed jaeger 21
canada goose 100
northern fulmar 1
snow goose 63

```

Note that long tails prints first, yet in the assignment statement, the last key/value pair listed is 'canada goose'. As with the `keys()` method, Python loops over the keys in the dictionary in an arbitrary order. There is no guarantee they will be seen alphabetically or in the order they were added to the dictionary.

When Python loops over a dictionary, it iterates the keys in the dictionary. It's a little easier to go from a dictionary key to the associated value than it is to take the value and find the associated key.

## Dictionary Operations

Like lists, tuples, and sets, dictionaries are mutable. Their methods are identical to those shown in Table 1.5, [Dictionary Methods](#), on page 314. The following code shows how the methods can be used:

```

>>> scientist_to_birthdate = {'Darwin': 1809, 'Newton': 1642,
...                            'Hawking': 1942}
>>> scientist_to_birthdate.keys()
dict_keys(['Darwin', 'Newton', 'Hawking'])
>>> scientist_to_birthdate.values()
dict_values([1809, 1642])
>>> scientist_to_birthdate.items()
dict_items([('Darwin', 1809), ('Newton', 1642), ('Hawking', 1942)])
>>> scientist_to_birthdate.get('Hawking')
1642
>>> scientist_to_birthdate.get('Curie', 1867)
1867
>>> scientist_to_birthdate
{'Darwin': 1809, 'Newton': 1642, 'Hawking': 1942}
>>> researcher_to_birthdate = {'Curie': 1867, 'Hopper': 1906,
...                                'Franklin': 1727}
>>> scientist_to_birthdate.update(researcher_to_birthdate)
>>> scientist_to_birthdate
{'Hopper': 1906, 'Darwin': 1809, 'Newton': 1642, 'Hawking': 1942,
... 'Franklin': 1727, 'Curie': 1867}
>>> researcher_to_birthdate
{'Franklin': 1727, 'Hopper': 1906, 'Curie': 1867}
>>> researcher_to_birthdate.clear()
>>> researcher_to_birthdate
()
```

| Method                        | Description                                                                                                                                                                                                                                |
|-------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>clear()</code>          | Removes all key/value pairs from dictionary D.                                                                                                                                                                                             |
| <code>get(k)</code>           | Returns the value associated with key k, or None if the key isn't present. (Usually you'll want to use <code>if key in dict:</code> )                                                                                                      |
| <code>get(k, v)</code>        | Returns the value associated with key k, or a default value v if the key isn't present.                                                                                                                                                    |
| <code>keys()</code>           | Returns the keys of dictionary D; keys are set-like objects—entries are guaranteed to be unique.                                                                                                                                           |
| <code>items()</code>          | Returns dictionary D's items as key-value pairs as set-like objects.                                                                                                                                                                       |
| <code>D.pop(k)</code>         | Removes key k from dictionary D and returns the value that was associated with k—if k isn't in D, an error is raised.                                                                                                                      |
| <code>D.pop(k, v)</code>      | Removes key k from dictionary D and returns the value that was associated with k; if k isn't in D, returns v.                                                                                                                              |
| <code>setdefault(k, v)</code> | Returns the value associated with key k in D.                                                                                                                                                                                              |
| <code>update(D, v)</code>     | Returns the value associated with key k in D; if there's no key in D, adds the key k with the value v to D and returns v.                                                                                                                  |
| <code>values()</code>         | Returns dictionary D's values as a list-like object—entries may or may not be unique.                                                                                                                                                      |
| <code>update(other)</code>    | Updates dictionary D with the contents of dictionary other. If each key in other has a value in D, replaces that key in D + value with the value from other. For each key in other, if that key isn't in D, adds that key/value pair to D. |

**TABLE 10—Dictionary Methods**

As you can see from this output, the `keys` and `values` methods return the dictionary's keys and values, respectively, while `item` returns the key-value pairs. Like the `range` object that you learned about previously, these are iterable sequences over which we can loop. Similarly, functions like `map` can be applied to them in exactly lists of key/value or key/value tuples.

Because dictionaries usually map values from one concept (identifiers) in our program to another (variables), the common use case variable names linking the two—hence, `var_to_id`—isable.

One common use of `var_to_id` is to loop over the keys and values in a dictionary together:

```
for key, value in dictionary.items():
    do something with key and value
```

For example, the `scientist_to_birthday` can be used to keep track of the scientists and their birth years:

```
res scientist_to_birthday = {(Rutherford : 1842, Darwin : 1809,
                             Turing : 1912),
    for scientist, birthday in scientist_to_birthday.items():
        print(f'{scientist} was born in {birthday}.')
}

Hawking was born in 1942
Darwin was born in 1809
Rutherford was born in 1842
```

Instead of a single `list` variable, there are two. The two parts of each of the two-item tuples returned by the method `items` is associated with a variable. Variable `scientist` refers to the first item in the tuple, which is the key, and `birthday` refers to the second item, which is the value.

## Dictionary Example

Back to birdwatching once again. Like before, we want to count the number of times each species has been seen. To do this, we create a dictionary that is initially empty. Each time we read an observation from a file, we check to see whether we have encountered that bird before—that is, whether the bird is already a key in our dictionary. If it is, we add 1 to the value associated with it. If it isn't, we add the bird as a key to the dictionary with the value 1. Here's the program that does this:

```
res observations_file = open('observations.txt')
res bird_to_observations = {}
res for line in observations_file:
    ...     bird = line.strip()
    ...     if bird in bird_to_observations:
    ...         bird_to_observations[bird] = bird_to_observations[bird] + 1
    ...     else:
    ...         bird_to_observations[bird] = 1
...
res observations_file.close()
res
res # Print over counted the number of times it was seen
for bird, observations in bird_to_observations.items():
    print(bird, observations)

More species 1
Long-tailed Jaeger 2
Canada Goose 5
no return values 1
```

This program can be substantially cut by using the `dict.get` method with three lines:

`... observations = observations.get(bird, 0) + 1`

```
>>> observations_file = open('observations.txt')
>>> bird_to_observations = {}
>>> for line in observations_file:
...     bird = line.strip()
...     bird_to_observations[bird] = bird_to_observations.get(bird, 0) + 1
...
>>> observations_file.close()
```

Using the `get` method makes the program shorter, but more programming overhead is introduced at a glance. If the first argument to `get` is not a key in the dictionary + returns 0; otherwise it returns the value associated with that key. After that, 1 is added to that value. The dictionary is updated to associate that sum with the key that `bird` refers to.

Instead of printing the birds' names in whatever arbitrary order they are inserted by the loop, you can create a list of the dictionary's keys, sort that list alphabetically, and then loop over the sorted list. This way, the entries appear in a guaranteed order:

```
>>> sorted_birds = sorted(bird_to_observations.keys())
>>> for bird in sorted_birds:
...     print(bird, bird_to_observations[bird])
...
canada geese 5
long-tailed jaeger 2
northern fulmar 1
swan geese 1
```

If order matters, then an ordered sequence like lists or tuples should be used instead of sets and dictionaries.

## 11.4 Inverting a Dictionary

You might want to print the birds in another order—an order of the number of observations, for example. To do this, you need to invert the dictionary: that is, create a new dictionary in which you swap the values as keys and the keys as values. That is a little trickier than it first sounds. There's no guarantee that the values are unique, so you have to handle what are called *collisions*. For example, if you invert the dictionary `{'P': 1, 'W': 1, 'R': 1}`, a key would be 1, but it's not clear what the value associated with it would be.

Since you'd like to keep all of the data from the original dictionary, you may need to use a collection, such as a list, to keep track of the values associated with a key. If we go this route, the inverse of the dictionary shown earlier would be `[('P', 1), ('W', 1), ('R', 1)]`. Here's a program to invert the dictionary of birds to observations:

```

>>> bird_to_observations
{'canada geese': 5, 'northern fulmar': 1, 'long-tailed jaeger': 3,
'snow geese': 1}

>>> # Insert the dictionary
>>> observations_to_birds_list = []
>>> for bird, observations in bird_to_observations.items():
...     if observations in observations_to_birds_list:
...         observations_to_birds_list[observations].append(bird)
...     else:
...         observations_to_birds_list.append([observations + bird])
...
>>> observations_to_birds_list
[[1, 'northern fulmar', 'snow geese'], [3, 'long-tailed jaeger'],
[5, 'canada geese']]

```

The program above keeps track of each key/value pair in the original dictionary, `bird_to_observations`. If that value is not yet a key in the inserted dictionary, `observations_to_birds_list`, it is added as a key and its value is a single-item list containing the key associated with from the original dictionary. On the other hand, if that value is already a key, then the key associated with it in the original dictionary is appended to its list of values.

Now that the dictionary is inserted, you can print each key and all of the items in its value list:

```

>>> # Print the inserted dictionary
... observations_sorted = sorted(observations_to_birds_list.items())
>>> for observations in observations_sorted:
...     print(observations, ':', end=" ")
...     for bird in observations_to_birds_list[observations]:
...         print(" ", bird, end=" ")
...     print()
...
1 : northern fulmar snow geese
2 : long-tailed jaeger
5 : canada geese

```

The outer loop goes over each key in the inserted dictionary, and the inner loop goes over each of the items in the values list associated with that key.

## 11.5 Using the In Operator on Tuples, Sets, and Dictionaries

As with lists, the `in` operator can be applied to tuples and sets to check whether an item is a member of the collection:

```

>>> odds = set([1, 3, 5, 7, 9])
>>> 9 in odds
True

```

```

>>> 6 in odds
False
>>> 10 in odds
False
>>> 5 in evens
True
>>> 11 in evens
False

```

When used on a dictionary, it checks whether a value is a key in the dictionary:

```

>>> bird_to_observations = { 'canada geese': 104, 'long-tailed jaegers': 21,
...     'northern gannet': 63, 'northern shrike': 11}
>>> 'canada geese' in bird_to_observations
True
>>> 108 in bird_to_observations
False

```

Note that the values in the dictionary are ignored. The operator only looks at the keys.

## 11.6 Comparing Collections

You've now seen strings, lists, sets, tuples, and dictionaries. They all have their uses. Here is a table comparing them:

| Collection | Mutable? | Ordered? | Use When...                                                                                                                               |
|------------|----------|----------|-------------------------------------------------------------------------------------------------------------------------------------------|
| str        | No       | No       | You want to keep track of text.                                                                                                           |
| list       | Yes      | Yes      | You want to keep track of an ordered sequence that you want to update.                                                                    |
| tuple      | No       | No       | You want to build an ordered sequence that you know won't change or that you want to use as a key in a dictionary or as a value in a set. |
| set        | Yes      | No       | You want to keep track of values, but order doesn't matter, and you don't want to keep duplicates. The values must be immutable.          |
| dictionary | Yes      | No       | You want to keep a mapping of keys to values. The keys must be immutable.                                                                 |

Table 15—Features of Python Collections

## 11.7 A Collection of New Information

In this chapter, you learned the following:

- Sets are used in Python to store unordered collections of unique values. They support the same operations as sets in mathematics.
- Tuples are another kind of Python sequence. Tuples are ordered sequences like lists, except they are immutable.
- Dictionaries are used to store unordered collections of key:value pairs. The keys must be immutable, but the values need not be.
- Looking things up in sets and dictionaries is much faster than searching through lists. If you have a program that is doing the latter, consider changing your choice of data structures.

## 11.8 Exercises

Here are some exercises for you to try on your own. Solutions are available at <https://nostarch.com/crackingtheinterviewsolutions>.

1. Write a function called `find_dups` that takes a list or `tuple` as its argument and returns a set of those integers that occur two or more times in the list.
2. Python's `set` objects have a method called `pop` that removes and returns an arbitrary element from the set. If the set gets empty, `pop` suddenly fails silently, for example, calling `pop` five times will return these results one by one, leaving the set empty at the end. Use this to write a function called `read_pairs` that takes two equal-sized lists of numbers and strings as input and returns a set of tuples; each pair must be a tuple containing one mole and one female. The elements of `mole` and `females` may be strings containing full names or just ID numbers—your function must work with both.
3. The PCD file format is often used to store information about molecules. A PCD file may contain many more lines, just begin with the word `AM: 11` (which may be in uppercase, lowercase, or mixed case), followed by spaces or tabs, followed by the name of the person who created the file. While a function can take a list of filenames as an input argument and returns the set of all author names found in those files.
4. The keys in a dictionary are guaranteed to be unique, but the values are not. Write a function called `count_values` that takes a single dictionary as an argument and returns the number of distinct values it contains. Given the input '`red: 1, green: 1, blue: 2`', for example, it should return 2.

- b. After doing a series of experiments, you have compiled a dictionary showing the probability of detecting certain kinds of subatomic particles. The particles' names are the dictionary's keys, and the probabilities are the values: `proton: 0.55, positron: 0.31, neutron: 0.18, muon: 0.11, neutrino: 0.05`. Write a function that takes a single dictionary of this kind as input and returns the particle that is least likely to be observed. Given the dictionary shown earlier, for example, the function would return `neutrino`.
- c. Write a function called `count_duplicates` that takes a dictionary as its argument and returns the number of values that appear two or more times.
- d. A balanced color is one whose red, green, and blue values add up to 1.0. Write a function called `is_balanced` that takes a dictionary whose keys are R, G, and B and whose values are between 0 and 1 as input and returns True if they represent a balanced color.
- e. Write a function called `dict_intersect` that takes two dictionaries as arguments and returns a dictionary that contains only the key:value pairs found in both of the original dictionaries.
- f. Programmers sometimes use a dictionary of dictionaries as a simple database. For example, in a database of information about famous scientists, you might have a dictionary where the keys are strings and the values are dictionaries, like this:

```

{'jessica': {'surname': 'Watson',
             'forename': 'Anne',
             'born': 1942,
             'died': None,
             'titles': 'professor researcher',
             'books': ['In the Shadow of Man',
                       'The Double Helix' [3]]},
 'franklin': {'surname': 'Franklin',
             'forename': 'Barbara',
             'born': 1920,
             'died': 1957,
             'titles': 'contributed to discovery of DNA'},
 ...
}

```

```

{'carroll': {'surname': 'Carroll',
             'forename': 'Rachael',
             'born': 1937,
             'died': 1964,
             'titles': 'raised awareness of effects of DDT',
             'books': ['Silent Spring' [1]}]
}

```

- Write a function called `dot_product` that returns the dot product of two lists of the same dimensions. In this example, the function should return `selfish_breeding * sumnet_votes + 100 * elect`:
- Write another function, called `dot_product`, that takes a dictionary of dictionaries in the format described in the previous question and returns `True` if and only if every one of the three dictionaries has exactly the same keys. (This function would return false for the previous example, since Franklin Franklin's entry doesn't contain the 'other' key.)
  - A sparse vector is a list of numbers that all sum to 1. For [0.1, 0.1, 0.1, 0.1, 0, 0, 0], summing all those terms in a list makes it easy to remember we're dealing with vectors instead of just raw numbers. For example, the vector shown earlier would be represented as [0.1, 0.1]. Because the vector it is meant to represent has the value 1 at index 0 and the value 3 at index 0.
  - The sum of two vectors is just the element-wise sum of their elements. For example, the sum of [1, 2, 3] and [4, 5, 6] is [5, 7, 9]. Write a function called `sparse_dot` that takes two sparse vectors stored as dictionaries and returns a new dictionary representing their sum.
  - The dot product of two vectors is the sum of the products of corresponding elements. For example, the dot product of [1, 2, 3] and [4, 5, 6] is 1\*4+2\*5+3\*6=32. Write another function called `sparse_dot` that calculates the dot product of two sparse vectors.
  - Your boss has asked you to write a function called `sparse_lcr` that will return the length of a sparse vector (just as Python's len returns the length of a list). What do you need to ask her before you can start writing it?

# Designing Algorithms

An **algorithm** is a set of steps that accomplishes a task, such as the steps involved in synthesizing calcium. You can think of a program, as well as the program itself, as an algorithm that is written in a programming language like Python. Writing a program directly in Python, without careful planning, can waste hours, days, or even weeks of effort. Instead, programmers often write algorithms in a combination of English and mathematics and then translate it into Python.

In this chapter, you'll learn an algorithm-writing technique called **top-down design**. You start by describing your solution in English, and then mark the phrases that correspond directly to Python statements. Those that don't correspond are then rewritten to more detail in English, until everything in your description can be written in Python.

Top-down design is easy to describe, but doing it requires a little practice. Often, parts of an algorithm written in English will be easy to translate into Python. In fact, an implementation may be reasonable but will contain bugs. This is common to many fields. In mathematics, for example, the first versions of “proofs” often handle corner cases well but fail the real test of rigor and rigorously showing. Mathematicians deal with this by breaking down increasingly complex, and programming good programmers, often deal with it by testing their code as they write it.

In this chapter, we have slacked the discussion of testing our code to make sure it works. In fact, the first versions we wrote had major bugs in them, and we found them only by going through testing. We will talk more about testing in Chapter 16, Testing and Debugging, on page 297.

## 12.1 Searching for the Smallest Values

This section will explain how to find the index of the smallest value in an unsorted list using three quite different algorithms. We'll go through a top-down design using each approach.

To start, suppose we have data showing the number of Chumback whales sighted off the coast of British Columbia over the past ten years:

```
809  234  477  478  367  122  99  162  324  476
```

The first value, 809, represents the number of sightings in year 1, and the last one, 476, represents the number of sightings last year.

We want to know how many sightings per year have improved their total numbers. Our first question is: what was the lowest number of sightings during those years? This code tells us just that:

```
ccc counts = [809, 234, 477, 478, 367, 122, 99, 162, 324, 476]
ccc min(counts)
99
```

If we want to know in which year the population reached out, we can use `index` to find the position of the minimum:

```
ccc counts = [809, 234, 477, 478, 367, 122, 99, 162, 324, 476]
ccc loc = min(counts)
ccc min_index = counts.index(loc)
ccc print(loc,min_index)
6
```

And here is a more “elegant” version:

```
ccc counts = [809, 234, 477, 478, 367, 122, 99, 162, 324, 476]
ccc counts.index(min(counts))
6
```

Now, what if we want to find the indices of the two smallest values? Lists don't have a method to do this directly, so we'll have to design an algorithm ourselves and then translate it to a Python function.

Here is the header for a function that does this:

```
def first_two_smallest(L)
    """L is a list of floats >= 0.0. L is not empty. Return (I, J) where I < J and
    L[I] <= L[J] and L[I] <= all other elements in L except L[J]."""
    ccc first_two_smallest([809, 234, 477, 478, 367, 122, 99, 162, 324, 476])
    (6, 7)
```

As you may recall from Section 3.6, *Divide-and-Conquer Algorithms: A Recipe*, on page 47, the next step in the function design recipe is to write the function body.

There are at least four ways we could do this, each of which will be equivalent to top-down design. We'll start by giving a very high-level description of each. Each of these descriptions is the first step to developing your own code for that approach.

- **Find, remove, find.** Find the index of the minimum, remove that item from the list, and find the index of the new minimum item in the list. After we have the second index, we must go back the value in numerical and, if necessary, adjust the second index to account for that reflection.
- **Find, identify minimum, get indices.** Sort the list, get the two smallest numbers, and then find their indices in the original list.
- **Walk through the list.** Examine each value in the list in order; keep track of the two smallest values found so far, and update those values when a new smaller value is found.

The first two algorithms modify the list either by removing an item or by sorting the list. It's vital that our algorithms put things back the way we found them, or the people who call our functions are going to be annoyed with us.

While you are investigating these algorithms in the next few pages, consider this question: Which one is the fastest?

### Find, Remove, Find

Here is the algorithm again, rewritten with one instruction per line and explicitly discussing the parameter *L*:

```
def find_two_smallest(L):
    """(list of float) --> tuple of (int, int)

    Return a tuple of the indices of the two smallest values in list L.

    Examples:
    find_two_smallest([100, 333, 477, 479, 357, 222, 33, 252, 324, 475])
    (6, 7)
    """

    # Find the index of the minimum item in L
    # Remove that item from the list
    # Find the index of the new minimum item in the list
    # Put the smallest item back in the list
    # If necessary, adjust the second index
    # Return the two indices
```

In addition to the first step, find the index of the minimum item in L, or return the output produced by calling `index(L)`, and test that there are no methods that do exactly that. We'll refine it:

```
def find two smallest():
    """(see above)

    # Get the minimum item in L           -- Pyth. func. to use
    # Find the index of that minimum item -- Pyth. func. to use
    # Remove that item from the list
    # Find the index of the new minimum item in the list
    # Put the smallest item back in the list
    # If necessary, adjust the second index
    # Return the two indices
```

These first two statements match Python functions and methods: `min` does the first, and `list.index` does the second. (There are other ways; for example, we could have written a loop to do the search.)

We see that `list.index` does the third, and the refinement of "Find the index of the new minimum item in the list" is also straightforward.

Note that we've left some of our English statements as comments, which makes it easier to understand why the Python code does what it does:

```
def find two smallest():
    """(see above)

    # Find the index of the minimum item in the list
    minIndex = min(L)
    minI = L.index(minIndex)
    L.remove(minIndex)

    # Find the index of the new minimum
    next_smallest = min(L)
    min2 = L.index(next_smallest)

    # Put the smallest item back in the list
    # If necessary, adjust the second index
    # Return the two indices
```

Since we removed the smallest item, we need to put it back, also. When we remove a value, the position of the following values will change by one.

So, since `min2` has been removed, if we want to get the indices of the two largest values in the original list, we might need to add 1 to `min2`:

```

def find_two_smallest(L):
    """See above.

    # Find the index of the minimum and remove that item
    smallest = min(L)
    minI = L.index(smallest)
    L.remove(smallest)

    # Find the index of the new minimum
    next_smallest = min(L)
    min2 = L.index(next_smallest)

    # Put smallest back into L
    # Put next_smallest at the position of the previous
    # smallest since we've just taken it
    L[minI] = smallest
    L[min2] = next_smallest

```

That's enough rambling (hopefully) so let's roll in Python:

```

def find_two_smallest():
    """List of float) --> tuple of (int, int)

    Return a tuple of the indices of the two smallest values in list L.

    See: find_two_smallest([0.0, 5.5, 4.5, 3.5, 2.5, 3.0, 2.0, 1.5, 4.5, 5.0])
    (6, 7)
    """

    # Find the index of the minimum and remove that item
    smallest = min(L)
    minI = L.index(smallest)
    L.remove(smallest)

    # Find the index of the new minimum
    next_smallest = min(L)
    min2 = L.index(next_smallest)

    # Put smallest back into L
    # Insert minI at next_smallest
    L[minI] = smallest

    # Put next_smallest in the position of the previous
    # smallest since we've just taken it
    if minI < min2:
        min2 += 1

    return (minI, min2)

```

That seems like a lot of work, and I am. Even if you do right by code, you'll be thinking through all those steps. But by writing them down first, you have a much greater chance of getting it right with a minimum amount of work.

## Sort, Identify Minimums, Get Indices

Here is the second algorithm, written with one instruction per line.

```
def find_two_smallest(L):
    """List of float --> tuple of (int, int)

    Return a tuple of the indices of the two smallest values in List L.

    >>> find_two_smallest([600, 321, 127, 128, 327, 323, 66, 325, 324])
    (6, 7)
```

*Notes:*

- Sort a copy of L
- Get the two smallest numbers
- Find their indices in the original List L
- Return the two indices

This looks straightforward; we can use built-in function `sorted`, which returns a copy of the list with the items in order from smallest to largest. We could have used method `sort` on `L`, but that breaks a fundamental rule: never mutate the contents of parameters unless the documentation says so.

```
def find_two_smallest(L):
    """List of float --> tuple of (int, int)

    Get a sorted copy of the list so that the smallest values are in the
    # front.
    temp_list = sorted(L)
    smallest = temp_list[0]
    next_smallest = temp_list[1]

    # Find their indices in the original list.
    # Return the two indices.
```

Now we can find the indices and return them the same way we did in `find_largest_index`.

```
def find_two_smallest(L):
    """List of float --> tuple of (int, int)

    Return a tuple of the indices of the two smallest values in List L.

    >>> find_two_smallest([600, 321, 127, 128, 327, 323, 66, 325, 324])
    (6, 7)
```

*Notes:*

- Sort a sorted copy of the List so that the two smallest items are in the # front.
- `temp_list = sorted(L)`
- `smallest = temp_list[0]`

```

next_smallest = temp[1].get(1)

# sorted this list to see what's going on here
print(i, index(next_smallest))
print(j, index(next_smallest))

return iindex, jindex

```

## Walk Through the List

Our first algorithm starts the search process for the first two:

```

def find_two_smallest(L):
    """List of floats or ints of size >= 2"""

```

Return a tuple of the indices of the two smallest values in List L.

```

    i, j = find_two_smallest([1/3, -333, -444, -555, -222, -33, -222, -444, -111,
        111, 77])

```

- Examine each value in the List in order
- Keep track of the indices of the two smallest values found so far
- Update these values when a new smaller value is found
- Return the two indices

We'll move the second line before the first one because it describes the whole process. Let's take a step. Also, whenever we print out the index, we think of iteration; the third line is part of that iteration, so we'll ignore it.

```

def find_two_smallest(L):
    """List of floats or ints of size >= 2"""

```

- Keep track of the indices of the two smallest values found so far
- Examine each value in the list in order
- Update these values when a new smaller value is found
- Return the two indices

Every loop has three parts: an initialization section to set up the variables we'll need, a loop condition, and a loop body. Here, the initialization will set up *i* and *j*, which will be the indices of the smallest two items considered so far. A natural choice is to set them to the first two items of the list.

```

def find_two_smallest(L):
    """List of floats or ints of size >= 2"""

```

- Initialize *i* and *j* to the first two indices of the sorted list and their smallest values at the beginning of *L*
- Examine each value in the List in order
- Update these values when a new smaller value is found
- Return the two indices

We can turn that first function a couple lines of code into English version in no time:

```
def find_lowest(l):
    # see above
```

- Set min and max to the indices of the smallest and next-smallest values at the beginning of  $L$ .

```
if L[0] < L[1]:
    min, max = 0, 1
else:
    min, max = 1, 0
```

- Compute even values in the list in order.
- Update max value when a new smaller value is found.
- Return the two indices.

We have a couple of choices now. We can iterate with a `for` loop over the values, a `for` loop over the indices, or a `while` loop over the indices. Since we're trying to find *minmax* and we want to keep all of the items in the list, we'll use a `for` loop over the indices—and we'll start at index 2 because we've examined the first two values already. At the same time, we'll refine the statement in the body of the loop to mention `min` and `max`:

```
def find_lowest(l):
    # see above
```

- Set min and max to the indices of the smallest and next-smallest values at the beginning of  $L$ .

```
if L[0] < L[1]:
    min, max = 0, 1
else:
    min, max = 1, 0
```

- Compute even values in the list in order.
- for i in range(2, len(l)):
 • Update max value when a new smaller value is found.
 • Return the two indices.

Now for the body of the loop. We'll pick apart "update max and/or min2 when a new smaller value is found." There are three possibilities:

- If  $i$  is smaller than both `min` and `max`, then we have a new smallest item: `min` temporarily holds the current smallest, and `max` temporarily holds the third smallest. We need to update both of them.
- If  $i$  is larger than `min` and smaller than `max`, we have a new second smallest.
- If  $i$  is larger than both, we skip it.

```

def find_two_smallest(L):
    """Return a tuple of the indices of the smallest and next-smallest
    values in the descending list L.
    If L[0] < L[1]:
        min1, min2 = 0, 1
    else:
        min1, min2 = 1, 0

    # Compare each value in the list in order
    for i in range(2, len(L)):
        if L[i] < min1 or L[i] < min2:
            if L[i] < min1:
                min2 = min1
                min1 = i
            else:
                min2 = i
    return (min1, min2)

```

All of these are easily translated to Python; in fact, we don't even need code for the "longer than both" case:

```

def find_two_smallest(L):
    """Return a tuple of the indices of the smallest and next-smallest values in list L.
    If L[0] < L[1]:
        min1, min2 = 0, 1
    else:
        min1, min2 = 1, 0

    # Compare each value in the list in order
    for i in range(2, len(L)):
        if L[i] < min1 or L[i] < min2:
            if L[i] < min1:
                min2 = min1
                min1 = i
            else:
                min2 = i
    return (min1, min2)

```

```

    if new second smallest >
        min1 &gt; min2
            min1 = 1
    else
        min2 = 1

```

```
return (min1, min2)
```

## 12.2 Timing the Functions

Programs are evaluated not only by how fast they run and how much memory they use. These two measures—time and space—are fundamental to the theoretical study of algorithms. They are also pretty important from a pragmatic point of view. Fast programs are more useful than slow ones, and programs that need more memory than what your computer has could potentially crash at all.

This section introduces one way to time how long code takes to run. You'll see how to run the three functions we developed to find the two lowest values in a list on 1,400 randomly meeting kangaroos in Darwin, Australia, from 1982 to 1996.<sup>1</sup>

Module `time` contains functions related to time. One of these functions is `perf_counter`, which returns a time in seconds. We can call it before and after the code we want to time and take the difference to find out how many seconds have elapsed. We multiply by 1000 in order to convert from seconds to milliseconds:

```

import time

t1 = time.perf_counter()

# Code to time goes here

t2 = time.perf_counter()
print('The code took {:.3f} ms'.format((t2 - t1) * 1000))

```

We'll want to time all three of our `find_two_smallest` functions. Rather than copying and pasting the timing over three times, we'll write a function that takes another function as a parameter as well as the list to search in. This timing function will return how many milliseconds it takes to execute a call on the function. After the timing function is the main program that reads the file of kangaroo pressings and then calls the timing function with each of the find two smallest functions.

<sup>1</sup> See <http://kangaroo-data.s3.amazonaws.com/>.

```

import time
import find_remove_find
import sort_then_find
import walk_through

def time_find_two_smallest(find_func, list):
    """Function. (list) -> float

    Return how many seconds find_func takes to execute.

    """
    t1 = time.perf_counter()
    find_func(list)
    t2 = time.perf_counter()
    return t2 - t1 * 1000

if __name__ == '__main__':
    # Read the file containing problem
    raw_lines = []
    raw_lines_file = open('1000000.txt', 'r')
    for line in raw_lines_file:
        raw_lines.append(line.strip())
    raw_lines_file.close()

    # Time each of the approaches
    find_remove_find_time = time_find_two_smallest(
        find_remove_find.find_two_smallest, raw_lines)

    sort_then_find_time = time_find_two_smallest(
        sort_then_find.find_two_smallest, raw_lines)

    walk_through_time = time_find_two_smallest(
        walk_through.find_two_smallest, raw_lines)

    print('Find remove find took {} millis.'.format(find_remove_find_time))
    print('Sort then find took {} millis.'.format(sort_then_find_time))
    print('Walk through the list took {} millis.'.format(walk_through_time))

```

The execution times were as follows:

| Algorithm             | Running Time (ms) |
|-----------------------|-------------------|
| Find, remove, find    | 0.06ms            |
| Sort, identify, index | 0.36ms            |
| Walk through the list | 0.26ms            |

Note how small these times are. No human being can notice the difference between values that are less than a millisecond: if this code never has to

process lists with more than 1,400 values, we would be just fine in choosing an implementation based on simplicity or clarity rather than on speed.

But what if we wanted to process millions of values? Then, however, the performance of other two algorithms on 1,400 values, doesn't much does that tell us about how each will perform on data sets that are a thousand times larger? That will be covered in [Chapter 13, Searching and Sorting](#), on page 237.

### 12.3 All a Minimum, You Saw This

In this chapter, you learned the following:

- The most effective way to design algorithms is to use top-down design, in which problems break down into subproblems until the steps are small enough to be translated directly into a programming language.
- Almost all problems have more than one correct solution. Choosing between them often involves a trade-off between simplicity and performance.
- The performance of a program can be characterized by how much time and memory it uses. This can be determined experimentally by profiling its execution time, or by profile time as with function `perf_counter` from module `time`.

### 12.4 Exercises

Here are some exercises for you to try on your own. Solutions are available at <http://www.oreilly.com/radar/problems.html>.

- a. A DNA sequence is a string made up of the letters A, T, G, and C. To find the complement of a DNA sequence, As are replaced by Ts, Ts by Gs, Gs by Cs, and Cs by Gs. For example, the complement of ATTCGATGT is TTAACCGCTT.
  - i. Write an outline in English of the algorithm you would use to find the complement.
  - ii. Review your algorithm. Will any characters be changed to their complement and then changed back to their original value? If so, rewrite your outline. Hint: Convert one character at a time, rather than all at once. As, Ts, Gs, or Cs at once?
  - iii. Using the algorithm that you have developed, write a function named `complement` that takes a DNA sequence as an arg and returns the complement string.
- b. In this exercise, you'll create a function that finds the minimum or maximum value in a list, depending on the caller's request.

- a. Write a loop (including initialization) to find both the minimum value in a list and that value's index in our pass through the list.
  - b. Write a function named `min_index` that takes one parameter `list` and returns a tuple containing the minimum value in the list and that value's index in the list.
  - c. You might also want to find the maximum value and its index. Write a function named `max_index` that has two parameters `list` and `if_max`. If the Boolean parameter refers to `True`, the function returns a tuple containing the maximum and its index, and if it refers to `False`, it returns a tuple containing the maximum and its index.
3. In [The Beatles Years](#), on page 17%, you learned how to read some files from the Times Series library. In particular, you learned about the [Hubble Catalog](#), which describes the number of galaxies for the years produced from 1931 to 1949. This file contains one value per year per line.
- a. Write an outline in English of the algorithm you would use to read the values from this data set to compute the average number of galaxies observed per year.
  - b. Translate your algorithm into Python by writing a function named `average_galaxy` that takes a filename as a parameter and returns the average number of galaxies per year.
4. Write a set of `assert`s for the `find_min_index` function. Think about what kinds of data are interesting: long lists or short lists, and what order the items are in. Here is one list to test with: `[1, 2]`. What other interesting cases are there?
5. What happens if the functions to find the two smallest values in a list are passed a list of length one? What should happen, and why? How about length zero? Write new `assert`s to describe what happens.
6. This one is a fun challenge.

[Edsger Dijkstra](#) is known for his work on programming languages. He came up with a neat problem that he called the Dutch National Flag problem: given a list of strings, each of which is either red, green, or blue (each is repeated several times in the list), rearrange the list so that the strings are in this order of the Dutch national flag—all the red strings first, then all the green strings, then all the blue strings.

Write a function called `dutch_flag` that takes a list and solves this problem.

# Searching and Sorting

A huge part of computer science involves studying how to organize, store, and retrieve data. The amount of data is growing exponentially; according to IBM, 90 percent of the data in the world has been generated in the past two years.<sup>1</sup> There are many ways to organize and process data, and you need to develop an understanding of how to analyze how good your approach is. This chapter introduces you to some tools and concepts that you can use to tell whether a particular approach is faster or slower than another.

As you know, there are many solutions to each programming problem. If a problem involves a large amount of data, a slow algorithm will mean the problem can't be solved in a reasonable amount of time, even with an immensely powerful computer. This chapter includes several examples of both slower and faster algorithms. Try running them yourself; experiencing just how close (or fast) something is has a much more profound effect on your understanding than the data we include in this chapter.

Searching and sorting are fundamental parts of programming. There are several ways to do both of them. In this chapter, we will develop several algorithms for searching and sorting lists, and then we will use them to explore what it means for one algorithm to be faster than another. As a bonus, this approach will give you another set of examples of how there are many solutions to any problem, and that the approach you take to solving a problem will dictate which solution you end up with.

## 13.1 Searching a List

As you have already seen in [Table 10. List Methods](#), on page 141, Python lists have a method called `index` that searches for a particular item:

<sup>1</sup> [http://www-03.ibm.com/ibm/strategic/whitepaper/ibm\\_ibm\\_90percent.html](http://www-03.ibm.com/ibm/strategic/whitepaper/ibm_ibm_90percent.html)

**index(...)**

```
i : index position (start, [stop])) -> integer    return the index of value
```

Just as `for` loops start at the first item in the list and examine each item in turn, for values that will soon become clear, this technique is called **linear search**. “Linear search” is used to find an item in an unsorted list. If there are duplicate values, our algorithm will find the **leftmost** one:

```
>>> l = ['d', 'a', 'b', 'a']
>>> l.index('a')
1
```

We’re going to write several versions of linear search in order to demonstrate how to compare different algorithms that all solve the same problem.

After we do this analysis, we will see that we can search a sorted list much faster than we can search an unsorted list.

**An Overview of Linear Search**

Linear search starts at index 0 and looks at each item one by one. At each index, we ask this question: Is the value we are looking for at this current index? We’ll show three variations of this. All of them use a loop of some kind, and they are all implementations of `l[0]`.

```
def Linear_search(lst, value):
    """(list, object) -> int
    Return the index of the first occurrence of value in lst, or return
    -1 if value does not occur in lst.
```

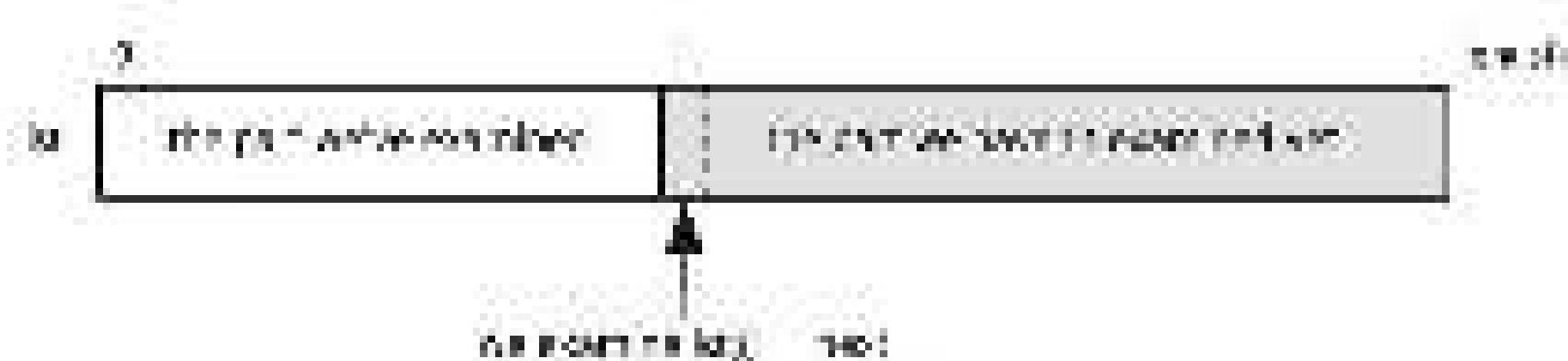
```
>>> l = [1, 2, 3, 4, 5]
>>> l[0]
1
>>> l[1]
2
>>> l[2]
3
>>> l[3]
4
>>> l[4]
5
```

• **return the index of the first occurrence of value in lst, or return -1 if value does not occur in lst.**

The algorithm in the function body describes what every variation will do to look for the value.

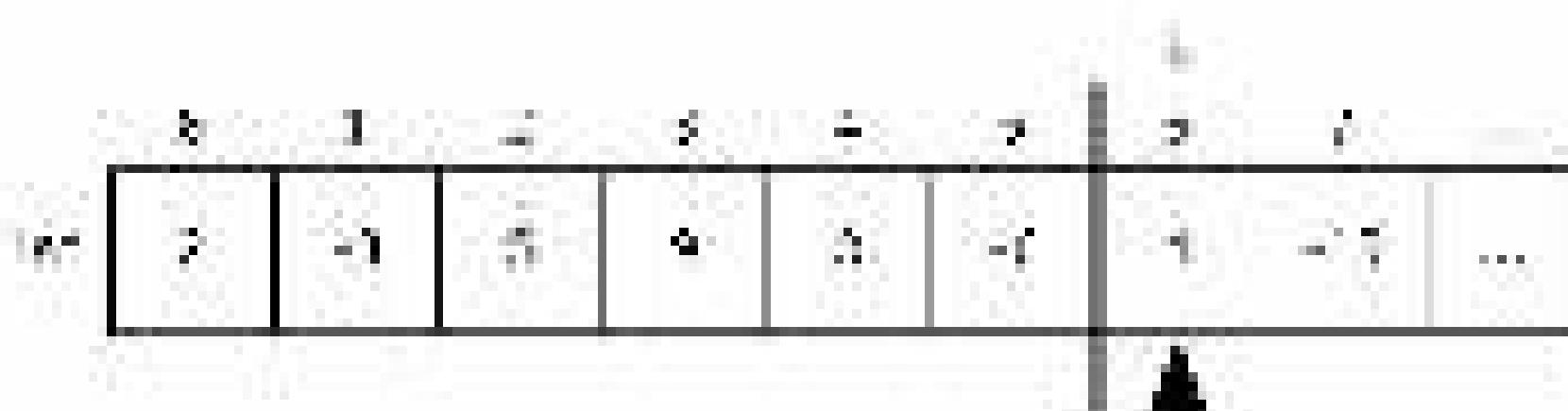
We’ve found it to be helpful to have a picture of how linear search works. (We will use pictures throughout this chapter for both searching and sorting.)

Because all of our iterations examine index 0 first, then index 1, then 2, and so on, that means that, partway through our searching process, we have this structure:



There is a part in the list that we've examined and another part that remains to be examined. We use variables to mark the current index.

Here's a concrete example of where we are searching for a value in a list that starts like this: [2, -8, 5, 3, -6, 4, 12, ...]. We don't know how long the list is, but let's say that after six iterations we have examined elements at indices 0, 1, 2, 3, 4, and 5. Index 6 is the index of the next item to examine:



That vertical line divides the list in two: the part we have examined and the part we haven't. Because we stop when we find the value, we know that the value isn't in the first part.



This picture is sometimes called an invariant of binary search. An invariant is something that remains unchanged throughout a process. But variables are changing—how can that picture be an invariant?

Here's a visual connection to the picture:

`list[0:4]` doesn't contain value, and 0 and 4 are `list[6]`.

This summary says that we know that word wasn't found before index (and that it is somewhere between 0 and the length of the list). If our code matches that word `wanted`, that word `wanted` is in inventory of the code, and so is the pasture `sheep`.

We can use invariants to come up with the initial values of our variables. For example, with linear search, at the very beginning the entire list is unknown—we haven't examined anything:



Variables `i` and `list` are initialized at the beginning, because that is the section with the valid code. `i` is 0, `list` is empty; further, `[, 0, 0]` is an empty list, which is exactly what we were according to the `wanted` section of the invariant. So the initial value of `i` should be 0 in all of our variants of linear search.

### The While Loop Version of Linear Search

Let's develop our first version of linear search. We need to refine our comments to get them closer to Python:

`Examine every index i in list, starting at index 0`

`If list[i] is the value we are looking for, stop executing`

Here's a refinement:

`i = 0 # The index of the next item in list to examine`

`While the unknown section isn't empty, and list[i] is not the value we are looking for`

`add 1 to i`

That's easier to translate. The unknown section is empty when `i > len(list)`, so it won't simply add `1` to `i` (it will). Here is the code:

```
def linear_search(list, value):
    i = 0
    if i < len(list):
```

`Return the index of the first occurrence of value in list, or return`  
 `-1 if value is not in list`

```
    else:
        return -1
    i += 1
    if i >= len(list):
```

`return -1`

```
(defun linear-search (l v)
  (loop (unless (or (null l) (= (car l) v))
    (setf l (cdr l))))
```

$i = 0$  # The index of the next item in  $l$  to examine.

# Keep going until we reach the end of  $l$  or we find our target value.

while  $i < \text{length}(l)$  and ( $\text{last}(l) \neq \text{value}$ )

$i = i + 1$

# If we fall off the end of the loop, we didn't find  $v$ .

if  $i = \text{length}(l)$

  return  $i$

else:

  return  $i$

This version uses `unless` to take the current index and move on through the values in  $l$ , skipping the initial two iterations where we have no initial values to examine. When we find the value we are looking for,

The first check in the loop condition,  $i < \text{length}(l)$ , makes sure that we still have values to look at. If we were to omit that check, then if  $i$  were  $\text{length}(l)$ , we would end up trying to access  $l[i](i)(i)$ . This would result in an `indexError`.

The second check,  $(\text{last}(l) = \text{value})$ , causes the loop to exit when we find  $v$ . The loop body guarantees to exit the loop when we haven't reached the end of  $l$ , and when  $\text{last}(l)$  isn't the value we are looking for.

At the end, we return  $i$  or  $>$ , which is either the index of the  $l[i]$  the `last` loop that evaluated to  $v$  or  $>$  if  $v$  wasn't in  $l$ .

### The For Loop Version of Linear Search

The first version evaluates two Boolean subexpressions each time through the loop. But the first clause,  $i = \text{length}(l)$ , is almost unnecessary. It evaluates to  $\text{true}$  almost every time through the loop, so the only effect it has is to make sure we don't attempt to index past the end of the list. We can instead exit the function as soon as we find the value:

$i = 0$  # The index of the next item in  $l$  to examine

for each index  $i$  in  $l$ :

  if  $\text{last}(l) = \text{value}$  we are looking for  
    return  $i$

If we get here,  $value$  was not in  $l$ , so we return `>`.

In this version, we use Python's `for` loop to iterate through each index.

```
def linear_search(lst, value):
    """(list, object) -> int

    Exactly the same code as before, but ...
    """
    for i in range(len(lst)):
        if lst[i] == value:
            return i
    return -1
```

With this version, we no longer need the first check because the `for` loop checks each index of the list for us. This `for` loop iteration is significantly faster than our first version; we'll see this a bit more much later.

### Sentinel Search

The last linear search we will study is called sentinel search. In sentinel is a guard whose job it is to stand *watch*. Remember that one problem with the while-loop linear search is that we check  $i = \text{len}(x)$  every time through the loop even though  $i$  can never evaluate to false except when  $i = \text{len}(x)$ . So we'll play a trick: we'll add  $x[-1]$  to the end of  $x$  before we search. That way we're guaranteed to find it! We also need to remove it before the function calls so that the user knows which  $x$  to use when he or she has called this function:

Set up the sentinel: append `value` to the end of `lst`.

```
i = 0 # The index of the next item in lst to examine
```

While `lst[i]` isn't the value we are looking for:

Add 1 to `i`.

Remove the sentinel:

```
return i
```

This translation isn't in Python:

```
def linear_search(lst, value):
    """(list, object) -> int
```

Exactly the same code as before ...

Add the sentinel:

```
lst.append(value)
```

## Linear search

```

1. If  $v = \text{first element}$  return index.
2. If  $v < \text{first element}$  return -1.
3. If  $v > \text{last element}$  return -1.

4. Remove the first item.
list.pop(0)

5. If we reached the end of the list, or didn't find  $v$ :
if  $i == \text{length}$ :
    return -1
else:
    return  $i$ .

```

All three of our linear search functions are correct. Which one you prefer is largely a matter of taste: some programmers like returning in the middle of a loop, so they won't have the second version. Others dislike modifying parameters in any way, so they won't like the third version. Still others will dislike that extra check that happens in the first version.

### Timing the searches

Here is a program that we used to time the three searches on a list with about ten million values:

```

import time
import linear_search
import linear_search_2
import linear_search_3

def time_itisearch(L, v):
    """function: object, float --> number
    Time how long it takes to run function search to find
    value v in list L.
    """
    t1 = time.perf_counter()
    resultL = v
    t2 = time.perf_counter()
    return t2 - t1 * 1000.0

def print_time(v, L):
    """object, list --> NoneType
    Print the number of milliseconds it takes for linear_search(v, L)
    to run for list index, the middle 7linear_search, etc for linear
    linear_search, and standard search.
    """
    print("Time for linear search: ", time_itisearch(L, v))
    print("Time for standard search: ", time_itisearch_3(L, v))
    print("Time for middle 7linear search: ", time_itisearch_2(L, v))
    print("Time for linear search: ", time_itisearch(L, v))

```

```

    # Get that index's running time.
    t1 = time.perf_counter()
    index = L[i]
    t2 = time.perf_counter()
    index_time = (t2 - t1) * 1000.0

    # Get the other three running times.
    while_time = time_it(linear_search_1, linear_search_1, i)
    for_time = time_it(linear_search_2, linear_search_2, i)
    sentinel_time = time_it(linear_search_3, linear_search_3, i)

    print(f'{i}: {t1:.4f} {t2:.4f} {index_time:.4f} {while_time:.4f} {for_time:.4f} {sentinel_time:.4f}')

```

L = list(range(1000000)) # A list 1 million items with successive results

```

print_time(0, L) # How fast is it to search the list beginning?
print_time(500000, L) # How fast is it to search near the middle?
print_time(999999, L) # How fast is it to search near the end?

```

This program makes use of function `perf_counter()` built into module `time`. Function `time_it()` will call whichever search function is given in `i` and return how long that search took. Function `print_time()` calls `time_it()` with the various linear search functions we have been exploring and prints their search times.

## Linear Search Running Time

The running times of the first linear searches with that of Python's `list.index` are compared in Table 17, *Running Times for Linear Search (in milliseconds)*. This comparison used a list of 1,000,000 items and three test cases: an item near the front, an item roughly in the middle, and the last item. Except for the first case, where the speeds differ by very little, our simple linear search takes about thirteen times as long as the one built into Python, and the `for` loop search and `sentinel` search take about five and seven times as long, respectively.

| Case   | while | for  | sentinel | <code>list.index</code> |
|--------|-------|------|----------|-------------------------|
| First  | 0.01  | 0.01 | 0.01     | 0.01                    |
| Middle | 1201  | 315  | 687      | 106                     |
| Last   | 28735 | 1088 | 1394     | 212                     |

Table 17—Running Times for Linear Search (in milliseconds)

What is more interesting is the way the running times of these functions increase with the number of items they have to examine. Roughly speaking, when they have to look through twice as much data, every one of them takes

twice as long). This is reasonable because, in sorting a list, adding 1 to an integer, and evaluating the loop control expression, require the computer to do a fixed amount of work. Doubling the number of times the loop has to be executed therefore doubles the total number of operations, which in turn would double the total running time. This is why this kind of search is called linear: the time to do it grows linearly with the amount of data being processed.

## 13.2 Binary Search

Is there a faster way to find values than by doing a linear search? The answer is yes, we can do much better, provided the list is sorted. To understand how, think about finding a name in a phone book. You open the book in the middle, glance at a name on the page, and immediately know which half to look in next. After checking only two names, you have eliminated  $\frac{1}{2}$  of the numbers in the phone book. Even in a huge city like Toronto, whose phone book has hundreds of thousands of entries, finding a name takes only a few steps.

This technique is called **binary search**, because each step divides the remaining data into two equal parts and eliminates one of the two halves. To figure out how fast it is, think about how big a set can be searched in a total number of steps. One step divides two values, two steps divide four, three steps divide  $2^3 = 8$ , four divide  $2^4 = 16$ , and so on. Thinking this around,  $N$  values can be searched in roughly  $\log_2 N$  steps.

More exactly,  $N$  values can be searched in  $\lceil \text{ceil}(2 \log_2 N) \rceil$  steps, where  $\text{ceil}(x)$  is the ceiling function that rounds a value up to the nearest integer. As shown in Table 18, *Logarithmic Growth*, this increases much less quickly than the time needed for linear search.

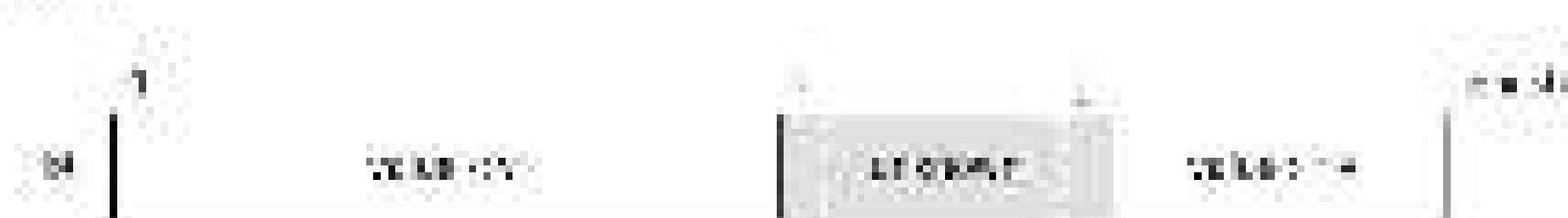
| Searching $N$ items | Worst Case—Linear Search | Worst Case—Binary Search |
|---------------------|--------------------------|--------------------------|
| 1 item              | 1 step                   | 1 step                   |
| 1,000 items         | 1,000 steps              | 10 steps                 |
| 10,000 items        | 10,000 steps             | 14 steps                 |
| 100,000 items       | 100,000 steps            | 17 steps                 |
| 1,000,000 items     | 1,000,000 steps          | 20 steps                 |
| 10,000,000 items    | 10,000,000 steps         | 24 steps                 |

Table 18—Logarithmic Growth

The key to binary search is to keep track of three parts of the list: the left part, which contains values that are smaller than the value we are searching for; the right part, which contains values that are equal to or larger than the

value we are searching for; and the middle part, which contains values that we haven't yet examined—the unknown section. If there are duplicate values, we want to return the index of the *leftmost* one (as we did with linear search), where *leftmost* means the “*equal-to*” sticker is stuck on the left.)

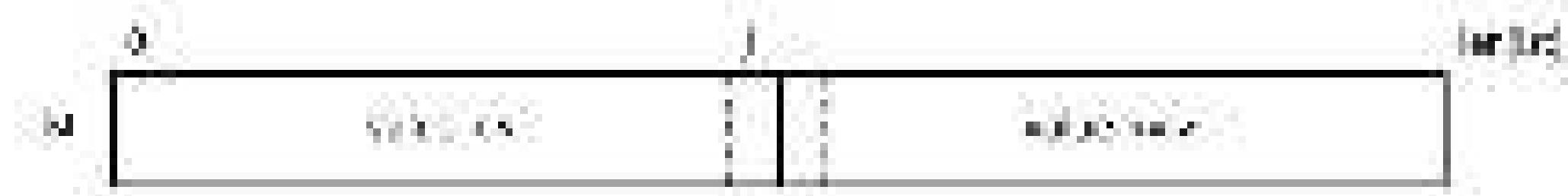
We'll use two variables to keep track of the boundaries. I will mark the *index* of the first unknown value, and J will mark the *index* of the last unknown value:



At the beginning of the algorithm, the unknown section makes up the entire list, so we will set  $i = 0$  and  $j = \text{length of the list} - 1$ .



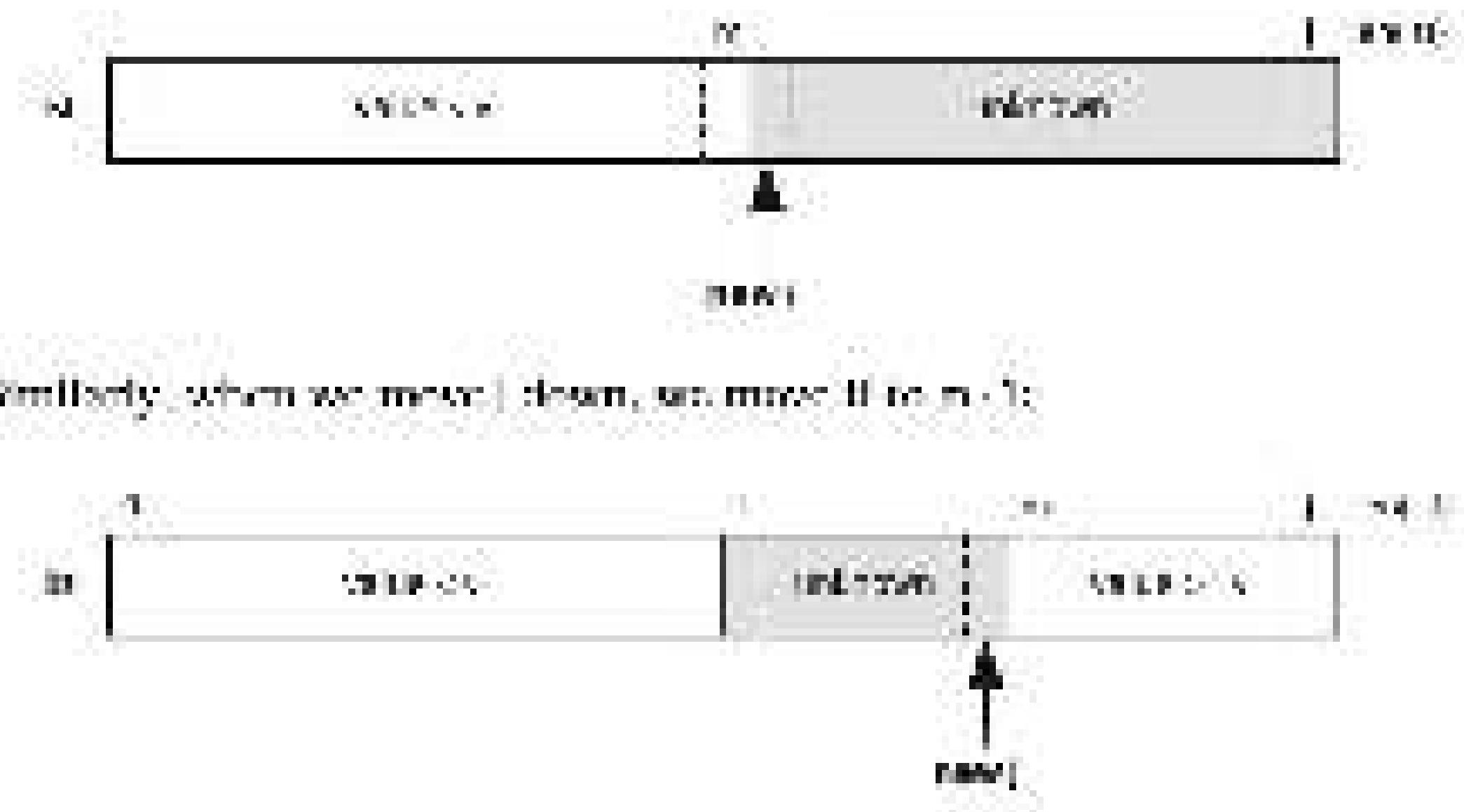
We are done when that *unknown* section is empty—when we've examined every item in it, *i.e.*, this happens when  $i = j + 1$ —when the *values* array (when  $i = j$ , there's still one item left in the unknown section). Here's a picture of what the values are when the *Unknown* section is empty:



To make progress, we will set either  $i$  or  $j$  to the *middle* of the range between them, to the full *index* at which  $i < j + 1$ . (Please note the use of integer division: we are calculating an index, so we need an integer.)

Think for a moment about the value of  $i$ . If it is less than  $s$ , we need to move  $i$  up, while if it is greater than  $s$ , we should move  $j$  down. But where exactly do we move them?

When we move  $i$  up, we don't want to set it to the *unknown* section, because  $i$  isn't included in the range; instead, we set it to one past the middle, in other words, to  $i + 1$ .



The completed function is as follows:

```
def binary_search(L, x):
    """(list, object) -> int

    Return low index of the first occurrence of x in L or return
    len(L) if x is not found.

    See binary_search([4, 5, 6, 7, 8, 9, 10], 7).
    """
    low = 0
    high = len(L) - 1
    while low <= high:
        mid = (low + high) // 2
        if L[mid] == x:
            return mid
        elif L[mid] < x:
            low = mid + 1
        else:
            high = mid - 1
    return -1
```

# More detail and right solution of the previous example.

L = [4, 5, 6, 7, 8, 9, 10]

j = len(L) - 1

```

while i < j + 1:
    m = (i + j) // 2
    if L[m] < v:
        i = m + 1
    else:
        j = m - 1

if 0 <= i < len(L) and L[i] == v:
    return i
else:
    return -1

```

If `_new_ == '_over_'`:

```

import doctest
doctest.testmod()

```

There are a lot of ways because the algorithm is quite complicated, and we wanted to test pretty thoroughly. Our basic error cases:

- The value is the first item.
- The value occurs twice. We want the index of the first one.
- The value is in the middle of the list.
- The value is the last item.
- The value is smaller than everything in the list.
- The value is larger than everything in the list.
- The value isn't in the list, but it is larger than some and smaller than others.
- The list has no items.
- The list has one item.

In Chapter 13, *Testing and Debugging*, on page 297, you'll learn a different testing framework that allows you to write tests in a separate Python file (it's making doctests shorter and easier to read; only a couple of examples are necessary), and you'll learn strategies for coming up with your own test cases.

## Binary Search Running Time

Binary search is much more complicated to write and understand than linear search. Is it ever enough to make it worth the effort? To find out, we can compare it to linear. As before, we search for the first, middle, and last items in a list with about ten million elements. (See [Table 19, Running Times for Binary Search](#), on page 249.)

The results are impressive. Binary search is up to several thousand times faster than its linear counterpart when searching ten million items. Most importantly, if we double the number of items, binary search takes only one more iteration, while the time for linear nearly doubles.

| Case   | Estimate | Binary search | Ratio |
|--------|----------|---------------|-------|
| First  | 0.007    | 0.02          | 0.38  |
| Middle | 105      | 0.02          | 5910  |
| Last   | 211      | 0.02 (Worst)  | 11051 |

Table 19—Running Times for Binary Search

Note also that although the time taken for linear search grows in step with the index of the item found, there is no such pattern for binary search. No matter where the item is, it takes the same number of steps.

### Built-In Binary Search

The Python standard library’s `sorted` module includes binary search functions that are slightly faster than our binary search. `binary_search_left` finds the index where an item should be inserted to a list to keep it in sorted order, assuming it is sorted to begin with. `binary_search_right` does the insertion.

The word `left` in the name signals that these functions find the leftmost (lowest index) position where they can do their job; the complementary functions `binary_right` and `nearest` find the rightmost.

There’s a problem, though: binary search assumes that the list is sorted, and sorting is time- and memory-intensive. When does it make sense to sort the list before we search?

### 13.3 Sorting

Now let’s look at a slightly harder problem. The following table<sup>2</sup> shows the number of acres lost to forest fires in Canada from 1918 to 1987. What were the worst years?

| 1918 | 1919 | 1920 | 1921  | 1922 | 1923 | 1924 | 1925 | 1926 | 1927 | 1928 |
|------|------|------|-------|------|------|------|------|------|------|------|
| 1342 | 6029 | 2570 | 2004  | 5404 | 1009 | 1473 | 656  | 2027 | 4271 | 112  |
| 3158 | 1112 | 2291 | 4562  | 1525 | 520  | 2406 | 712  | 1027 | 216  | 112  |
| 2161 | 2044 | 2277 | 104   | 893  | 1024 | 101  | 1077 | 2048 | 112  | 112  |
| 1560 | 1112 | 371  | 1613  | 561  | 151  | 2051 | 101  | 1144 | 2212 | 112  |
| 1047 | 2111 | 4108 | 1743  | 102  | 174  | 2601 | 104  | 2041 | 2231 | 112  |
| 502  | 1742 | 1562 | 10764 | 2618 | 3134 | 1419 | 1261 | 1232 | 1474 | 112  |

Table 20—Acres Lost to Forest Fires in Canada (in thousands), 1918–1987

2. <http://www.thewebarchive.org/wayback/20110614084044/http://www.kirchner.com/binary/binary.html>

One way to find out how much time was consumed in the `is_sorted` part is to sort the list and then take the last `MinValue`, as shown in the following code:

```
def find_largest(L):
    if len(L) == 1:
        return L[0]
    else:
        copy = sorted(L)
        return max(copy[-1], find_largest(copy[:-1]))
```

This algorithm is short, clean, and easy to understand, but it relies on a bit of black magic. How does function `sorted` (and also method `list.sort()`) work, any way? And how is this slow, though?

It turns out that many sorting algorithms have been developed over the years, each with its own strengths and weaknesses. Broadly speaking, they can be divided into two categories: those that are simple but inefficient and those that are efficient but harder to understand and implement. We'll examine two of the former kind. The rest rely on techniques that are more advanced; we'll show you some of those, too, when it's only material we can use.

Both of the simple sorting algorithms keep track of two sections in the list being sorted. The section at the front contains values that are now in sorted order; the section at the back contains values that have yet to be sorted. Here is the main part of the invariant that we will use for our two simple sorts:



One of the first algorithms has an additional property to be invariant: the items in the sorted section must be smaller than all the items in the unsorted section. Both of these sorting algorithms will walk their way through the list, making the sorted section one item larger on each iteration. We'll see that there are two ways to do this. Here is an outline for our code:

```
i = 0 # The index of the first element that is not yet sorted
while i < len(L):
    # Do something to incorporate L[i] into the sorted section
```

```
i = i + 1
```

Most Python programmers would probably write the loop header as `i = 0` rather than `for i in range(0, n)` explicitly in the body of the loop. We're doing this here to explicitly highlight [1] setting up the loop invariant and to show the invariant separately from the work this particular algorithm is doing. The “do something” part is where the two simple sorting algorithms will differ.

### Selection Sort

*Selection sort works by searching the unknown section for the smallest item and moving it to the front. Here's our algorithm:*

```

i = 0 # The index of the first unsorted item in list
# Initially is sorted and these others are unsorted items in list.
while i < len(L):
    # Find the index of the smallest item in L[i:]
    # Swap that smallest item with the item at index i
    i = i + 1

```

As you can probably guess from this description, selection sort works by repeatedly finding the next smallest item in the unsorted section and placing it at the end of the sorted section. That works because we are selecting the items in order. On the first iteration, `i = 0`, and `L[0]` is the entire list. That means that on the first iteration we select the smallest item and move it to the front. On the second iteration we select the second-smallest item and move it to the second spot, and so on. (See Figure 12. First few steps in selection sort on page 233.)

In a file named `sorts.py`, we have started writing a `selection_sort` function, partially in English, as shown in the following code:

```

def selection_sort(L):
    """Sorts list L in-place.
    Precondition: L is a list.
    Postcondition: The items in L are sorted in increasing order.
    """
    for i in range(0, len(L) - 1):
        # Find the index of the smallest item in L[i:]
        # Swap that smallest item with the item at index i
        i = 0
        for j in range(i+1, len(L)):
            if L[j] < L[i]:
                i = j

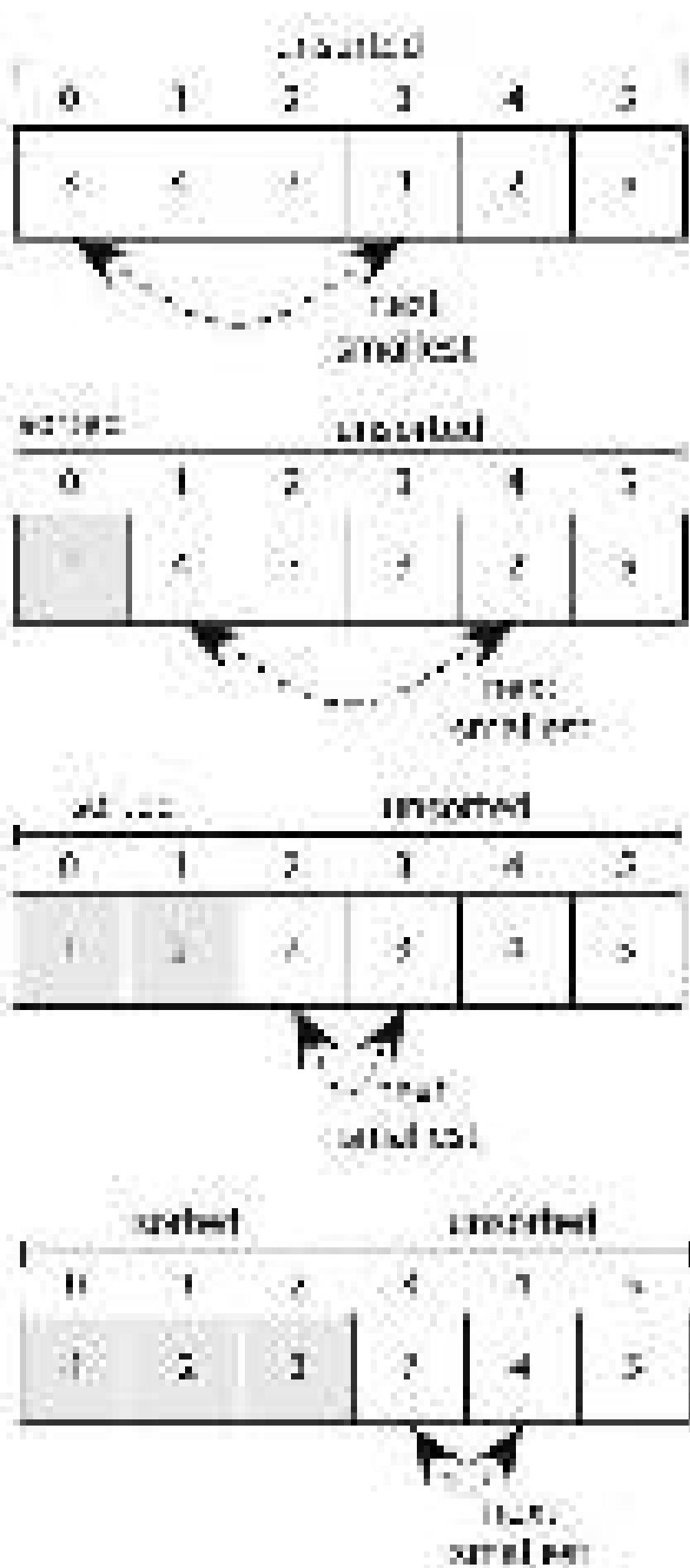
```

```

    i = 0
    while i < len(L):
        # Find the index of the smallest item in L[i:]
        # Swap that smallest item with the item at index i
        i = i + 1

```

We can reduce the second comment with a single line of code:

**Figure 2** First few steps in selection sort.

```
def selection_sort(L):
    """L: list of Comparable objects

    Reorder the items in L from smallest to largest.

    """
    for i in range(len(L)):
        min_index = i
        for j in range(i+1, len(L)):
            if L[j] < L[min_index]:
                min_index = j
        L[i], L[min_index] = L[min_index], L[i]
```

```

while i < len(l):
    # return index of the smallest item in l[i:]
    i[1] = find_min(l[i:][1])
    i = i + 1

```

Now all that's left is finding the index of the smallest item in  $l[i:]$ . This is complex enough that it's worth writing it in a function of its own:

```

def find_min(l):
    if len(l) == 1:
        return 0

    overall_min = l[0]
    overall_index = 0

    for i in range(1, len(l)):
        if l[i] < overall_min:
            overall_min = l[i]
            overall_index = i
    return overall_index

```

```

smallest = b # The index of the smallest so far.
a = b + 1
while a < len(l):
    if l[a] < Lsmallest:
        # We found a new item smaller than
        # smallest.
        smallest = a
    a = a + 1

```

```
return smallest
```

```

def selection_sort(l):
    if type(l) != list_type:
        raise TypeError("Expected the input to be a list-type")

    overall_min = l[0]
    overall_index = 0

    for i in range(1, len(l)):
        if l[i] < overall_min:
            overall_min = l[i]
            overall_index = i

```

```

    i = 0
    while i < len(l):
        smallest = find_min(l[i:])
        l[i], lsmallest = lsmallest, l[i]
        i = i + 1

```

function had to examine each item in  $L[i:]$ , keeping track of the index of the minimum item as far as variable  $smallest$ . Whenever it finds a smaller item, it updates  $smallest$  (because it is returning the index of the smallest value, it won't work if  $L[i:]$  is empty; hence the precondition).

This is complicated enough that a couple of doctests may not test enough. Here's a better set of cases for sorting:

- An empty list
- A list of length 1
- A list of length 2 (this is the sharpest case where  $smallest$  matters)
- An already-sorted list
- A list with all the same values
- A list with duplicates

Here are our expanded doctests:

```
def selection_sort(L):
    """Sorts list L using selection sort algorithm.

    Examples:
        selection_sort([4, 2, 3, 1]) == [1, 2, 3, 4]
        selection_sort([]) == []
        selection_sort([1]) == [1]
        selection_sort([-2, -1, 0, 1, 2]) == [-2, -1, 0, 1, 2]
        selection_sort([1, 0, 2, 3, 4]) == [0, 1, 2, 3, 4]
        selection_sort([4, 3, 2, 1]) == [1, 2, 3, 4]
        selection_sort([2, 3, 4, 1]) == [1, 2, 3, 4]
        selection_sort([3, 4, 1, 2]) == [1, 2, 3, 4]
        selection_sort([4, 3, 2, 1, 5]) == [1, 2, 3, 4, 5]
        selection_sort([5, 4, 3, 2, 1]) == [1, 2, 3, 4, 5]
    """
    for i in range(len(L) - 1):
        # Assume L[i:] is sorted
        smallest = i
        for j in range(i + 1, len(L)):
            if L[j] < L[smallest]:
                smallest = j
        L[i], L[smallest] = L[smallest], L[i]
```

```
[1, 2, 3, 4, 5, 6, 7, 8]
```

```
i = 0
while i < len(L):
    smallest = find_min(L, i)
    L[i], L[smallest] = L[smallest], L[i]
    i = i + 1
```

As with binary search, the docstring is asking that, as documentation for the function, it *checks* rather than helps identify bugs. Again, we'll see how to fix this in [Chapter 15: Testing and Debugging myjpy 207](#).

## Inception Sort

Like selection sort, insertion sort keeps a sorted section at the beginning of the list. Rather than swap all of the unsorted section for the next smallest item, though, it takes the next item from the *unsorted section* — the one at index  $i+1$  — inserts it where it belongs in the *sorted section*, increasing the size of the sorted section by one.

```
L = [8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
```

```
while i < len(L):
    # Move the current value to its correct position (L[i+1:i+2])
    i = i + 1
```

The reason why we use  $i+1:i+2$  is because the item at index  $i$  may be larger than everything in the sorted section, and if that is the case then the current item won't move.

To confirm, this is an follows (see [this notebook](#) as well):

```
def insertion_sort(L):
    """List --> NoneType
```

Reorder the items in  $L$  from smallest to largest.

```
new_L = [2, 4, 7, 1, 3, 5]
insertion_sort(new_L)
new_L
# [1, 2, 3, 4, 5, 7]
```

$i = 0$

```
while i < len(L):
    # Insert L[i] where it belongs in L[:i].
    i = i + 1
```

This is exactly the same approach as for selection sort; the difference is in the comment in the loop. Like we did with selection sort, we'll write a helper function to do that work:

```
def insert(L, b):
    """(list, int) -> NoneType
```

*Precondition: L[0:i] is already sorted.*  
*Insert L[i] where it belongs in L[0:i + 1].*

```
(cc, L = [2, 4, -1, 5, 3, 2])
(cc, insert(L, 2))
(cc, L)
[-1, 2, 4, 5, 3, 2]
(cc, insert(L, 4))
(cc, L)
[-1, 2, 3, 4, 5, 2]
```

# First we're to insert 4 at its starting position from L[0:i + 1] to L[i:i + 1] and then i += 1.

```
i = b
while i <= n and L[i - 1] > L[i]:
    i += 1
```

# Now i is the index of inserting the following value to the right.  
value = L[b]
del L[b]
L.insert(b, value)

```
def insertion_sort(L):
    """(list) -> NoneType
```

*Reorder the items in L from smallest to largest.*

```
(cc, L = [2, 4, 5, -1, 3, 2])
(cc, insertion_sort(L))
(cc, L)
[-1, 2, 3, 4, 5, 2]
```

$L_1 = \emptyset$

```
while L != []:
    insert(L, i)
    i = i + 1
```

How does *insert* work? It works by shifting on when  $L[i]$  belongs and then moving  $L$ . When does it happen? It happens after every value from  $L$  equal to it and before every value that is greater than it. We need the check  $i != i$  in

case L[i] is smaller than every value in Q[1:i], which will place the current item at the beginning of the list. (See Figure 1.8. Plus few steps in selection sort. This process till the last we write sorted for selection sort.)

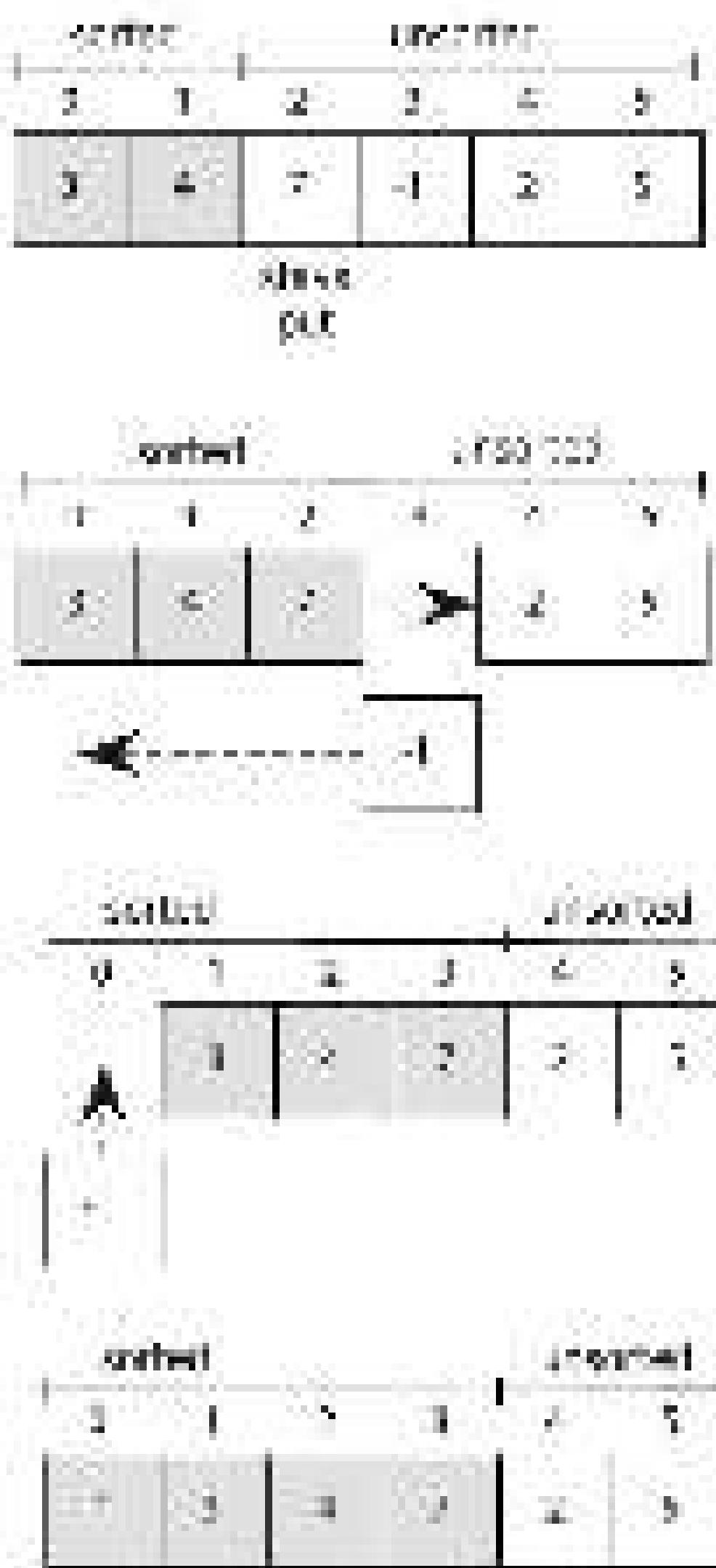


Figure 1.8—First few steps in selection sort

## Performance

We now have ten sorting algorithms. When should we use? Because each one has its own different implementation, it's reasonable to choose based on how fast they run.

the built-in `min()`. In writing a program to compare their running times, along with that for `sort()`:

```
import time
import random
from math import log10, floor
from math import insertion_sort

def build_list():
    """List --> NoneType

    Call this and ... we need our own function to do this so that we can
    treat it as we treat our own sorts.

    """
    L = []
    return L

def print_time(L):
    """List --> NoneType

    Count the number of comparisons it takes for selection sort. Insertion
    sort, and list.sort() to run.

    """
    print("Initial list: ", L)

    # print_time(L) -> NoneType
    for i in insertion_sort_selection_sort_time(L):
        if i[0] in [len(L), len(L)-1] and i[1] == 10000:
            continue

        L_copy = L[:]
        T1 = time.perf_counter()
        timeit(L_copy)
        T2 = time.perf_counter()
        print(f'{T2-T1:.10f}, {len(L)}, {i[0]}')

    print("\nPrint a random\n")

    for list_size in [10, 100, 200, 300, 400, 500, 1000]:
        L = list(range(list_size))
        random.shuffle(L)
        print_time(L)
```

The results are shown in Table 21. Running Times for Selection, Insertion, and `list.sort()` in milliseconds, no jury 200.

Something is very clearly wrong, because our sorting functions are thousands of times slower than the built-in `sort()`. What's more, the time required by our `sort()`s is growing faster than the size of the data. On a thousand items, for example, `selection_sort()` takes about 0.05 milliseconds per item, but on ten

| List length | Selection sort | Inserion sort | Is sort |
|-------------|----------------|---------------|---------|
| 1000        | 168            | 64            | 0.9     |
| 2000        | 593            | 208           | 0.6     |
| 3000        | 1017           | 364           | 0.9     |
| 4000        | 2007           | 1025          | 1.3     |
| 5000        | 3009           | 1066          | 1.6     |
| 10000       | 14574          | 6550          | 3.5     |

**Table 21—Running Times for Selection, Insertion, and Is sort (in milliseconds)**

thousand items, it needs about 1.45 milliseconds per item—slightly more than a tenth of insertion sort. What is going on?

To answer this, we examine what happens in the inner loops of our two algorithms. On the first iteration of selection sort, the inner loop examines every element to find the smallest. On the second iteration, it looks at all but one; on the third, at least at all but two, and so on.

If there are  $N$  items in the list, then the number of iterations of the inner loop, in total, is roughly  $N - (N - 1) + (N - 2) + \dots + 1$ , or  $\frac{N^2 + N}{2}$ . In other words, the number of steps required to sort  $N$  items is roughly proportional to  $N^2 + N$ . For large values of  $N$  we can ignore the second term and say that the time needed by selection sort grows as the square of the number of values being sorted. And indeed, examining the timing data further shows that doubling the size of the list increases the running time by four.

The same analysis can be used for insertion sort, since it also examines one element on the first iteration, two on the second, and so on. (It just examines the already sorted values rather than the unsorted values.)

So why is insertion sort slightly faster? The reason is that, on average, only half of the values need to be examined in order to find the location in which to insert the new value. While with selection sort, every value in the unsorted sublist needs to be examined in order to select the smallest item, this is not necessarily so much faster.

### 13.4 More Efficient Sorting Algorithms

The analysis of selection and insertion sort begs the question: how can I sort faster and more efficiently? The answer is the same as it was for binary search: by taking advantage of the fact that some values are already sorted.

## A First Attempt

Consider the following function:

```
import bisect
```

```
def bin_sort(values):
```

```
    """List -> List
```

```
    Returns a sorted version of the values. (This does not mutate values.)
```

```
    See: L = [2, 4, 3, -1, 5]
```

```
    See: bin_sort(L)
```

```
    See: L
```

```
    [-1, 2, 3, 4, 5, 7]
```

```
    ...
```

```
    result = []
```

```
    for v in values:
```

```
        bisect.insort(result, v)
```

```
    return result
```

This uses `list.insert()` in Figure 2.1 where we get each value from the original list into a new list that is kept in sorted order. As we have already seen, doing this takes time proportional to  $\log_2 N$ , where  $N$  is the length of the list. Since  $N$  values have to be inserted, the overall running time ought to be  $N \log_2 N$ .

As shown in the following table, this grows much more slowly with the length of the list than  $N^2$ :

| $N$  | $N^2$     | $N \log_2 N$ |
|------|-----------|--------------|
| 10   | 100       | 3.32         |
| 100  | 10,000    | 6.64         |
| 1000 | 1,000,000 | 9.90         |

Table 2.2: Sorting Times

Unfortunately, there's a flaw in this analysis. It's correct to say that `list.insert()` costs only  $\log_2 N$  time in Figure 2.1 when we insert a value, but usually inserting it takes time as well. To create an empty slot in the list, we have to move all the values above that slot up one place. On average, this means moving half of the  $N/2$  values, so the cost of insertion is proportional to  $N$ . Since there are  $N$  values to insert, our total time is  $N(N + \log_2 N)$ . For large values of  $N$ , this is  $N^2$  again roughly proportional to  $N^2$ .

## 13.5 Mergesort: A Faster Sorting Algorithm

There are several well-known, fast sorting algorithms: bubble sort, quicksort, and mergesort are the ones you are most likely to encounter in a future CS course. Most of them involve techniques that we haven't taught you yet, but mergesort can be written in terms we've seen. Mergesort is built around the idea that taking two sorted lists and merging them is proportional to the number of items in both lists. The running time for mergesort is  $\Theta(n \log n)$ .

We'll start with very small lists and keep merging them until we have a single sorted list.

### Merging Two Sorted Lists

Given two sorted lists  $L_1$  and  $L_2$ , we can produce a new sorted list by scanning along  $L_1$  over  $L_2$  and comparing pairs of elements. (We'll see how to produce these two sorted lists from  $BL$ .)

Here is the code to merge:

```
def merge(L1, L2):
    """Merge sorted lists L1 and L2 into a new list and return that new list.

    Examples:
    merge([1, 3, 5], [2, 4, 6]) == [1, 2, 3, 4, 5, 6]
    merge([1, 2, 3], [4, 5, 6]) == [1, 2, 3, 4, 5, 6]
    """

    result = []
    i1 = 0
    i2 = 0

    # For each pair of items (L1[i1], L2[i2]), copy the smaller item and,
    # while i1 < len(L1) and i2 < len(L2):
    if L1[i1] < L2[i2]:
        result.append(L1[i1])
        i1 += 1
    else:
        result.append(L2[i2])
        i2 += 1

    # Return any leftover items from the two sorted lists.
    # Note that one of these will be empty because of the loop condition.
    result.extend(L1[i1:])
    result.extend(L2[i2:])
    return result
```

$i_1$  and  $i_2$  are the indices into  $L_1$  and  $L_2$ , respectively; on each iteration, we compare  $L_1[i_1]$  to  $L_2[i_2]$  and copy the smaller item to the resulting list. At the

end of the lists, we have run out of items in one of the two lists, and the two `extend` calls will append the rest of the items to the result.

## Mergesort

Here is the header for `mergesort`:

```
def mergesort(L):
    """Merge sort algorithm.
```

Consider this trace for a short sorted-together list:

```
>>> L = [1, 4, 2, 3, 5, 6]
>>> mergesort(L)
>>> L
[1, 2, 3, 4, 5, 6]
```

`Mergesort`'s function is going to do the bulk of the work. Here is the algorithm, which creates and keeps track of a list of lists:

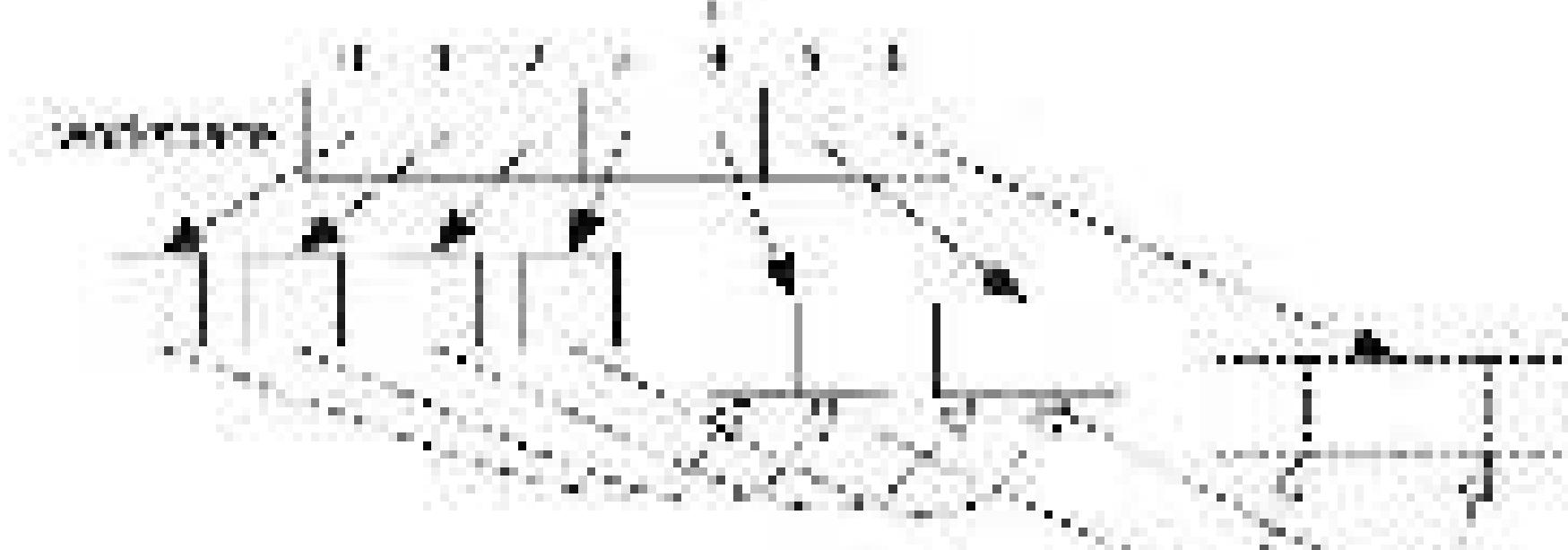
- Take `lls`, a list of lists, and make a list of one-item lists from it.
- As long as `lls` has two lists left to merge, merge them, and append the new list to the list of lists.

The first step is straightforward:

```
# Have a list of 2-item lists so that we can start merging.
nsubparts = 11
for i in range(len(LL)):
    nsubparts.append(LL[i:i+2])
```

The second step is trickier. How remove the two lists, then will turn into the same problem that we run into in `bin_search`? All the following lists will need to shift over, which takes time proportional to the number of lists.

Instead, we'll keep track of the index of the next two lists to merge. Initially, they will be at indices 0 and 1, and then 2 and 3, and so on:



Here is our **Merge** algorithm:

- Take list  $L$ , and make a list of one item (list) from it.
- Start index  $i$  set at 0.
- As long as there are two lists (at indices  $i$  and  $i+1$ ), merge them, append the new list to the list of lists, and increment  $i$  by 2.

With that, we can do **straight** insertion:

```
def merge(L):
    """(list) -> NoneType
```

Reorder the items in  $L$  from smallest to largest.

```
nn> L = [3, 4, 5, 1, 2, 6]
nn> merge(L)
nn> L
[1, 2, 3, 4, 5, 6]
```

```
# Merge two lists of items (as previously defined in mergeSort())
# Input: L1, L2
# Output: mergedL
for i in range(len(L1)):
    mergedL.append([L1[i], L2[i]])

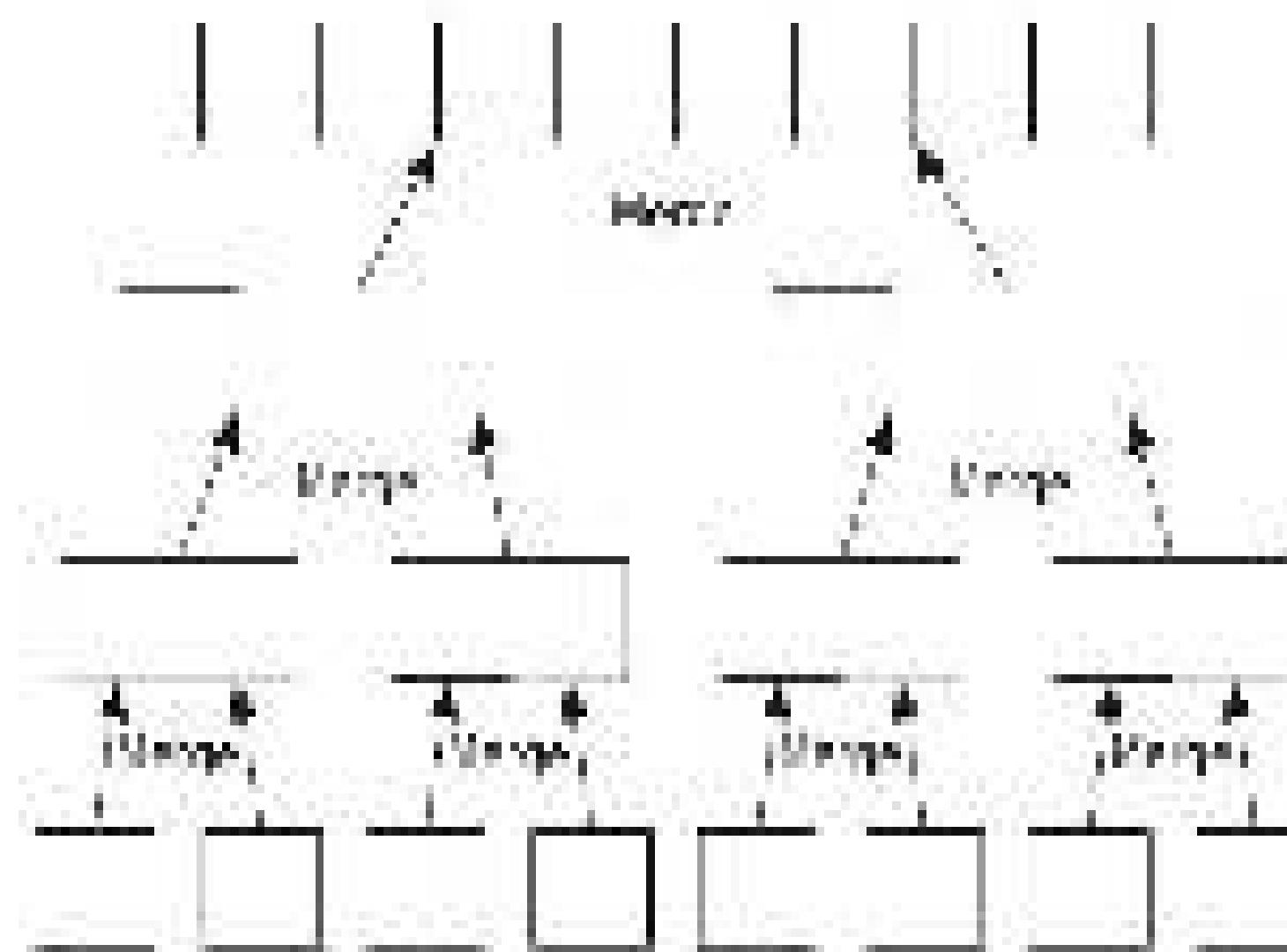
# Copy first two lists to memory, then merge them if len(L1) > len(L2)
i = 0
# as long as there are at least two more lists to merge, keep doing
while i < len(L1)-len(L2):
    L1 = noSpace[i]
    L2 = noSpace[i + 1]
    newList = merge(L1, L2)
    noSpace.append(newList)
    i += 2

# Copy the result over into L
if len(noSpace) != 0:
    L[:i] = noSpace[:i]; i
```

Notice that since we're always making new lists, we need to copy the last of the `noSpace` lists back into `L` as parameter!

## Mergesort Analysis

Mergesort is **linearithmic**,  $\Theta(N \log N)$ , where  $N$  is the number of items in  $L$ . The following diagram shows three-item lists being merged into two-item lists, then four-item lists, and so on until there is one  $N$ -item list. (See Section 13.5, [Mergesort: A Divide-and-Conquer Algorithm](#), on page 324.)



The first part of the function, creating the list of one-item lists, takes  $N$  items, one for each item.

The second loop, in which we continually merge lists, will take some time to analyze. We'll start with the very first iteration, in which we are merging two lists with about  $\frac{N}{2}$  items. As we've seen, function `merge` copies each element from the smaller exactly once, so with these two lists, this merge step takes roughly  $\frac{N}{2}$  steps.

On the previous iteration, there are two lists of size  $\frac{N}{4}$  to merge into one of the two lists of size  $\frac{N}{2}$ , and on the iteration before that there are four lists of size  $\frac{N}{8}$  to merge into the second list of size  $\frac{N}{4}$ . Each of these two merges takes roughly  $\frac{N}{8}$  steps, so the two together take roughly  $N$  steps total.

On the third iteration, there are four lists of size  $\frac{N}{16}$  to merge into the four lists of size  $\frac{N}{8}$ . Four merges of this size together also take roughly  $N$  steps.

We can calculate a list with  $N$  items a total of  $\log_2 N$  times using an analysis much like we used for binary search. Since at each level there are a total of  $N$  items to be merged, each of these  $\log_2 N$  levels takes roughly  $N$  steps. Hence `mergesort` takes time proportional to  $N \log_2 N$ .

That's an useful lot of code to sort a list! There are shorter and cleaner versions, but again, they rely on techniques that we haven't yet introduced.

Despite all the code and our somewhat modest approach, it creates a lot of `ArrayList` temporary lists just to be much, much faster than selection sort.

and insertion sort. Most importantly, it shows at the same time how the built-in sort() works.

| List Length | Selection Sort | Involution Sort | Merge Sort | Insertion Sort |
|-------------|----------------|-----------------|------------|----------------|
| 1000        | 145            | 64              | /          | 0.5            |
| 2000        | 588            | 128             | /          | 0.6            |
| 3000        | 1317           | 194             | 25         | 0.9            |
| 4000        | 2337           | 1666            | 52         | 1.3            |
| 5000        | 3699           | 1666            | 41         | 1.6            |
| 10000       | 14674          | 6050            | 88         | 3.0            |

Table 22—Running times for selection, involution, merge, and insertion (in milliseconds)

## 13.6 Sorting Out What You Learned

In this chapter, you learned the following:

- An *invariant* describes the data being used in a loop. The initial values for the variables used in the loop will establish the invariant, and the work done inside the loop will make progress toward the solution. When the loop terminates, the invariant is still true, but the solution will have been modified.
- Linear search is the simplest way to find a value in a list; but on average, the time required is directly proportional to the length of the list.
- Binary search is much faster—the average time is proportional to the logarithm of the list's length—but it works only if the list is in sorted order.
- Similarly, the average running time of simple sorting algorithms like selection sort is proportional to the square of the input size  $N$ , while the running time of more complex sorting algorithms grows as  $N \log N$ .
- Looking at how the running time of an algorithm grows as a function of the size of its inputs is the standard way to analyze and compare the algorithm's efficiency.
- Selection sort and insertion sort have almost the same invariant; the only difference is that with selection sort, the sorted section contains values that are smaller than all the values in the unsorted section. The two algorithms differ by how they make progress: selection sort selects the next-smallest item to put at the end of the sorted section, while insertion sort moves the next item into the sorted section.

## Big-Oh and all That

Our method of analyzing the performance of searching and sorting algorithms might seem like hand-waving. But there is, actually a well-developed mathematical theory called *Big-Oh* analysis that handles this. In the expression,  $O(n)$  = Big-Oh of  $n$ , the  $n$  being word count,  $O(n^2)$  = Big-Oh of  $n^2$ , etc. It's simplified theory by some constraints. In fact,  $O(n^2)$  is equivalent to  $\Theta(n^2)$  or  $\Omega(n^2)$ . The  $\Theta$  is based on the worst-case running time. Complexity refers to the average algorithmic complexity, such as those sorting functions that execute in  $n^2$  time and those that execute in  $n \log n$  time.

These distinctions have important practical implications. In particular, one of the biggest problems in the field of computer science today is whether the first few of algorithms taught in school still work as we know; perhaps being on the same or not. Almost certainly, with the growth of the web and its use has been widespread of applications of  $O(n^2)$  and  $O(n^3)$  time for the first course. And, if you want the largest security gains, then many of the algorithms used to encrypt data in banking and military applications you will see on the Web will be much more vulnerable to attack than expected.

## 13.7 Exercises

There are some exercises for you to try on your own. Solutions are available at <https://tinyurl.com/yd2qzg2j> under **Exercises**.

1. All three versions of linear search return `nil`. If I search all three to search from the end of the list instead of from the beginning. Make sure you test them.
2. For the new versions of linear search: there are duplicates values, which do they find?
3. **Binary search is significantly faster than linear search** but requires that the list be sorted. As you know, the running time for the best binary algorithm is on the order of  $N \log_2 N$  where  $N$  is the length of the list. If we search a lot of times on the same list of data, it makes sense to sort it once before doing the searching. Roughly how many times do we need to search in order to make sorting and then searching faster than using the `list` in `search`?
4. Given the unsorted list [2, 3, 4, 2, 3, 1, 2], show what the contents of the list would be after each iteration of the loop over `i` sorted using the following:
  - selection sort
  - bubble sort

6. Another sorting algorithm is bubble sort. Bubble sort involves keeping a sorted section at the end of the list. The list is traversed, pairs of elements are compared, and larger elements are swapped into the higher position. This is repeated until all elements are sorted.
- Using the English description of bubble sort, write an outline of the bubble sort algorithm in English.
  - Continue using English until you have a Python algorithm.
  - This is now a function called `bubble_sort()`.
  - Try it out on the test cases from exercise 5.
6. In the description of bubble sort in the previous exercise, the sorted section of the list was at the end of the list. In this exercise, bubble sort will maintain the sorted section at the beginning of the list. Make sure that you are still implementing bubble sort!
- Rewrite the English description of bubble sort from the previous exercise with the necessary changes so that the sorted elements are at the beginning of the list instead of at the end.
  - Using your English description of bubble sort, write an outline of the bubble sort algorithm in English.
  - Write function `bubble_sort()`.
  - Try it out on the test cases from `exercise_5()`.
7. Modify the “merge” program to compare bubble sort with insertion and selection sort. Explain the results.
8. The analysis of bubble sort: “Since  $N$  values have to be sorted, the overall running time is  $O(N \log_2 N)$ .” Point out a flaw in this reasoning, and explain what else affects the overall time bound.
9. There are at least two ways to come up with loop conditions. One of them is to answer the question, “When is the work done?” and then negate it. In function `merge()` ([Merge Two Sorted Lists](#), line 19), the answer is, “When we run out of items in one of the two lists,” which is described by the expression `l1 == len(l1) or l2 == len(l2)`. Negating this leads to our condition `l1 != len(l1) and l2 != len(l2)`.
- Another way to come up with a loop condition is to ask, “What are the valid values of the loop index?” In function `merge()`, the answer is “0 to  $N - 1$ ”

- ✓  $a = b \times k(1)$  and  $0 <= b < k(2)$ , where  $b$  and  $k$  start at zero we can drop the comparisons with zero, joining us  $a < k(1)$  and  $0 < k(2)$
- Is there another way to do it? Have you tried both approaches? Which do you prefer?
19. In function `merge()` in `MergeSort.java` on page 200, there are two calls to `covers()`. They are there because, when the preceding happens, one of the two lists still has some part that haven't been processed. Rewrite that loop so that these extra calls won't be needed.

# Object-Oriented Programming

Imagine you've been hired to help write a program to keep track of books in a bookstore. Every record about a book would probably include the title, authors, publisher, price, and ISBN, which stands for International Standard Book Number, a unique identifier for a book.

Read this code and try to guess what it prints:

```
python_book = Book()
```

```
    'Practical Programming'  
    ('Campbell', 'Gries', 'Maurits')  
    'Pragmatic Bookshelf'  
    1978.999999999999  
    25.99
```

```
survival_book = Book()
```

```
    'The Pragmatic Programmer's Survival Manual'  
    ('Carter')  
    'Pragmatic Bookshelf'  
    1992.999999999999  
    14.99
```

```
print("I was written by {} authors and costs {}.".format(  
    python_book.title, python_book.num_authors(), python_book.price))
```

```
print("I was written by {} authors and costs {}.".format(  
    survival_book.title, survival_book.num_authors(), survival_book.price))
```

You might guess that this code creates two book objects and called `Practical Programming` and one called `New Pragmatist's Survival Manual`. You might even guess the output:

```
Practical Programming was written by 3 authors and costs 25.99  
New Pragmatist's Survival Manual was written by 1 authors and costs 14.99
```

There's a problem, though. Our code doesn't run. Python doesn't have a `Book` type. And that's what this chapter is about: how to define and use your own types.

## 14.1 Understanding a Problem Domain

In our book example, we wrote the code based on what we assumed to do with books. The idea of a book originates from the problem domain: keeping track of books in a bookstore. We thought about this problem domain and figured out what features of a book we cared about.

We might have decided to keep track of the number of pages, the date it was published, and much more. What you decide to keep track of depends exactly on what your program is supposed to do.

It's common to define multiple related types. For example, if this book was part of an online store, we might also have an `Inventory` type, perhaps a `Shop` type, and much more.

Object-oriented programming involves several defining and using new types. As you learned in [Section 7.1, “Modules, Classes, and Methods,”](#) on page 115, a class is how Python represents a type. Object-oriented programming involves at least three phases:

1. **Understanding the problem domain.** This step is crucial: you need to know what your customer wants (your boss, perhaps a friend or colleague even just perhaps yourself) before you can write a program that does what the customer wants.
2. **Figuring out what types you may need.** A good starting point is to read the descriptions of the problem domain and look for the main nouns and noun phrases.
3. **Figure out what features you want each type to have.** Here you should write some code that uses the type you're thinking about, much like we did with the `Reps` code at the beginning of this chapter. This is a lot like the examples step in the function design recipe, where you decide what the code that you're about to write should do.
4. **Writing a class that represents this type.** You now need to tell Python about your type. To do this, you will write a class, including a set of methods inside that class. (You will use the function design recipe as you design and implement each of your methods.)
5. **Testing your code.** Your methods will have been tested automatically as you followed the function design recipe. But it's important to think about how the various methods will interact.

## 14.2 Function “isinstance,” Class Object, and Class Book

Function `isinstance` reports whether an object is an instance of a class—that is, whether an object has a particular type:

```
>>> isinstance(100, int)
True
>>> isinstance(100.0, int)
False
```

What is an instance of `int`? Int 553 is not.

Python has a class called `object`. Every other class is based on it:

```
>>> help(object)
Help on class object in module builtins:
```

```
class object
    |  the most basic type
```

Function `isinstance` reports that both `int` and `float` are instances of class `object`:

```
>>> isinstance(55.2, object)
True
>>> isinstance('abc', object)
True
```

Even classes and functions are instances of `object`:

```
>>> isinstance(str, object)
True
>>> isinstance(max, object)
True
```

What's happening here? Well, there is Python is derived from class `object`, and so every instance of every class is an object.

Using `object` or `object`'s type, we say that `class object` is the *superclass* of class `str`, and `class str` is a *subclass* of `class object`. This *superclass* information is available in the `help` documentation for a type:

```
>>> help(str)
Help on class str in module builtins:

class str(object)
```

Notice we see that `class SyntaxError` is a *subclass* of `class Exception`:

```
>>> help(SyntaxError)
Help on class SyntaxError in module builtins:

class SyntaxError(Exception)
```

Class object has the following attributes (all attributes are variables that refer to methods, functions, variables, or even other classes):

```
>>> dir(object)
['__class__', '__delattr__', '__dict__', '__format__', '__getattribute__',
 '__hash__', '__init__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
 '__setattr__', '__str__', '__subclasshook__']
```

Every class in Python, including ones that you didn't automatically inherit them, will inherit from class object; they are automatically part of every class. More generally, every subclass inherits the features of its superclass. This is a powerful tool: it helps avoid a lot of duplicate code and makes inheritance between related types consistent.

Let's try this out. Here is the complete class that we can write:

```
>>> class Book:
...     """Information about a book."""
...
...
```

Just the keyword `class` tells Python that we're defining a new function, followed by `Book`, which is the name that we're defining a new type.

Much like `int` is a type, `Book` is a type:

```
>>> type(int)
<class 'int'>
>>> type(Book)
<class 'type'>
```

One last thing: note that, in fact, because it has inherited all the attributes of class `object`,

```
>>> dir(Book)
['__class__', '__delattr__', '__dict__', '__dir__', '__eq__',
 '__format__', '__hash__', '__getattribute__', '__gt__', '__ge__',
 '__init__', '__le__', '__lt__', '__ne__', '__reduce__', '__reduce_ex__',
 '__setattr__', '__str__', '__subclasshook__']
```

If you look carefully, you'll see that this list is nearly identical to the output for `dir(object)`. There are four extra attributes in class `Book`: every subclass of class `object` automatically sees those three attributes in addition to the inherited ones:

```
'__dict__', '__module__', '__qualname__', '__weakref__'
```

We'll get to those attributes later on in this chapter in [What Are These `book` Attributes?](#) on page 383. First, let's create a `Book` object and give that book a title and a list of authors:

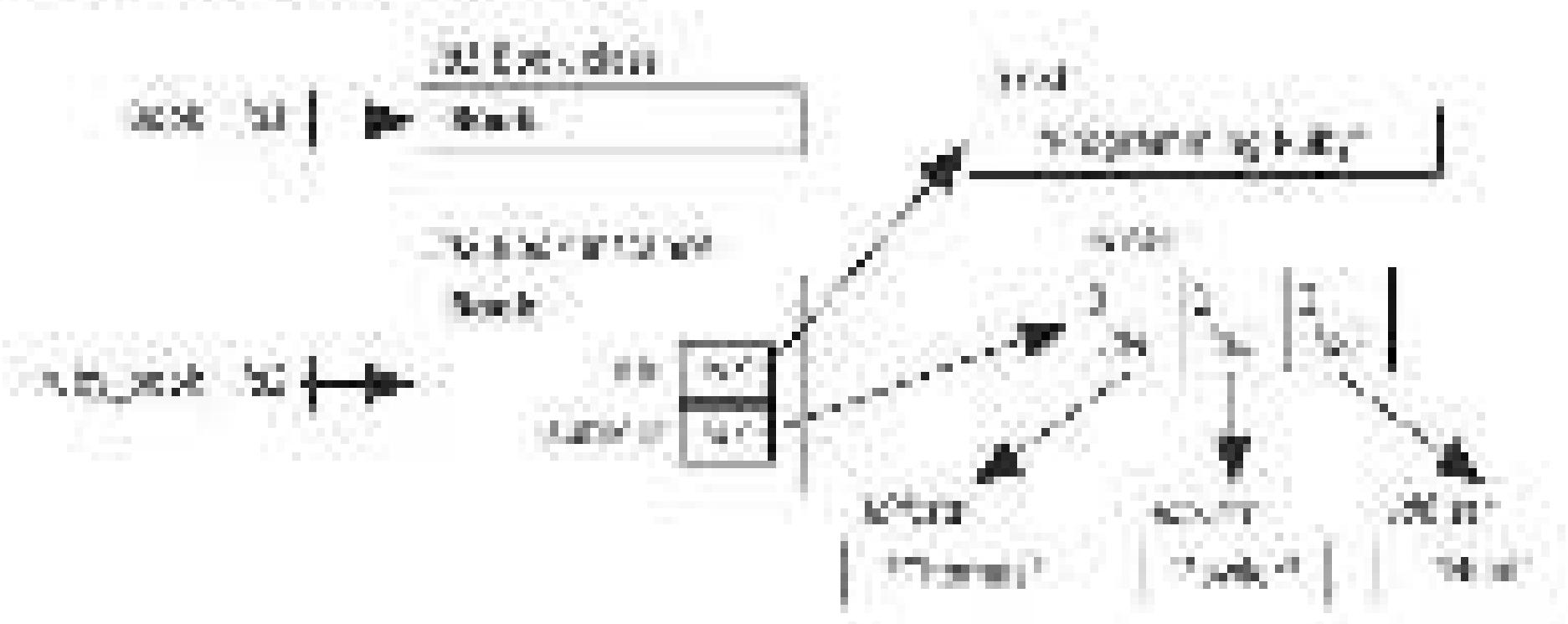
```
>>> ruby_book = Book()
>>> ruby_book.title = "Programming Ruby"
>>> ruby_book.authors = ['Thomas', 'Edwin', 'Hunt']
```

The first assignment statement creates a new object, and then assigns that object to variable `ruby_book`. The second assignment statement creates a new variable inside the `Book` object; this variable refers to the string "Programming Ruby". The third assignment statement creates variable `authors`, also inside the `Book` object, which refers to the list of strings ["Thomas", "Edwin", "Hunt"].

Variables `title` and `authors` are called **instance variables** because they are variables inside an instance of `Book`. We can access these instance variables through variable `ruby_book`:

```
>>> ruby_book.title
'Programming Ruby'
>>> ruby_book.authors
['Thomas', 'Edwin', 'Hunt']
```

In the expression `ruby_book.title`, Python looks variable `ruby_book`, then sees the dot and goes to the memory location of the book object, and then looks for variable `title`. Here is a picture:



We can even get help on our `Book` class:

```
>>> help(Book)
Help on class Book in module __main__:

class Book(builtins.object):
    |  Information about a book.

    |  Data descriptors defined here:

    |      __dict__
    |          dictionary for instance variables defined here

    |      __weakref__
    |          list of weak references to the object (if defined)
```

The first line tells us that we asked for help on class Book. After that is the header for class Book. The `__init__` part tells us that Book is a subclass of class object. The next line shows the Book constructor. Look at a section called “class \_\_init\_\_”, which are special pieces of information that Python keeps with every user-defined class that it uses for its own purposes. Again, we’ll get to [Code 14.5 What Are Those Special Attributes?](#) on page 293.

### 14.3 Writing a Method in Class Book

As you saw in [Chapter 7, Using Methods, exercise 1.15](#), there are two ways to call a method. One way is to access the method through the class, and the other is to use object-oriented syntax. These two calls are equivalent:

```
>>> str.capitalize('brooding')
'Brooding'
>>> 'brooding'.capitalize()
'Brooding'
```

We’d like to be able to write similar code involving class Book. For example, we might want to be able to ask how many authors a Book has:

```
>>> book = book.Book('ruby book')
A
>>> ruby_book = book.Book()
B
```

Right. That is tough, until we define a method called `num_authors` inside Book. Here it is:

```
class Book(object):
    def __init__(self, title):
        self.title = title
    def num_authors(self):
        return len(self.authors)
    return len(self.authors)
```

Book methods now accept `self` just like a function, except that it has a parameter called `self`. The type contract states that `self` refers to a Book. According to this class definition in the file `book.py`, we can import it, create a Book object, and call `num_authors` in two different ways:

```
>>> import book
>>> ruby_book = book.Book()
>>> ruby_book.title = 'Programming Ruby'
>>> ruby_book.authors = ['Thomas E. Fetter', 'Book']
>>> book.Book.num_authors(ruby_book)
```

```
<<< ruby>>> book.run_authors()
x

class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author
        self.publisher = None
        self.year = None
        self.price = None

    def run_authors(self):
        print(f'Book {self.title} has {len(self.authors)} authors')
        for author in self.authors:
            print(f'{author}')


<<< ruby>>> book.run_authors()
book.Book.run_authors(ruby_book)
```

The first part says to look in the imported module. In that module is class Book. Inside Book is method run\_authors. The argument to the call, ruby\_book, is passed to parameter self.

Python treats the second call on run\_authors exactly as it did the first: the first call is equivalent to this one:

```
<<< ruby>>> book.run_authors()
```

This second version is much more common because it lists the object first; we think of that version as asking the book how many authors it has. Thinking of methods like this very commonly helps develop an object-oriented mentality.

In the ruby\_book example, we assigned the title and the list of authors after the Book object was created. That approach isn't realistic: we don't want to have to type those extra assignment statements every time we create a book. Instead, we'll write a method that does this for us as we create the book. This is a special method and is called `__init__`. We'll also include the publisher, ISBN, and price as parameters of `__init__`:

```
class Book:
    """Information about a book including title, list of authors,
    publisher, ISBN, and price."""
    def __init__(self, title, author, publisher, isbn, year, price):
        self.title = title
        self.author = author
        self.publisher = publisher
        self.isbn = isbn
        self.year = year
        self.price = price
```

```
<<< python>>> book = Book('How to Do Everything',
                    'Computer Networks', 'O'Reilly',
                    '9780596100393', 2004, 49.95)
create a new book entitled 'title', authored by the author 'author',
published by 'publisher' with ISBN 'isbn' and present price '$price'

>>> book.title
'How to Do Everything'
>>> book.year
2004
>>> book.price
49.95
>>> book.authors
['Computer Networks']
>>> book.publisher
'O'Reilly'
```

```
'Pragmatic Semantics'  
www.johnwiley.com/wiley  
'Wiley-Blackwell'  
www.johnwiley.com/wiley  
Wiley
```

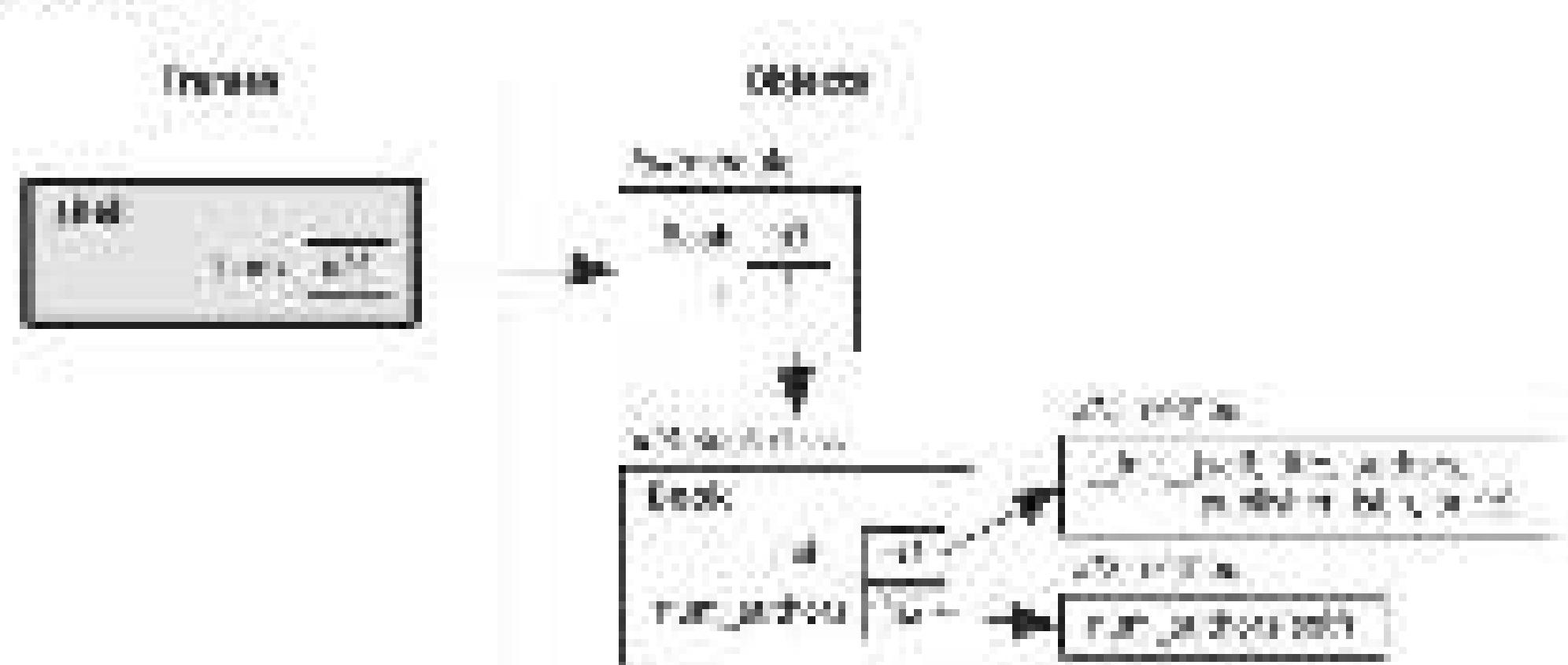
multi.title = title  
# Copy the authors list to ease the code generation that will later.  
multi.authors = authors[1]  
multi.publisher = publisher  
multi.ISBN = isbn  
multi.price = price

newAuthorList()  
= {  
 'Dowd' : 'John'  
}

newBookList()  
= {  
 'Pragmatic Semantics' : {  
 'title' : 'Pragmatic Semantics',  
 'author' : 'John Dowd',  
 'price' : 100,  
 'publisher' : 'Wiley-Blackwell',  
 'ISBN' : '978-0470-02200-1'  
 },  
 'Pragmatic Semantics' : {  
 'title' : 'Pragmatic Semantics',  
 'author' : 'John Dowd',  
 'price' : 100,  
 'publisher' : 'Wiley-Blackwell',  
 'ISBN' : '978-0470-02200-1'  
 }  
}

return len(multi.authors)

Note that we can exclude the code for methods just as we do for functions. This method contains a single (empty) return statement: the class definition. When Python executes this method, it creates a new object and assigns it to variable `Book`:



## What's in an Object?

Methods belong to classes; instances contain no objects or methods. How can we access or reference them? We need a method to get them.

```
>>> book = Book()
>>> print(book)
>>> book.info()
'Book object at 0x1000000000000000'
>>> book.title
'Practical Programming'
>>> book.author
'Campbell Grissom'
>>> book.year
2014
>>> book.isbn
'978-1-59327-548-1'
```

What happened here? Python has added variables to the object. These are called *instance variables* and they have the prefix *book.* The methods *info()* and *\_\_str\_\_()* are also present.

```
>>> print(book.__dict__)
{'__class__': <class 'book.Book'>,
 '__dict__': <attribute '__dict__' of 'book.Book' objects>,
 '__module__': 'book',
 '__weakref__': <attribute '__weakref__' of 'book.Book' objects>,
 '_title': 'Practical Programming',
 '_author': 'Campbell Grissom',
 '_year': 2014,
 '_isbn': '978-1-59327-548-1'}
```

Notice that *get\_title()*, *get\_author()*, *get\_year()*, and *get\_isbn()* are all available in the object *book* because they are methods in the class *Book*.

**Method *\_\_init\_\_*** is called whenever a Book object is created. Its purpose is to initialize the new object. This method is sometimes called a constructor. Here are the steps that Python follows when creating an object:

1. It creates an object at a particular memory address.
2. It calls method *\_\_init\_\_* passing to the new object three parameters: *self*.
3. It initializes that object's memory address.

Let's try it out in the shell:

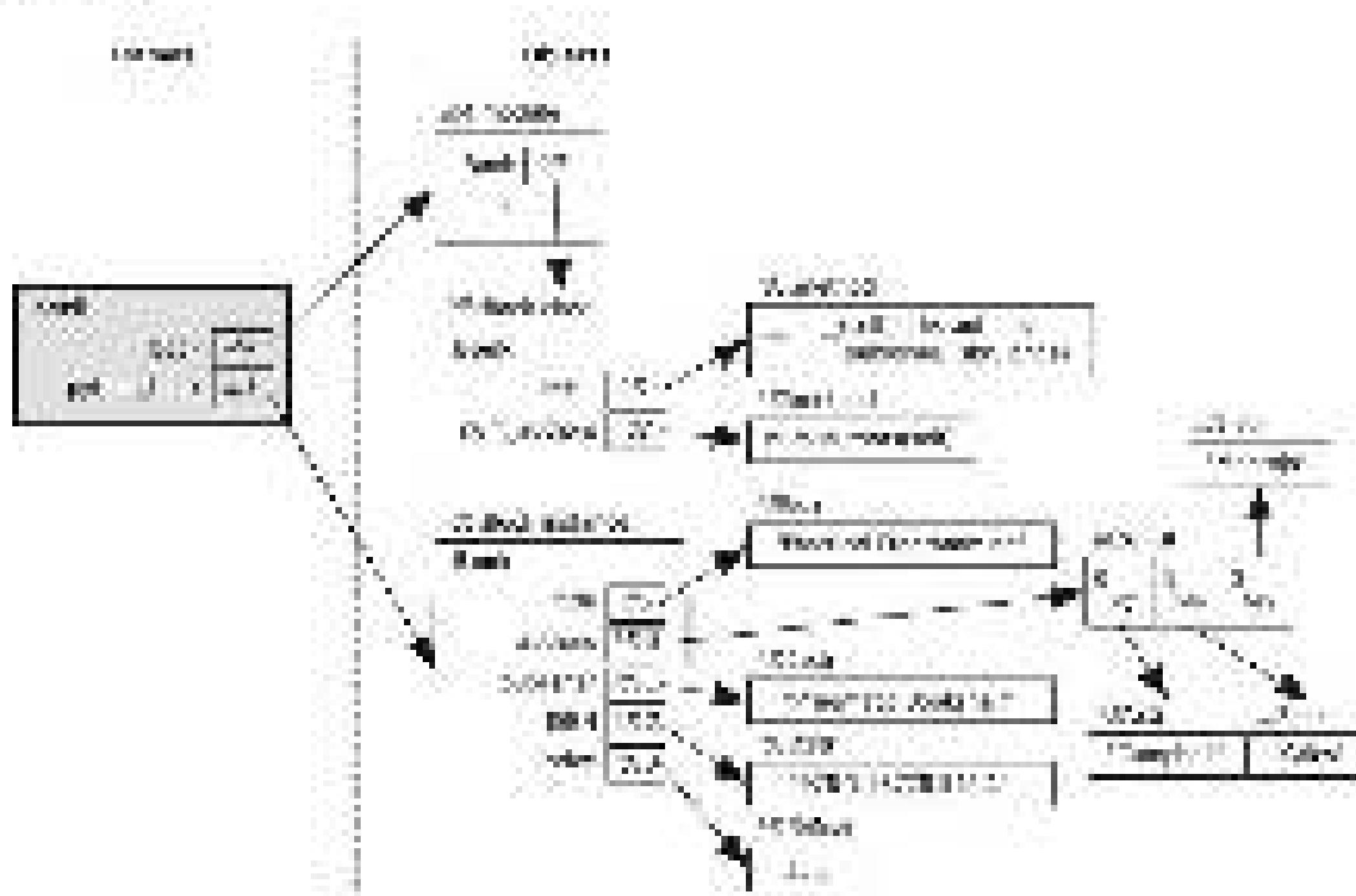
```
>>> import book
>>> python_book = book.Book()
...     'Practical Programming',
...     'Campbell', 'Grissom', 'Version 1',
...     'Pragmatic Bookshelf',
...     '978-1-59327-548-1'.
```

```

...      25.0)
>>> python_book.title
'Practical Programming'
>>> python_book.authors
['Campbell', 'Giles', 'Horrevoets']
>>> python_book.publisher
'Pragmatic Bookshelf'
>>> python_book.ISBN
'978-1-93770-545-1'
>>> python_book.price
25.0

```

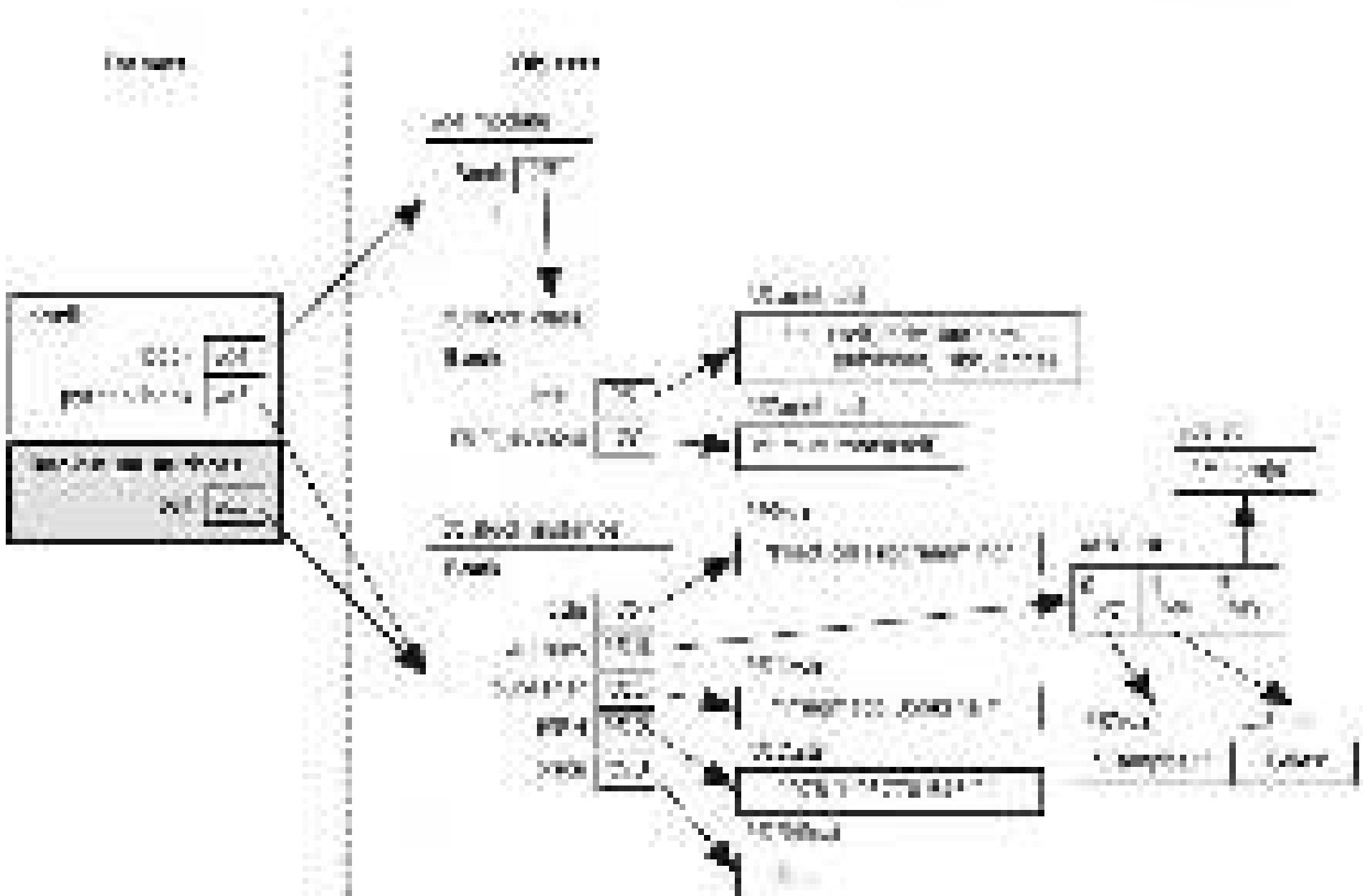
The following picture gives a picture of the memory model that results from this code:



`__str__`, `__method__`, and `__new__` are `method`s. As a reminder, they belong to the `book` class (`book.py`) rather than `Book`.

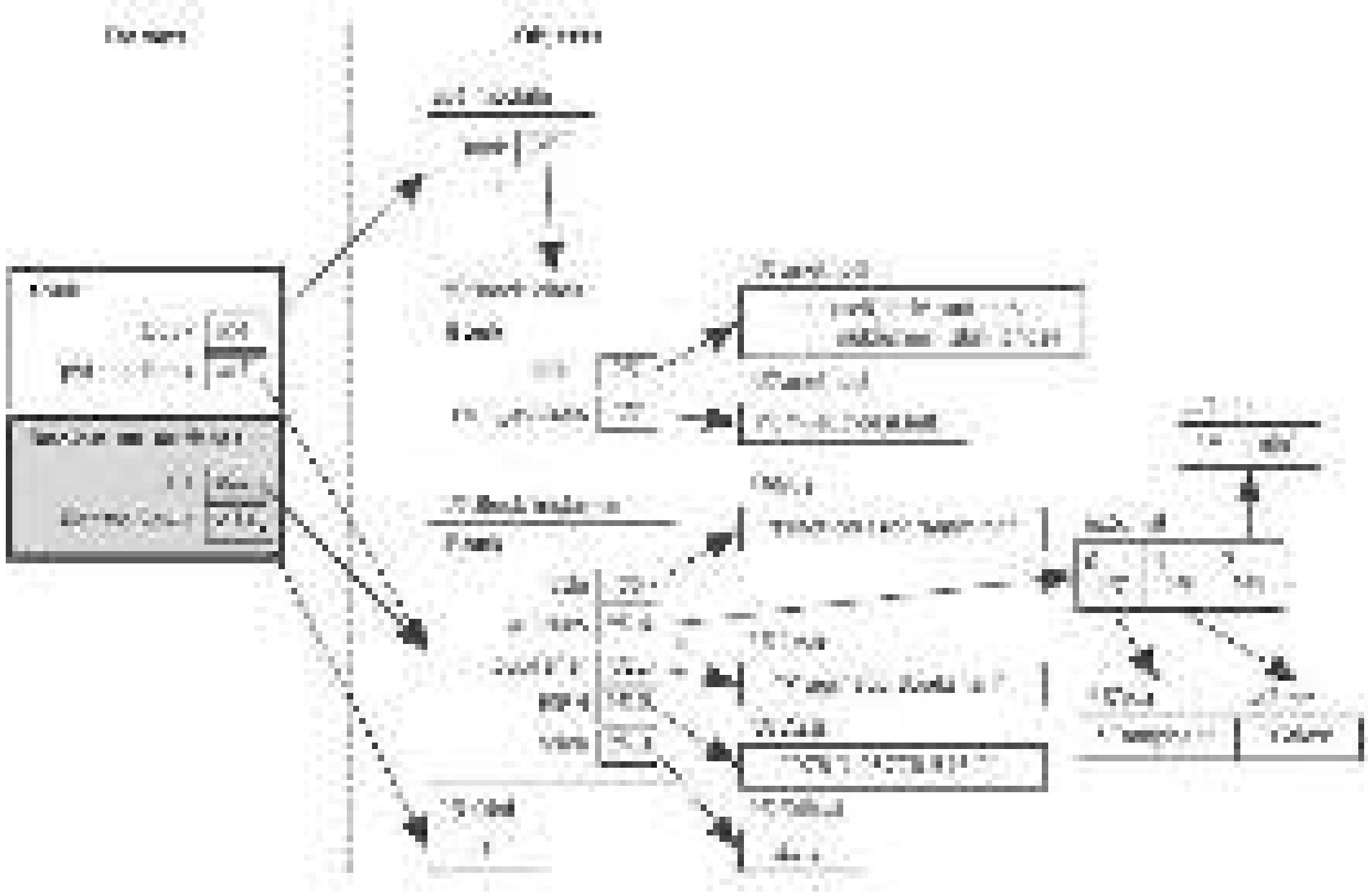
Python first finds the object that `python_book` refers to and calls its method `__str__`. There are no explicit arguments, so Python only passes in the `book` object that `python_book` refers to, assigning that object to the `self` parameter. (See [Figure 14. Coding a method](#) on page 279.)

The return statement, `return str(self)`, is then executed. The expression, `len(self.authors)`, is a function call. Python evaluates the argument `self.authors` by finding the object that `self` refers to and then, in that object, finds instances



**Figure 14** Calling a method

variable `size`. This equals 3, and the length of that list is the value that Python returns, as shown here:



With constructors, methods, and instance variables in hand, we can now create classes that look and work like those that come with Python itself.

## 14.4 Plugging Into Python Syntax: More Special Methods

In Section 7.4, “What Are These Underscores?”, on page 193, you learned that some Python syntax, such as `f(x)`, triggers method calls. For example, when Python sees “`abc`” → “`123`”, it turns that into `abc__str__(123)`. When we call `print()`, then `__str__()` is called to find out what string to print.

You can do this too. All you need to do is define these special methods inside your classes:

This output Python evaluates when we print a Book and can be very useful:

```
>>> python_book = Book()
...
>>> 'Practical Programming'
...
>>> ('Campbell', 'Eric', 'Richter')
...
>>> 'Pragmatic Bookshelf'
...
>>> '978-1-93776-548-1'
...
>>> 25.91
>>> print(python_book)
Book Book object at 0x00551438
```

This is the default behavior for converting objects to strings. It just shows us where the object is in memory. This is the behavior defined in class `object`'s method `__str__`, which our `Book` class has inherited.

If we want to present a more useful string, we need to replace one more special method, `__repr__`. `__repr__` is called when an informal human-readable version of an object is needed, and `__repr__` is called when an uninformative but possibly less readable output is desired. In particular, `__repr__` is called when print is used, and it is also called by function `str` and by string method `format`. Method `__repr__` is called when you ask for the value of a variable in the Python shell, and it is also called when a collection such as `list` is printed.

Let's define method `Book.__repr__` to print a useful output: this method goes inside class `Book`, along with `__init__` and `__author__`:

```
def __repr__(self):
    """Prints a book object's string representation, for use in the Python shell.

    Returns a string containing the book's title, author(s), and price.
    """
    return f'{self.title} by {self.authors} costs {self.price}.'
```

---

```
Author --> Eric S.
Author --> Eric S.
Author --> Eric S.
Author --> Eric S.
```

`price = 24.95` # format

```
def __str__(self):
    self.title = ' '.join(self.authors)
    self.pub_info = self.year, self.page, self.price
    return f'{self.title} ({self.authors}) {self.pub_info} {self.price}'
```

`print(book)` now shows meaningful information:

```
>>> python_book = Book()
...     'Practical Programming',
...     ['Campbell', 'Gries', 'Montejo'],
...     'Pragmatic Bookshelf',
...     '978-1-60000-345-1',
...     25.95
>>> print(python_book)
Title: Practical Programming
Author: Campbell, Gries, Montejo
Publisher: Pragmatic Bookshelf
ISBN: 978-1-60000-345-1
Price: $25.95
```

The method `__str__` is called to get an (unambiguously) string representation of an object. The string should include the type of the object as well as the values of any instance variables. Ideally, if we were to evaluate the string, it would create an object that is equivalent to the one that came from the `__str__`. We will show an example of `__str__` in [Section 14.6, “A Case Study: Metaclasses, Atoms, and PCD Files](#), on page 266.

The operator `==` triggers a call on method `__eq__`. This method is defined in class `Object`, and on those two lines of intuition, it makes `==` produce the exactly what an object is compared to itself. That means that even if two objects contain identical information they will not be considered equal:

```
>>> python_book_1 = book.Book()
...     'Practical Programming',
...     ['Campbell', 'Gries', 'Montejo'],
...     'Pragmatic Bookshelf',
...     '978-1-60000-345-1',
...     25.95
>>> python_book_2 = book.Book()
...     'Practical Programming',
...     ['Campbell', 'Gries', 'Montejo'],
...     'Pragmatic Bookshelf',
...     '978-1-60000-345-1',
...     25.95
>>> python_book_1 == python_book_2
False
>>> python_book_1 == python_book_1
True
>>> python_book_2 == python_book_2
True
```

We can override an inherited method by defining a new version in our subclass. This replaces the inherited method so that it is no longer used. As an example, we'll override method `__eq__` to compare two books by equality. Because ISBNs are unique, we can compare using them (and this method is called `__eq__`):

```
def __eq__(self, other):
    """Compare two Book objects for equality.

    Return True if the two books are equal based on their ISBNs.
    """
    return self.ISBN == other.ISBN
```

Here is our new method in action:

```
>>> python_book_1 = Book('Book 1',
...     'Practical Programming', [Campbell, Horst, Berndt], 1,
...     'Pragmatic Bookshelf', '978-1-933924-00-1', 25.0)
>>> python_book_2 = Book('Book 2',
...     'Practical Programming', [Campbell, Horst, Berndt], 1,
...     'Pragmatic Bookshelf', '978-1-933924-00-1', 25.0)
>>> survival_book = Book('Book 3',
...     'How to Program a Survival Toolkit', [Tarter], 1,
...     'Pragmatic Bookshelf', '978-1-933924-00-1', 15.0)
>>> python_book_1 == python_book_2
True
>>> python_book_1 == survival_book
False
```

Here are the basic rules for a method call dispatch:

1. Look in the current object's class. If we find a method with the right name, use it.
2. If we didn't find it, look in the superclass. Continue up the class hierarchy until the method is found.

Python has lots of other special methods. The official Python website gives a full list.

## 14.5 A Little Bit of OO Theory

Classes and objects are two of programming's power tools. They let good programmers do a lot in very little time. But with them, bad programmers can create a real mess. This section will introduce some underlying theory that will help you design reliable, reusable, object-oriented software.

## What Are Those Special Attributes?

In Section 14.5, *Function References, Class Objects, and Class Constructors*, we discussed three other special class attributes:

`__dict__`, `__module__`, and `__name__`.

Ever noticed you can do this:  
`class C:`  
 `pass`  
`c = C()`  
`c.__dict__`

That's because `c` is an instance of `C`, which means it has its own dictionary. What you might not have noticed is that `c` is also an instance of `object`. The following code illustrates this:

```
class C(object):
    pass

c = C()
print(c.__dict__)
# Output: {'__class__': <class 'C'>, '__module__': '__main__', '__dict__': <attribute '__dict__' of 'C' objects>, '__weakref__': <attribute '__weakref__' of 'C' objects>, '__doc__': None}
```

What is `__dict__`? It is the instance variable. It changes the contents of an object's dictionary. You can even change it to something else, although you don't recommend it.

Here are brief descriptions of some of the other special attributes of classes:

`__module__`: refers to the module stated to which the class of the object was defined.

`__name__`: used by Python to manage where the memory for an object can be located.

`__bases__` and `__qualname__`: referring to strings containing the simple and fully qualified names of classes, respectively. These values are usually identical, except when class is defined inside another class, in which case the fully qualified name contains both the outer class name and the inner class name.

`__dict__`: refers to an object's class object.

These are all special class attributes and they are all used by Python to properly manage inheritance and organization of classes.

## Encapsulation

To encapsulate something means to contain it in another kind of container. In programming, encapsulation means keeping data and the code that uses it in one place and hiding the details of exactly how they work together. For example, each instance of class `file` keeps track of what file on the disk it is reading or writing and where it currently is in that file. The class hides the details of how this is done so that programmers can use it without needing to know the details of how it was implemented.

## Polyorphism

**Polyorphism** means “having more than one form.” In programming, it means that an expression having a variable can mean different things depending on the type of the object to which the variable refers. For example, if `left` refers to a string, then `left[3]` produces a two-character string. If `left` refers to a list, on the other hand, the same expression produces a three-element list. Similarly, the expression `left[1:right]` can produce a number, a string, or a list, depending on the types of `left` and `right`.

Polyorphism is useful for writing modern programs because it cuts down on the amount of code you have to write to make and test. It lets us write a generic function to cover many kinds.

```
def nonBlankLinesIn(thing):
    """Return the number of non-blank lines in thing."""
    count = 0
    for line in thing:
        if line.strip():
            count += 1
    return count
```

And then we can apply it to a list of strings, a file, a web page on a site halfway around the world (see [Section 10.4: Files over the Internet](#) on page 191), or a single string wrapped up in class `String` to look like a file. Each of these four types has to be the subject of a long, muddled switch, such as `lambda`, below to produce its “next” element as long as there is one until there may “all done.” That means that instead of writing four functions to cover the varying lines of copying the lines into a list and then applying our function to that list, we can apply one function to all those types directly.

## Inheritance

Giving one class the same methods as another is one way to make them polyomorphic, but it suffers from the same flaw as initializing an object’s instance variables from scratch: the object. If a programmer forgets just one line of code, the whole program can fail for reasons that will be difficult to track down. A better approach is to use a third fundamental feature of object-oriented programming called inheritance, which allows you to reuse code in yet another way.

Whenever you create a class, you see, using inheritance, your new class automatically inherits all of the attributes of class `object`, much like a child

members inheritable from her or her parents. You can also declare that your new class is a subclass of some other class.

Here is an example. Let's say we're managing people at a university. There are students and faculty. (This is a gross oversimplification for purposes of illustrating inheritance; we're ignoring administrative staff, caretakers, food providers, and more.)

Both students and faculty have names, postal addresses, and email addresses; each student also has a student number, a list of courses taken, and a list of courses he or she is currently taking. Each faculty member has a faculty number and a list of courses he or she is currently teaching. (Again, this is a simplification.)

We'll have a Faculty class and a Student class. We must teach them to have names, addresses, and email addresses. But duplicate code is generally a bad thing, so we'll avoid it by (1) Defining a class, perhaps called Person, and keeping track of those features in Person. Then we'll make both Faculty and Student subclasses of Person:

#### `class Person:`

```
    + member of a University, int
    def __init__(self, name, address, email)
        self.name = name
        self.address = address
        self.email = email
```

Create a new student object that will have name, address, and email properties.

```
self.name = "John"
self.address = "address"
self.email = "email"
```

#### `class Faculty(Person):`

```
    + faculty member at a University, int
```

```
def __init__(self, name, address, email, faculty_num,
            courses = [], faculty_teaching = None)
    super().__init__(name, address, email)
    self.faculty_number = faculty_num
    self.courses_teaching = courses
```

Create a new faculty named `rafa`, with `name` address, `email` address, `faculty_number` `faculty_num`, and `empty` list of `courses`.

```
rafa = Faculty("rafa", "address", "email",
               faculty_num = 12345,
               courses_teaching = [])
```

```
class Student(Worker):
    #<student number> > student_number, ...
```

```
def __init__(self, name, address, email, student_number,
            courses_taken=0, courses_being_taught=0):
    Worker.__init__(self, name, address, email)
    self.student_number = student_number
    self.courses_taken = courses_taken
    self.courses_being_taught = courses_being_taught
```

Create a new student named `jan`, with `face` address, `mail` address, `student_number` `1234`, an empty list of `courses taken`, and an empty list of `current courses`.

```
super().__init__(name, address, email)
self.student_number = student_number
self.courses_taken = []
self.courses_being_taught = []
```

Both `class Student`—`class FacultyWorker` and `class Worker`—is Python class for `Faculty` and `Student` are subclasses of `class Worker`. That means that they inherit all of the attributes of `class Worker`.

The first line of both `Faculty.__init__` and `Student.__init__` function is one which provides a reference to the `superclass` part of the object, `Worker`. That means that both of those first lines call method `__init__`, which was inherited from `class Worker`. Notice that we just pass the relevant parameters in as arguments to this call, just as we would with any method call.

If we import these into the shell, we can create both faculty and students:

```
>>> paul = Faculty('Paul Morris', 'Aja...', 'pmorris@carleton.edu', 12345)
>>> paul.name
'Paul Morris'
>>> paul.address
'pmorris@carleton.edu'
>>> paul.faculty_number
12345
>>> jan = Student('Jan Campbell', 'Toronto', 'campbellj@carleton.edu',
...                 12345)
>>> jan.name
'Jan Campbell'
>>> jan.address
'campbellj@carleton.edu'
>>> jan.student_number
12345
```

Both the `Faculty` and `Student` objects have inherited the features defined in their `Worker`:

Often, you'll want to expand the behavior inherited from a superclass. As an example, we might want to add a `method` inside `class Worker`:

```
def str (self):
    return "Member(%d, %d, '%s')  
def __init__(self, id, age, name):
    self.id = id
    self.age = age
    self.name = name
    self.address = None
    self.formation = None
```

With this method added to Cisco Meraki's both Router and Switch libraries it

and Paul = Pauline Paul, 4100, [pauline@arcor.de](mailto:pauline@arcor.de), +3294-  
9000-1011111  
"Pauline just says it's been a nice day"  
and Paul (Paul)  
Paul wrote:  
A.W.  
M.1995/6, T1 (GTO, etc)

That's not quite enough though: for other details, we want to know what the Faculty's s. does, adding the faculty number and the list of courses the faculty member is teaching, and a student service should include the equivalent student service information.

We'll use super again to access the liberated `Penbox_ST_method` and to append the `Penbox_Shortcut` command:

## **der Kriegsfall.**

Return a string representation of this factory.

~~new faculty = Faculty('Paul', 'Mysl', 'Engineering, Corvallis, OR')~~ 2234  
~~new faculty = str\_1~~  
~~'Paul Myslak' refers to the Faculty object 2234 in Figure 8.2.~~

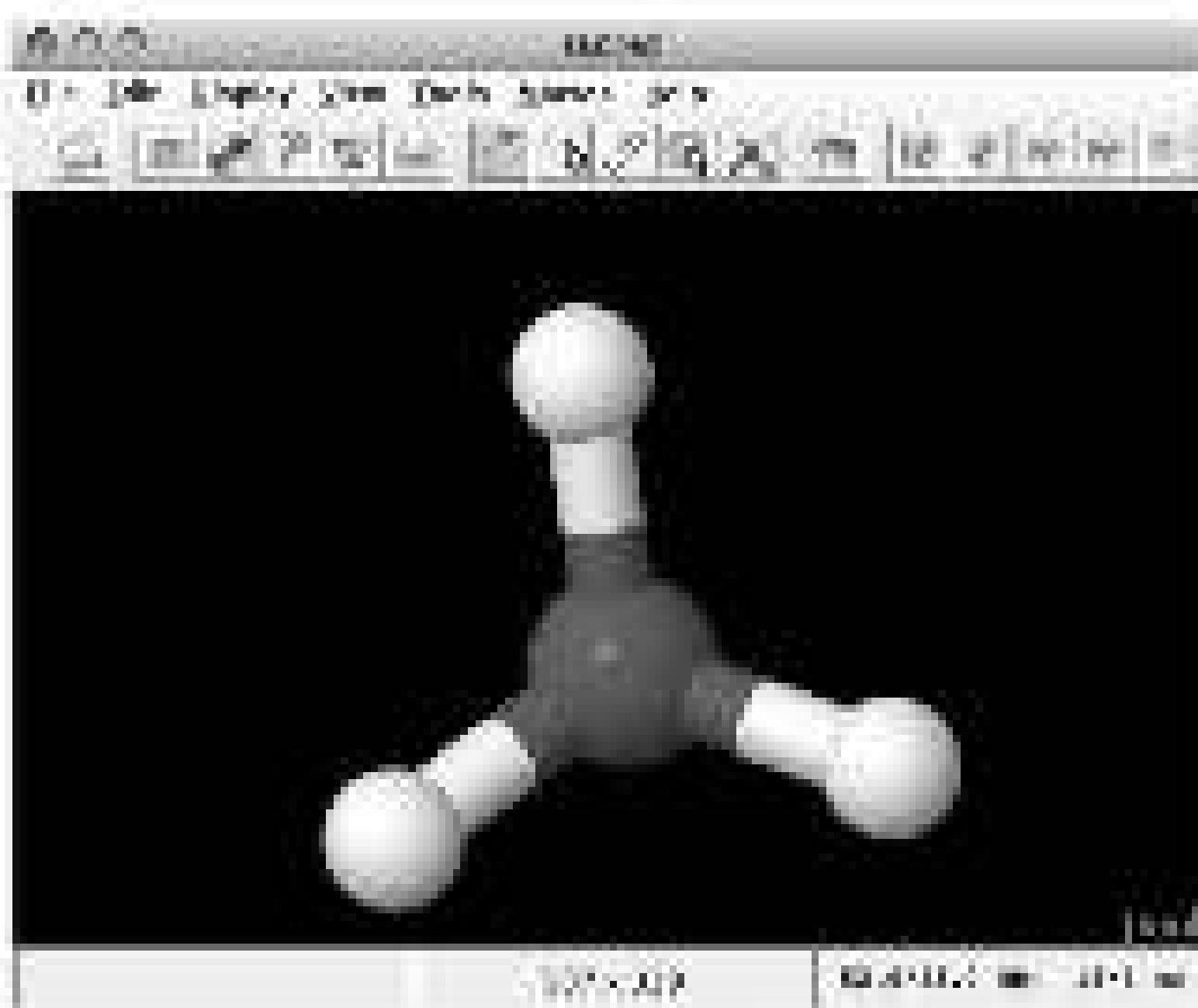
With this we get the standard output

:<> paul = Faculty[Paul], "Also", portugueseStudents.dir, "3331">  
:<> atp(paul)

```
'Paul/ytj@ppg-toronto:~/edu/1304/rocces' ~
one point (pos1)
Paul rocco
3.0
pp-classes.toronto.edu
1.2.0
Courses:
```

## 14.6 A Case Study: Molecules, Atoms, and PDB Files

Molecular graphic visualization tools allow for interactive examination of molecular structures. Most read PDB-formatted files, which we describe in Section 14.7. [http://PDB.Molsoft.com/pdb/1S04.pdb](#). For example, Jmol displays the following snapshot to a Java-based open-source 3D viewer for these structures.



In a molecular visualization, every atom, molecule, bond, and so on has a location in 3D space, usually defined as a vector, which is an arrow from the origin to where the structure is. All of these structures can be rotated and translated.

A vector is usually represented by  $\alpha$ ,  $\beta$ , and  $\gamma$  coordinates that specify how far along the  $x$ -axis,  $y$ -axis, and  $z$ -axis the vector extends.

Here is how ammonia can be specified in PDB format:

| COMPOUND | AMMONIA                |
|----------|------------------------|
| ATOM     | 1 H 0.287 -0.363 0.000 |
| ATOM     | 2 H 0.287 0.727 0.000  |
| ATOM     | 3 H 0.771 -0.727 0.000 |
| ATOM     | 4 H 0.771 0.727 -0.000 |
| END      |                        |

In our simplified PDB format, a molecule is made up of numbered atoms. In addition to the number, an atom has a symbol and  $(x, y, z)$  coordinates. For example, one of the atoms in ammonia is nitrogen, with symbol N at coordinates (1.187, -0.211, 1.14). In the following sections, we will look at how we could translate these atoms into object-oriented Python.

## Class Atom

We might want to create an atom like this using information we read from the PDB file:

```
nitrogen atom(1), N, 1.187, -0.211, 1.14
```

To do this, we'll need a class called Atom with a constructor that creates all the appropriate instance variables:

### Class Atom

```
class Atom:
    """An atom with a name, symbol, and coordinates. All
    fields are floats except name and symbol.

    def __init__(self, num, sym, x, y, z):
        """name, num, symbol, x, y, z are floats
        Create an atom with number num, string symbol sym, and float
        coordinates (x, y, z).

        self.number = num
        self.center = [x, y, z]
        self.symbol = sym

    def __str__(self):
        """Atom(1), N, 1.187, -0.211, 1.14
        Return a string representing this atom in this format.

        return '{:}(1), {}({},{},{})'.format(
            self.symbol, self.number[0], self.number[1], self.number[2])

    def __eq__(self):
        """Atom(1) == self
        Return a string representation of this atom in this format.

        return 'Atom(1), N, 1.187, -0.211, 1.14'
```

```

    return "(x1,y1,z1), (x2,y2,z2), (x3,y3,z3))" + str(x)
    self.xname = self.symbol
    self.xname[0], self.xname[1], self.xname[2])

```

We'll use these later when we define molecules for molecules.

In simulations, one common operation is translation, or moving an atom to a different position. We'd like to be able to write this in order to shift an atom's position by  $\alpha/2$  units:

```
nitrogen.translate(0, 0, 2)
```

This code works as expected if we add the following method to class Atom:

```

def translate(self, x, y, z):
    (self.xname, self.yname, self.zname) = (x, y, z)
    print("Atom " + self.name + " moved to (%s, %s, %s)." % (x, y, z))

```

```

    self.xcenter = (self.xcenter[0] + x,
                    self.xcenter[1] + y,
                    self.xcenter[2] + z)

```

## Class Molecule

Remember that we read PDB files one line at a time. When we reach the line containing CCP4IC APPEND, we know that we're building a complex structure: a molecule with a name and a list of atoms. Here's the start of a class for this, including an add method that adds an atom to the molecule:

```
class Molecule:
    """A molecule with a name and a list of Atoms. ***
```

```
    def __init__(self, name):
        self.name = name
        self.list = None
```

Create a Molecule object with no atoms.

```

    self.list = None
    self.list = []
```

```
    def addAtom(self, atom):
        (Molecule, Atom) == > NoneType
```

Add *a* to my list of atoms.

```
    self.list.append(atom)
```

As we read through the remaining PDB information, we will always see we find them; here is the code from Section 10.7, *MoleculeReader*, on page 291, rewritten to return a *Molecule* object instead of a list of tuples:

```
from molecule import Molecule
from atom import Atom

def read_molecule(r):
    """read(r) -> Molecule

    Read a single molecule from r and return it,
    or return None to signal end of file.

    If there isn't another line, return the end of the file.
    line = r.readline()
    if not line:
        return None

    # None of the entries are comments.
    key, num = line.split()

    # Other lines are either empty or have one entry.
    molecule = Molecule(key)
    reading = True

    while reading:
        line = r.readline()
        if line.strip() == '':
            reading = False
        else:
            key, num, kind, x, y, z = line.split()
            molecule.add(Atom(num, kind, float(x), float(y), float(z)))

    return molecule
```

If we compare the two versions we can see the code is nearly identical. It's just as easy to read the new version as the old—more so even, because it includes type information. Here are the *\_str\_* and *\_repr\_* methods:

```
def __str__(self):
    """Molecule() -> str

    Return a string representation of this Molecule in this format:
    ATOM1 ATOM2 ... N
```

```
    self = ''
    for atom in self.atoms:
        self = self + atom.__str__() + '\n'
```

```

    # Strip off the last atom.
    res = res[:2]
    return "Molecule(" + str(res) + ", " + res[-1] + ")"
}

def __repr__(self):
    """Molecule(self) >>> self

    Return a string representation of this Molecule or this Forest.
    Molecule: <Molecule: (ATOMS: ATOMS ... )>
    Forest: <Forest: FORESTS>
    """

    res = ""
    for atom in self.atoms:
        res += atom.__repr__() + " "
    # Strip off the last atom.
    res = res[:-2]
    return "Molecule(" + str(res) + ", " + res[-1] + ")"
}

```

We'll add a version method to `Molecule` to make it easier to move:

```

def translate(self, x, y, z):
    """Molecule, number, number, number) -> NoneType

    Move this Molecule, including all Atoms, by (x, y, z).
    """

    for atom in self.atoms:
        atom.translate(x, y, z)

```

And here we'll call it:

```

ammonia = Molecule("NH3")
ammonia.translate(1, 0, 0)
ammonia.translate(0, 1, 0)
ammonia.translate(0, 0, 1)
ammonia.translate(1, 1, 1)
ammonia.translate(0, 0, 0)

```

## 14.7 Classifying What You've Learned

In this chapter, you learned the following:

- In object-oriented languages, new types are defined by creating classes. Classes support encapsulation: in other words, they combine data and the operations on it so that other parts of the program can ignore implementation details.
- Classes also support polymorphism. If two classes have methods that work the same way, instances of those classes can replace one another without the rest of the program being affected. This enables “plug-and-

play programming, in which different objects can perform different operations, depending on the object it is operating on.

- Finally, new classes can be defined by inheriting features from existing ones. The new class can override the features of its parent and/or add entirely new features.
- When a method is defined in a class, the first argument must have variable `self` that represents the object the method is being called on. By convention, this argument is called `self`.
- Some methods have special predefined meanings in Python. In particular, their names begin and end with two underscores. `__init__` methods are called when constructing objects (`_init_`) or converting them to strings (`__str__` and `__repr__`); whereas, `__eq__` and `__lt__` are used in function annotations.

## 14.8 Exercises

Here are some exercises for you to try for your own. Solutions are available at [http://programcandy.ca/python/oop/coding\\_exercises.html](http://programcandy.ca/python/oop/coding_exercises.html).

- a. In this exercise, you will implement class `Country`, which represents a country with a name, a population, and an area.

- b. Here is a sample interaction from the Python shell:

```
>>> canada = Country('Canada', 3448279, 9984670)
>>> canada.name
'Canada'
>>> canada.population
3448279
>>> canada.area
9984670
```

The code above cannot be executed yet because class `Country` does not exist. Define `Country` with a constructor method (`__init__`) that has four parameters: a country's name, its population, and its area.

- c. Consider this code:

```
>>> canada = Country('Canada', 3448279, 9984670)
>>> usa = Country('United States of America', 313614642, 9363375)
>>> canada.is_larger(usa)
True
```

In class `Country`, define a method named `is_larger` that takes two `Country` objects and returns `True` if and only if the first has a larger area than the second.

i. Consider this code:

```
>>> canada.population_density()
2.4535722002227005
```

In class `Country`, define a method named `population_density` that returns the population density of the country (people per square km).

ii. Consider this code:

```
>>> usa = Country('United States of America', 323814048, 9328075)
>>> print(usa)
United States of America has a population of 323814048 and an 9328075
square km.
```

In class `Country`, define a method named `_str_` that returns a string representation of the country in the format shown above.

iii. After you have written `_str_`, this screen shot shows that a `_repr_` method would be useful:

```
>>> canada = Country('Canada', 3448279, 6604670)
>>> canada
<__main__.Country object at 0x7f2ab20b550>
>>> print(canada)
Canada has population 3448279 and an 6604670 square km
>>> [canada]
[<__main__.Country object at 0x7f2ab20b550>]
>>> print([canada])
[<__main__.Country object at 0x7f2ab20b550>]
```

Define the `_repr_` method in `Country` to produce a string that behaves like this:

```
>>> canada = Country('Canada', 3448279, 6604670)
>>> canada
Country('Canada', 3448279, 6604670)
>>> [canada]
[Country('Canada', 3448279, 6604670)]
```

ii. In this exercise, you will implement a `Continent` class, which represents a continent with name and a list of countries. Use `Country` with the class `Country` from the previous exercise. If `Country` is defined in another module, you'll need to import it.

a. Here is a sample interaction from the Python shell:

```
>>> canada = country.Country('Canada', 3448279, 6604670)
>>> usa = country.Country('United States of America', 323814048,
... 9328075)
>>> mexico = country.Country('Mexico', 112386588, 1549550)
```

>>> countries = [canada, usa, mexico]  
 >>> north\_america = Continent('North America', countries)  
 >>> north\_america.name  
 'North America'  
 >>> for country in north\_america.countries:  
     print(country)

Canada has a population of 34462776 and is 9604672 square km.  
 United States of America has a population of 31334040 and is 936675 square km.

Mexico has a population of 11200686 and is 1043050 square km.

The code above cannot be executed yet, because class `Continent` does not exist. Define `Continent` with a constructor (`__init__`) that has three parameters: `continent`, `name`, and `list_of_countries`.

### b. Consider this code:

```
>>> north_america.total_population()  

26772227
```

In class `Continent`, define a method `total_population` that returns the sum of the populations of the countries in this continent.

### c. Consider this code:

```
>>> print(north_america)  

North America  

Canada has a population of 34462776 and is 9604672 square km.  

United States of America has a population of 31334040 and is 936675 square km.  

Mexico has a population of 11200686 and is 1043050 square km.
```

In class `Continent`, define a method `__str__` that returns a string representation of the continent in the format shown above.

## 3. In this exercise you'll write `_str_` and `_eq_` methods for several classes.

- In class `Student` write a `_str_` method that includes all the student information and an address includes the student number, the list of courses taken, and the list of current courses.
- Write `_eq_` methods in classes `Person`, `Student` and `Faculty`.

Create a few `Student` and `Faculty` objects and call `print` or `len` to verify the functionality that you wrote in.

4. Write a class called `Human` to keep track of information about a human, including a variable for the body length (in millimeters; they are about 1 mm to length), gender (either `female` or `male`), and age (in days). Include methods `setLength()` and `getLength()`.

**5. Consider this code:**

```
Line segment s = LineSegment(Point(0, 0), Point(3, 2));
double segmentSlope();
3.0
double segmentLength();
2.73861327743379
```

In this example, you will write two classes, `Point` and `LineSegment`, so that you can run the code above and get the same results.

- Write a `Point` class with an `int` constructor that takes two numbers as parameters.
- In the same file, write a `LineSegment` class whose constructor takes two `Point`s as parameters. The first `Point` should be the start of the segment.
- Write a `slope` method in the class `LineSegment` that computes the slope of the segment. (Hint: The slope is a float larger than zero.)
- Write a `length` method in class `LineSegment` that computes the length of the segment. (Hint: Use `>>` to raise `a` to the `n`th power. To compute `b^n`, square `b`, raise `a` number of times.)

# Testing and Debugging

How can you tell whether the programme you write works correctly? Following the [buncher test](#) recipe from [Section 3.8, Testing Your Functions: A Recipe](#), on page 47, you include an example call or two in the docstring. The last step of the recipe is calling your function to make sure that it returns what you expect. But are one or two calls enough? If not, how many do you need? How do you pick the arguments for these function calls? In this chapter, you will learn how to choose good test cases and how to test your code using Python's built-in module.

Finally, what happens if your tests fail, resulting in bugs? (See [Section 1.8, What's a Bug?](#), on page 1.) How can you tell where the problem is in your code? In this chapter, you'll see how to find and fix bugs in your programs.

## 15.1 Why Do You Need to Test?

Quality assurance, or QA, checks that software is working correctly. Over the last thirty years, programmers have learned that quality isn't something you can just add at the end; you can't make up a program after it has been written. Quality has to be designed in, and software must be tested and reviewed to check that it meets standards.

The [quality cost](#) of finding bugs after they've been put into your software actually makes you more productive overall. The reason can be seen in the graph of [Thechim's curve](#), shown in [Figure 15.1](#); the more bugs there are, the harder it is to remove them. The more experience you have, the graph ([Figure 2.8](#)) shows that the later you find a bug, the more expensive it is to fix. So catching bugs early reduces overall effort.

Most good programmers today don't just test their software while writing it; they build their tests so that other people can run them months later and a dozen times wrong. This takes a little more time up front but makes

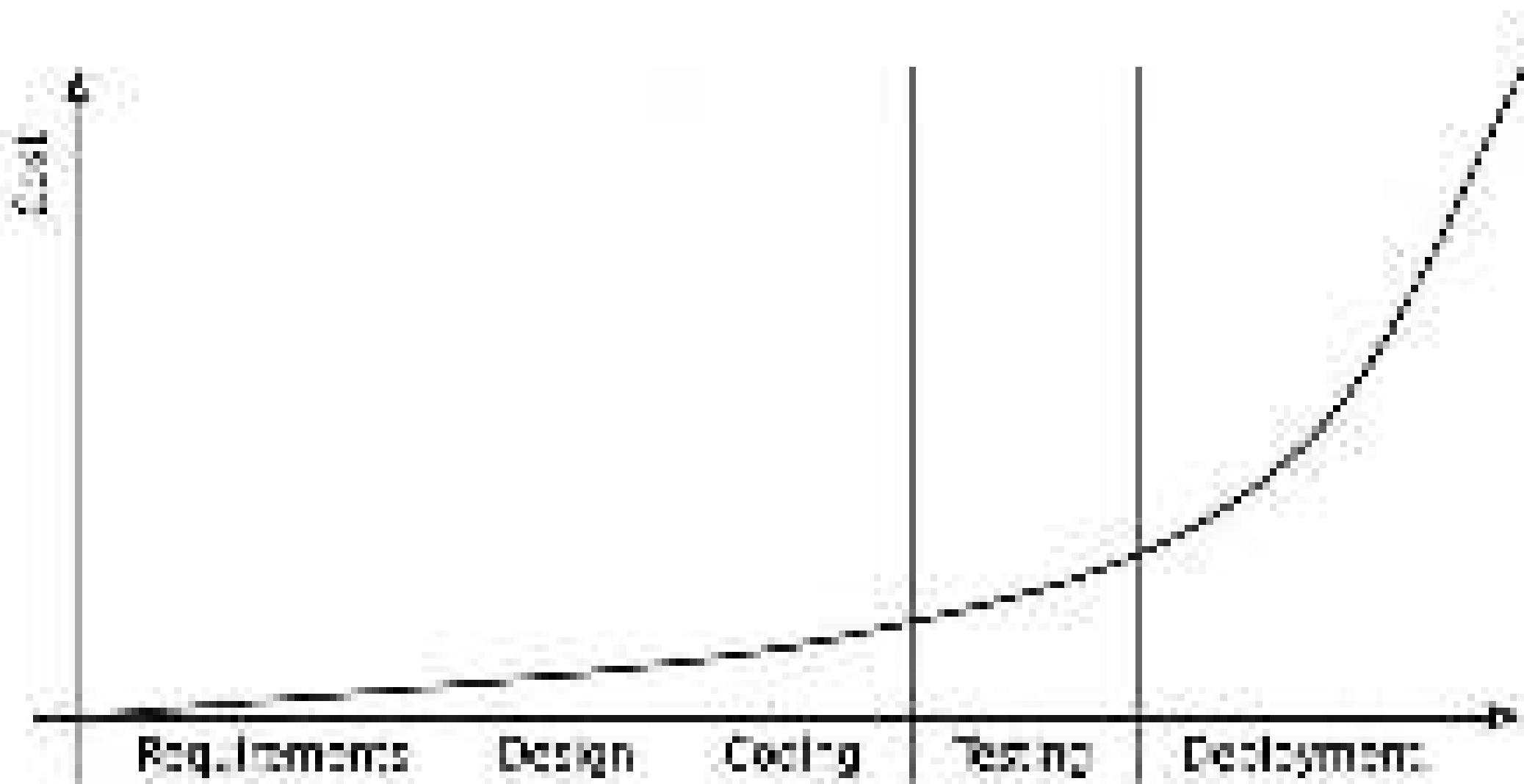


Figure 15—Nehru's curve: the later a bug is discovered, the more expensive it is to fix.

programmers more productive overall, since every hour invested in preventing bugs saves two, three, or ten frustrating hours tracking bugs down.

In Section 9.8, [Testing Your Code Sequentially](#), on page 109, you learned how to run tests using Python’s `unittest` module. As part of the function design process (see [Section 9.6, Designing Nine Functions: A Review](#), on page 471), you learned to include example calls on your function in the docstring. You can then use `unittest` to test those function calls and have it compare the output you expect with the actual output you get when it runs that function call.

## 15.2 Case Study: Testing above\_freezing

The first function that we’ll test is `above_freezing` from Section 9.8, [Test a Function Sequentially](#), on page 109:

```
def above_freezing(celsius):
    """number) -> bool
    Return True iff temperature celsius degrees is above freezing.

    <<< above_freezing(2, 2)
    True
    <<< above_freezing(-2)
    False
    <<<

    return celsius > 0
```

In this section, we run the example calls from the docstring using `check()`. But we're missing a test: what happens if the temperature is zero? In the next section, we'll write another version of this function that behaves differently at zero and will discuss how one running set of tests is incomplete.

## Choosing Test Cases for `above_freezing`

Before writing our test cases, we must decide which test cases to use. Function `above_freezing` takes one argument, a number. So for each test case, we need to choose the value of that argument. There are billions of numbers between from and we can't possibly test them all, so how do we decide which values to use? For `above_freezing`, there are two categories of numbers: values below freezing and values above freezing. We'll pick one value from each category to use in our cases.

Looking at it another way, this particular function returns a Boolean, so we need at least two tests: one that causes the function to return `True` and another that causes it to return `False`. In fact, that's what we already did in our example function call in the docstring.

In `above_freezing`, the first example call uses 5.5 as the value of the argument, and that value is above freezing so the function should return `True`. This test case represents the temperatures that are above freezing. We chose that value 5.5 since the balance of possible positive floating-point values may not all then would work just as well. For example, we could have used 100.6, 28, 32.25, or any other number greater than 0 to represent the "above freezing" category. The second example call uses -2, which represents the temperatures that are below freezing. Therefore, we could have used -1.943, -999, or any other value less than 0 to represent the "below freezing" category, but we chose to use -2. Again, our choice is arbitrary.

Are we missing any test case categories? Imagine that we had written our code using the `==` operator instead of the `>` operator:

```
def above_freezing(celcius):
    """Return True iff temperature celcius degrees is above freezing.

    <>> above_freezing(5.5)
    True
    <>> above_freezing(-2)
    False
    <>>
```

`return celcius >= 0`

both variants of the function produce the expected results for the two preceding examples, but the code is different from before, and it won't produce the same result in all cases. We neglected to test one category of inputs: temperatures at the freezing mark. These cases have the one bit of boundary work we often called boundary cases since they lie at the boundary between two different possible behaviors of the function (in this case, between temperatures above freezing and temperatures below freezing). Experience shows that boundary cases are much more likely to contain bugs than other cases, so it's always worth figuring out what they are and testing them.

Sometimes there are multiple boundary cases. For example, if we had a function that determined which state water was in—solid (ice), liquid, or gas—then the two boundary cases would be the melting point and the boiling point.

To summarize: the following table shows each category of inputs, the value we chose to represent that category, and the value that we expect the call on the function to return in that case:

| Test Case Description       | Assumed Value | Expected Return Value |
|-----------------------------|---------------|-----------------------|
| Temperatures above freezing | 73            | True                  |
| Temperatures below freezing | -2            | False                 |
| Temperatures at freezing    | 0             | False                 |

Table 24. Test Cases for `above_freezing`.

Now that all categories of inputs are covered, we need to run the third test. Running the third test in the Python shell reveals that the value returned by `above_freezing(0)` isn't `False`, which is what we expected:

```
...
```

```
above_freezing(0)
False
...
```

It took three test cases to cover all the categories of inputs for this function, but `0` isn't a magic number. The three cases had to be carefully chosen. If the three tests had all fallen into the same category (say, temperatures above freezing, 5, 30, and 300) they wouldn't have been sufficient. It's the quality of the cases that matters, not the quantity.

## Testing above freezing Using UnitTest

Once you decide which test cases are needed, you can run one of two approaches that you've learned about so far to actually test the code. The first is to call the functions and read the results yourself to see if they match

what you expected. The second is to run the functions from the `check()` using module code. The latter approach is preferable because the comparison of the actual value returned by the function to the value we expect to be returned is done by the program and not by a human, see IDEs built-in assert now below.

In this section, we'll introduce another of Python's modules, `unittest`. A "test" is code that just one function component of a program, like we did with `check()`. We'll use `unittest` to test each function in our module independently from the others. This approach contrasts with system testing, which looks at the behavior of the system as a whole, just as the external users will.

To follow along, see page 266, you learned how to write classes that absent code from where. Now you'll write test classes that inherit from `TestCase`. Over time, we'll close this functionality gap:

```
import unittest
import temperature

class TestTemperature(unittest.TestCase):
    """Test for Temperature above_freezing()"""

    def test_above_freezing(self):
        """Test a temperature that is above freezing."""
        expected = True
        actual = temperature.above_freezing(2)
        self.assertEqual(expected, actual,
                         "The temperature is above freezing.")

    def test_above_freezing_below(self):
        """Test a temperature that is below freezing."""
        expected = False
        actual = temperature.above_freezing(-5)
        self.assertEqual(expected, actual,
                         "The temperature is below freezing.")

    def test_above_freezing_at_freezing(self):
        """Test a temperature that is at freezing."""
        expected = False
        actual = temperature.above_freezing(0)
        self.assertEqual(expected, actual,
                         "The temperature is at the freezing mark.")

if __name__ == '__main__':
    unittest.main()
```

The name of our new class is `AddressBook`, and it's saved in the file `test_addressbook.py`. The class has three of its own methods, one per each test case. Each test case follows this pattern:

```
    def test_method(self, expected, actual):
    assertEqual(expected, actual)
    print("Success")
```

`expected` = Value we expect will be returned.  
`actual` = Value of the function being tested.  
`self.assertEqual(expected, actual)`:  
 Error message in case of failure.

In our test method, there is a call to `method.assertEqual()`, which has been inherited from class `unittest.TestCase`. To assert something is to claim that it is true; however, we are asserting that the expected value and the actual value should be equal. Method `assertEqual()` compares the two arguments (which are the expected return value and the actual return value from calling the function being tested) to see whether they are equal. If they aren't equal, the third argument, a string, is displayed as part of the failure message:

At the bottom of the file, the call to `unittest.main()` executes every method that begins with the prefix `test`:

When the program in `test_addressbook.py` is executed, the following results are produced:

---

Running tests in 0.000s.

OK

The first line of output has three dots, one dot per test method. A dot indicates that a test was run successfully—thus, the test case passed.

The summary after the dashed line tells you that it took 0.000s and ran three tests, that it took less than a millisecond to run, and that everything was successful [OK].

If our `sayHello` function were taking 0.1 seconds instead of 0.000s and our test drove freezing test, that program would return, instead of three passes (as indicated by the three dots), three passes and a failure:

.F.

---

Fatal error thrown freezing at [redacted] testAddressBook.py:  
 Test is terminating, there is no freezing

The stack trace recent calls [redacted]

File "test\_addressbook.py", line 22, in test\_freezing  
 The temperature is at the freezing mark."

`AssertionError: False != True : The temperature is at the freezing mark.`

Each of these is a `False`.

### FAILED (Failure)

The `+` indicates that a test case failed. The error message tells you that the failure happened in method `test_freezing_mark`. The error is an `AssertionError`, which indicates that when we asserted that the expected and actual values should be equal, we were wrong.

The expression `False != True` comes from our call on `assertEquals`: variable `celsius` was false, variable `mark` was true, and of course those aren't equal. Additionally, the string that was passed as the third argument to `assertEquals` is part of that error message: "The temperature is at the freezing mark."

Note that the three calls on `assertEquals` were placed in three separate methods. We could have put them all in the same method. But then `testMark` would have been considered a single test case. That is, when the module was run, we would see only one result: If any of the three calls on `assertEquals` failed, the entire test case would have failed. Only when all three passed would we see the expected dot.

As a rule, each test case you design should be implemented in its own test method.

Now that you've seen both `assert` and `assertFalse`, which should you use? We prefer `assertFalse` for several reasons:

- For large test suites, it's nice to have the testing code in a separate file rather than in a very long `setUp`.
- Each test case can be in a separate method, so the tests are independent of each other. With `assert`, the changes to one test will affect the subsequent ones, so more care must be taken to properly set up the objects for each iteration and easier to make sure they are independent.
- Because each test case is in a separate method, we can write a docstring that describes the test case so that other programmers understand how the test cases differ from each other.
- The third argument to `assertEquals` is a string that appears as part of the error message produced by a failed test, which is helpful for providing a better description of the test case. With `assertFalse`, there is no straightforward way to customize the error message.

## 15.3 Case Study: Testing running\_sum

In Section 16.2, Case Study: `Rockin about_frequencies` on page 216, we looked at a program that involved only immutable types. In this section, you'll learn how to test functions involving mutable types, like lists and dictionaries.

Suppose we need to write a function that modifies a list so that it contains a running sum of the values in it. For example, if the list is [1, 2, 3], the list should be mutated so that the first value is 1, the second value is the sum of the first two numbers, 1 + 2, and the third value is the sum of the first three numbers, 1 + 2 + 3, so we expect that the list [1, 2, 3] will be modified to be [1, 3, 6].

Following the function design recipe from [Section 5.8, Designing New Functions](#) A Recipe, on page 67, here is a file named `sums.py` that contains the completed function with one (passing) example test:

```
def running_sum(L):
    """List of numbers --> NoneType
```

Modify L so that it contains the running sum of its original items.

```
xxx L = [4, 6, 2, 8, 0]
xxx running_sum(L)
xxx L
[4, 4, 6, 10, 10]
```

```
for i in range(len(L)):
    L[i] = L[i] + L[i-1]
```

The structure of the test in the doctesting is different from what you've seen before, because there is no return value, just a sum return value. Writing a test that checks whether `None` is returned isn't enough to know whether the function call worked as expected. You also need to check whether the list passed to the function is mutated in the way you expect it to be. To do this, we follow these steps:

- Create a variable that refers to a list.
- Call the function, passing that variable as an argument to it.
- Check whether the list that the variable refers to was mutated correctly.

Following those steps, we created a variable, `xxx`, that refers to the list [4, 6, 2, 8, 0], called `running_sum()`, and `running_sum()` that mutates that list now to be [4, 4, 6, 10, 10].

Although this test case passes, it doesn't guarantee that the function will always work—and in fact there is a bug. In the next section, we'll design a set of test cases to more thoroughly test this function and discover the bug.

### Choosing Test Cases for running sum

Function `running_sum` has one parameter, which is a list of numbers. For our test cases, we need to decide with on the size of the list and the values of the items. For size, we should test with the empty list, a short list with one item and another with two items (the shortest case where two numbers interact), and a longer list with several items.

When passed either the empty list or a list of length one, the modified list should be the same as the original.

When passed a two-item list, the first number should be unchanged and the second number should be changed to be the sum of the two original numbers.

For longer lists, things get more interesting. The values can be negative, positive, or zero, so the resulting values might be bigger than, the same as, or less than they were originally. We'll think over what longer lists have four cases: all negative values, all zero, all positive values, and a mix of negative, zero, and positive values. The resulting tests are shown in the table:

| Test Case Description        | List Before      | List After        |
|------------------------------|------------------|-------------------|
| Empty list                   | []               | []                |
| One-item list                | [3]              | [3]               |
| Two-item list                | [2, 3]           | [2, 3]            |
| Multiple items, all negative | [-1, -3, -2, -4] | [-1, -6, -5, -12] |
| Multiple items, all zero     | [0, 0, 0, 0]     | [0, 0, 0, 0]      |
| Multiple items, all positive | [1, 2, 3, 5]     | [1, 6, 9, 15]     |
| Multiple items, mixed        | [4, 0, 2, 5, 0]  | [4, 4, 6, 10, 0]  |

Table 25 Test Cases for running sum

Now that we've decided on our test cases, the next step is to implement them using `unittest`.

### Testing running sum using `Unittest`

The test running sum will have the following code in `test_running_sum.py`:

```
running_sum([1, 2, 3, 4]) == [1, 3, 6, 10]
running_sum([-1, -3, -2, -4]) == [-1, -6, -5, -12]
running_sum([0, 0, 0, 0]) == [0, 0, 0, 0]
running_sum([1, 2, 3, 5]) == [1, 6, 9, 15]
running_sum([4, 0, 2, 5, 0]) == [4, 4, 6, 10, 0]
```

```

import unittest
import sum as sum

class TestSummingCumulative(unittest.TestCase):
    """Tests for sum()."""

    def test_summing_num_empty_list():
        """Test an empty list."""
        argument = []
        expected = 0
        sum_.summing_num(argument)
        self.assertEqual(expected, argument, "The list is empty.")

    def test_summing_num_one_item():
        """Test a one-item list."""
        argument = [5]
        expected = 5
        sum_.summing_num(argument)
        self.assertEqual(expected, argument, "The list contains one item.")

    def test_summing_num_two_items():
        """Test a two-item list."""
        argument = [2, 3]
        expected = 5
        sum_.summing_num(argument)
        self.assertEqual(expected, argument, "The list contains two items.")

    def test_summing_num_with_all_negative():
        """Test a list of negative values."""
        argument = [-1, -3, -5, -4]
        expected = -13
        sum_.summing_num(argument)
        self.assertEqual(expected, argument, "The list contains only negative values.")

    def test_summing_num_with_all_zeros():
        """Test a list of zeros."""
        argument = [0, 0, 0, 0]
        expected = 0
        sum_.summing_num(argument)
        self.assertEqual(expected, argument, "The list contains only zeros.")

    def test_summing_num_with_all_positives():
        """Test a list of positive values."""

```

```
argument = [4, 2, 2, 6]
expected = [4, 6, 8, 14]
assert running_sum(argument)
will.assertEqual(expected, argument)

    -> The list contains only positive values."}
```

```
def test_running_sum_with_nots(ss):
    """Test a list containing mixture of negative values, zeros and
    positive values."""
    argument = [-4, 0, 2, -5, 0]
    expected = [-4, 0, 2, 1, 1]
    assert running_sum(argument)
    will.assertEqual(expected, argument)

    -> The list contains a mixture of expected values. <0><0> &<0>
        + Unchecked values."
```

with(  
 assert\_raises):

Next we run the tests and see that only three of them pass (the empty list, a list with several zeros, and a list with a mixture of negative values, zeros, and positive values):

..TT.TT

---

```
FAIL: test_running_sum_with_nots (__main__.TestRunningSum)
Test a list of negative values.

Traceback (most recent call last):
  File "test_running_sum.py", line 36, in test_running_sum_with_nots
    "the list contains only negative values."
AssertionError: Lists differ: [-4, 0, 2, -5, 0] != [-4, -12, -14, -12]
```

First differing element 0:

```
-4
-5
```

```
+ [-4, -6, -9, -12]
+ [-5, -10, -15, -17] -> The list contains only negative values.
```

---

```
FAIL: test_running_sum_with_positives (__main__.TestRunningSum)
Test a list of positive values.
```

```
Traceback (most recent call last):
  File "test_running_sum.py", line 36, in test_running_sum_with_positives
    "the list contains only positive values."
AssertionError: Lists differ: [4, 6, 8, 14] != [12, 18, 22]
```

First differing element 0:

3

TH

+ [4, 6, 8, 10]

+ [10, 12, 15, 21] : The list contains only positive values.

---

FAIL: test\_running\_mean\_one\_item (\_\_main\_\_ TestRunning5)

Test a one-item list.

---

Traceback (most recent call last):File "test\_running\_mean.py", line 23, in test\_running\_mean\_one\_item  
 self.assertEqual(expected, argument, "The list contains one item")  
AssertionError: Lists differ: [5] != [10]

First differing element 0:

5

TH

[6]

+ [10] : the list contains one item

---

FAIL: test\_running\_mean\_two\_items (\_\_main\_\_ TestRunning5)

Test a two-item list.

---

Traceback (most recent call last):File "test\_running\_mean.py", line 23, in test\_running\_mean\_two\_items  
 self.assertEqual(expected, argument, "The list contains two items")  
AssertionError: Lists differ: [2, 7] != [3, 15]

First differing element 0:

2

T

+ [2, 7]

+ [7, 12] : The list contains two items

---

Run 7 tests in 0.002s

FAIL: (Failure)

The four that failed were a list with one item, a list with two items, a list with all negative values, and a list with all positive values. To find the bug, let's focus on the simplest test case, the single-item list.

---

FAIL: test\_running\_mean\_one\_item (\_\_main\_\_ TestRunning5)

Test a one-item list.

```

-----
```

check (most recent call last):
 File "c:\cygwin\home\campbell\python\exercises\code\runningsum.py", line 21, in test\_running\_sum\_and\_items
 self.assertEqual(result[expected\_index], expected\_value)
AssertionError: Lists differ: [5] != [10]
First differing element 0:
5
10

- [5]
- [10]: The list contains one item

For this test, the list argument was [5]. After the function call, we expected the list to be [5], but the list was supposed to become [10]. Looking back at the function definition of running sum, which takes in b, the for loop body executes the statement  $b[0] = b[1] - b[0]$ .  $b[1]$  refers to the last element of the list—the “ $-$ ” and  $b[1]$  refers to that same value. Banged  $b[1]$  shouldn't be changed, since the running sum of  $b[1]$  is simply  $b[1]$ .

Looking at the other three failing tests, the failure messages indicate that the first different elements are those at index 0. The same problem that we saw—the bug that item  $b[0]$  supports two values as well.

So how did CodeChef pass these tests? In those cases,  $b[1] + b[0]$  produced the same value that  $b[0]$  originally referred to. For example, for the list containing a mixture of values [4, 2, 2, 2, 2], the item at index 0 happened to be 4, so  $b[1] + b[0]$  evaluated to 4, and that matched  $b[0]$ 's original value. Interestingly, the simplest single-item test case revealed the problem while the more complex test cases that involved a list of multiple values hid it.

To fix the problem, we can adjust the for loop header to start the summation from index 1 rather than from index 0:

```

def running_sum(L):
    """List of numbers --> list of numbers
    Modifies L so that it contains the running sum of its original items.

    Examples:
    >>> L = [4, 0, 2, 2, 0]
    >>> running_sum(L)
    >>> L
    [4, 4, 6, 1, 1]
    """
    for i in range(1, len(L)):
        L[i] = L[i] + L[i-1]

```

When the tests run, they all return `True`:

```
.....
-----
Run 7 tests in 0.000s
```

`OK`

In the next section, you'll see some general guidelines for choosing test cases.

## 15.4 Choosing Test Cases

Having a set of tests that pass is good; it shows that your code does what it should in the situations you've thought of. However, for any large project there will be situations that don't occur to you. Tests can show the absence of many bugs, but it can't show that a program is fully correct.

It's important to make sure you have good *test coverage*: that your test cases cover important situations. In this section, we provide some heuristics that will help you think up with a fairly thoughtful set of test cases.

Now that you've seen two example sets of tests, we'll give you an overview of things to think about while you're developing tests for other functions. Some of them overlap and not all will apply in every situation, but they are all worth thinking about while you are figuring out what to test.

Think about *sets*. When a test involves a collection such as a list, string, dictionary, or file, you need to do the following:

- + Test the empty collection.
- + Test a collection with one item (i.e., a singleton).
- + Test a general case with several items.
- + Test the simplest interesting case, such as sorting a list containing `None` values.

Think about *relationships*. A relationship is a constraint between two things. Examples of relationships are completeness, even/odd, probability, and alphabetical/nonalphabetical. If a function deals with two or more different categories or situations, make sure you test all of them.

Think about *boundaries*. If a function behaves differently around a particular boundary or threshold, test exactly that boundary case.

Think about *order*. It's human behavior differently when values appear in different orders. Identify those orders and test each one of them. For the sorting example mentioned above, you'll want one test case where the items are in order and one where they are not.

If you carefully plan your test, make something to those ideas and your task passes the tests, there's a very good chance that it will work for all other cases as well. Over time you'll commit fewer and fewer errors. Whenever you find an error, figure out why it happened; as you repeatedly isolate them, you'll subsequently become more conscious of them. And that's really the whole point of developing *per*-quality: The more you do it, the less likely it is for problems to arise.

## 15.5 Hunting Bugs

Bugs are discovered through testing and through program use, although the latter is what good testing can help avoid. Regardless of how they are discovered, tracking down and eliminating bugs in your programs is part of every programmer's life. This section introduces some techniques that can make debugging more efficient and give you more time to do the things you'd rather be doing.

Debugging a program is like diagnosing a medical condition. To find the cause, you start by working backward from the symptoms (or, for a program, its incorrect behavior), then you come up with a solution and test it to make sure it actually fixes the problem.

At first, that's the right way to do it. Many beginners make the mistake of skipping the diagnosis stage and trying to cure the program by changing things at random. Renaming a variable or swapping the order in which two functions are defined might actually fix the program, but millions of such changes are possible. Trying them one after another in a particular order can be an inefficient waste of many, many hours.

Here are some rules for tracking down the cause of a problem:

1. **Understand what the program is supposed to do.** Sometimes this means doing the calculation by hand to see what the correct answer is. Other times it means reading the documentation (or the assignment handout) carefully or writing it down.
2. **Reproduce, follow.** You can debug bugs only when they go wrong, so find a few cases that misuse the program, all reliably. Once you have one, try to find a simpler way of doing this often provided *is simple* is a good check to allow you to fix the underlying problem.
3. **Attack and inspect.** Once you have a bug, fix the code. In general, try to find the first moment when something goes wrong. Examine the inputs to the function or block of code where the problem first becomes

stable. If these results are not what you expected, look at how they were created, and so on.

4. Change one thing at a time, given reason. Replacing random bits of code with different ones just might be responsible for your problem or unlikely to do much good. After all, you got it wrong the first time...! Each time you make a change, run to your test cases immediately.
5. Keep records. After working on a problem for an hour, you won't be able to remember the results of the tests you've run. Like any other scientist, you should keep records. Some programmers use a lab notebook; others keep a file open in an editor. Whatever works for you, make sure that when the time comes to seek help, you can tell your colleagues exactly what you've learned.

## 15.6 Bugs We've Put In Your Ear

By this chapter, you learned the following:

- Finding and fixing bugs early reduces overall effort.
- When choosing test cases, you should consider size, distribution, boundary cases, and more.
- To test your functions, you can write subclasses of `unittest.TestCase`. The advantages of using them include keeping the testing code separate from the code being tested, being able to keep the tests independent of one another, and being able to determine which individual test cases.
- To debug software, you have to know what it is supposed to do and be able to repeat the failure. Stippling the conditions that must be present is an effective way to narrow down the set of possible causes.

## 15.7 Exercises

Here are some exercises for you to try on your own. Solutions are available at <http://pragprog.com/titles/cj/exercises-programming>.

1. Your lab partner claims to have written a function that appends each value in a list with twice the preceding value (and the first value with 0). For example, if the list `[1, 2, 3]` is passed as an argument, the function is supposed to turn it into `[0, 2, 4]`. Here's the code:

```
def double_processing(values):
    """ Append of number, -- baseType
```

Replace each item in the list with twice the value of the

crosses zero, and replace the first one with 0.

```
def f(x, y, A):
    """Yank's algorithm"""
    if x < 0:
        return 0
    else:
        return x + f(y, A)
```

```
# values in D:
temp = returned()
returned() == 0
for x in range(1, len(returned())):
    returned() == 2 * temp
    temp = returned()
```

Although the example test passes, this code contains a bug. Write a set of unit tests to identify the bug. Explain what the bug in Yank's function is, and fix it.

3. Your job is to come up with tests for a function called `line_intersection`, which takes two lines as input and returns their intersection. More specifically:
  - Lines will be represented as pairs of distinct points, such as `[[0, 0], [0, 1]]`.
  - If the lines don't intersect, the function should return `None`.
  - If the lines intersect in one point, the function must return the point of intersection, such as `(0.5, 1.2)`.
  - If the lines are coincident (that is, lie on top of each other), the function returns the first argument (that is, a line).

What are the six most informative test cases you can think of? That is, if you were allowed to run only six tests, which would tell you the most about whether the function was implemented correctly?

Write out the inputs and expected outputs of these six tests, and explain why you would choose them.

4. Using `utest`, write four tests for a function called `all_subsets` in a module called `bs_subsets` that takes a string as its input and returns the set of all nonempty substrings that start with the first character. For example, given the string "test" as input, `all_subsets` would return the set `{'t', 'te', 'tes', 'test'}`.
4. Using `utest`, write the five most informative tests you can think of for a function called `is_prime` in a module called `is_prime.py`; this takes a list of

input to the input and return the list if they are sorted in non-decreasing order (in opposition to strictly increasing order, because of the possibility of duplicate values), and take otherwise.

6. The following function is broken. Use doctests to describe what it's supposed to do:

```
def sort_and_reduce(values):
    """ (list) -> AnyType
    Return the minimum and maximum value from values.
    """
    min = None
    max = None
    for value in values:
        if value < min:
            min = value
        if value > max:
            max = value
    print("The minimum value is", min, "and the maximum value is", max)
    print("The maximum value is", max, "and the minimum value is", min)
```

What does it actually do? What line(s) do you need to change to fix it?

7. Suppose you have a data set of survey results where respondents can potentially give their age. Missing values are read as `None`. Here is a function that computes the average age from that list:

```
def average(values):
    """ (list of int or None) -> float
```

Starts the average at the average of values. New items in values are added, and they are not counted toward the average.

```
>>> average([25, 30])
25.5
>>> average([None, 20, 30])
25.5
>>>
```

```
count = 0 # The number of values seen so far
total = 0 # The sum of the values seen so far
for value in values:
    if value is not None:
        total += value
        count += 1
```

`count == 0`

```
    return total / count
```

Unfortunately it does not work as expected:

```
>>> import test_average  
>>> test_average.average([None, 30, 20])  
30.000000000000002
```

- a. Using unittest, write a set of tests for function `average` in a module called `test_average.py`. The tests should cover cases involving lists with and without missing values.
- b. Modify function `average` so it correctly handles missing values and *passes all of your tests*.

# Creating Graphical User Interfaces

Most of the programs in previous chapters are not interactive. Once launched, they run in the computer, waiting, giving us a chance, to share them or provide new input. The few that do communicate with us do so through the kind of text-only command-line user interface, or CLI, that would have already been considered old-fashioned in the early 1990s.

As you already know, most modern programs interact with users via a graphical user interface, or GUI, which is made up of windows, mouse buttons, and so on. In this chapter, we will show you how to build simple GUIs using a Python module called `tkinter`. Along the way, we will introduce a different way of structuring programs called event-driven programming. A traditionally structured program usually has control over what happens when, but an event-driven program must be able to respond to unpredictable moments.

There is [several toolkits](#) you can use to build GUIs in Python. It is the only one that comes with a standard Python installation.

## 16.1 Using Module `Tkinter`

Every `Tkinter` program consists of three things:

- Windows, buttons, scrollbars, text menus, and other widgets—anything that you can see on the computer screen (Generally, the term *widget* means any user interface; in programming, it's shorter “window gadget.”)
- Modules, functions, and classes that manage the data that is being shown in the GUI—you are familiar with these; they are the tools you've seen so far in this book.
- An event manager that looks for events such as mouse clicks and keystrokes and reacts to these events by calling event handler functions.

Here is a small but complete Tk root window:

```
import tkinter
window = tkinter.Tk()
window.mainloop()
```

This is a class that represents the representation of a tkinter GUI. This root window's mainloop method handles all the events for the GUI, so it's important to create only one instance of it.

Here is the resulting GUI:



The root window is initially empty; you'll see in the next section how to add widgets to it. If the window on the screen is closed, the window object is destroyed though we can create a new root window by calling Tk again. All of the applications we will create here only contain windows, but additional windows can be created using the `Toplevel` widget.

The call to `mainloop` doesn't exit until the window is destroyed (which happens when you click the appropriate window title bar of the window), so the code following that call won't be executed until later:

```
import tkinter
window = tkinter.Tk()
window.mainloop()
print("Finally run!")
```

When you try this code, you'll see that the call to `print` isn't yet evaluated until after the window is destroyed. That means that if you want to make changes to the GUI after you have called `mainloop`, you need to do it in an event handling function.

The following table gives a list of some of the available Tk variables:

| Widget      | Description                                         |
|-------------|-----------------------------------------------------|
| Button      | A clickable button.                                 |
| Canvas      | An area used for drawing or displaying images.      |
| Checkbutton | A checkable box that can be selected or unselected. |
| Entry       | A single line text field that the user can type in. |
| Frame       | A container for widgets.                            |
| Label       | A single line display for text.                     |
| Listbox     | A drop down list that the user can select from.     |
| Menu        | A drop down menu.                                   |
| Message     | A multiline display for text.                       |
| MenuItem    | An item in a drop down menu.                        |
| Text        | A多-line text field that the user can type in.       |
| Toplevel    | An additional window.                               |

Table 26—Other Widgets

## 16.2 Building a Basic GUI

Labels are widgets that are used to display short pieces of text. Here we create a label that belongs to the root window—the parent widget—and we specify the text to be displayed by assigning it to the `text` parameter:

```
import tkinter
```

```
window = tkinter.Tk()
label = tkinter.Label(window, text="This is our label")
label.pack()
```

```
window.mainloop()
```

Here is the resulting GUI:



Method call `label.pack()` is crucial. Each widget has a method called `pack` that places it in its parent widget and then tells the parent to resize itself if necessary. If we try to run this code but omit `label.pack()`, the label would be displayed but will be displayed improperly.

Labels display text. Often, applications will want to update a label's text as the program runs to show things like the name of a file or the time of day. One way to do this is simply to assign a new value to the `label['text']` using method `config`:

```
import tkinter

window = tkinter.Tk()
label = tkinter.Label(window, text="First Text")
label.pack()
label.config(text="Second Text")

window.mainloop()
```

Run the previous code on live shell line from the Python shell to see how the label changes. This code will not display the window at all if you run it as a program because we haven't called method `mainloop()`.

## Using Mutable Variables with Widgets

Suppose you want to display a string, such as the current time or a score. In a game, in several places in a GUI—the application's status bar, some dying books, and so on. Calling method `config` on each widget every time there is new information isn't ideal, but as the application grows, we face the dilemma that will longer to update at least one of the widgets that displaying the string. What we really want is a **StringVar** “factory” which widgets care about its value and can alter them itself when that value changes.

Python's strings, integers, floating-point numbers, and booleans are immutable, so `tkinter` does provide one class for each of the immutable types. `StringVar`, `IntVar`, `FloatVar` for float, and `BooleanVar` for bool. (The case of the word `int` is intentional; it is short for “double precision floating-point number.”) These mutable types can be used instead of the immutable ones; here we show how to use a `StringVar` instead of a str:

```
import tkinter

number = tkinter.IntVar()
data = tkinter.StringVar()
data.set("Text to display")
label = tkinter.Label(window, textvariable=data)
label.pack()

window.mainloop()
```

Note that this time we assign to the `textvariable` parameter of the `Label` rather than the `text` parameter.

The values in `StringVar` containers are set and retrieved using the methods `set` and `get`. Whenever a `set` method is called, it tells the `label` and any other widgets it has been assigned to that it's time to update the GUI.

There is one small trap here: for now, because of the way `tkinter` is structured, you cannot create a `StringVar` in any other variable or module until you have created the root Tk window.

## Grouping Widgets with the Frame Type

A `tkinter.Frame` is a container, much like the root window is a container. Frames are not directly visible in the screen; instead, they are used to contain other widgets. The following code creates a frame, puts it inside a window, and then adds three labels to the frame:

```
import tkinter

window = tkinter.Tk()
frame = tkinter.Frame(window)
frame.pack()
first = tkinter.Label(frame, text="First Label")
first.pack()
second = tkinter.Label(frame, text="Second Label")
second.pack()
third = tkinter.Label(frame, text="Third Label")
third.pack()

window.mainloop()
```

Note that we call `pack()` every widget. If we omit one of these calls, that widget will not be displayed.

Here is the resulting GUI:



In this particular case, putting the three labels in a frame is no different than simply putting the labels directly into the root window. However, with a more complicated GUI, you can use multiple frames to format the window's content and layout.

Here is an example with the same three labels but with two frames instead of one. The second frame has a visible border around it.

```
import tkinter

window = tkinter.Tk()
frame = tkinter.Frame(window)
frame.pack()
frame2 = tkinter.Frame(window, borderwidth=4, relief='sunken')
frame2.pack()
first = tkinter.Label(frame, text="First Label")
first.pack()
second = tkinter.Label(frame2, text="Second Label")
second.pack()
third = tkinter.Label(frame, text="Third Label")
third.pack()

window.mainloop()
```

```

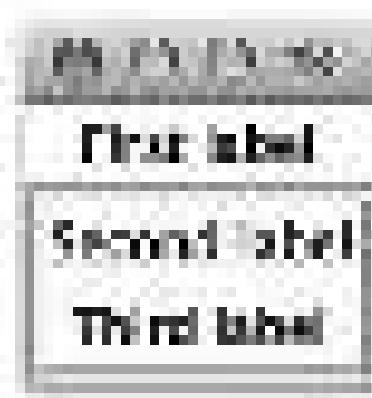
second = tkinter.Label(frame, text='Second Label')
second.pack()
third = tkinter.Label(frame, text='Third Label')
third.pack()

window.mainloop()

```

We specify the border width using the `borderwidth` keyword argument (0 is the default) and the border style using `relief` (`FLAT` is the default). The other border styles are `SOLID`, `RIDGE`, `GROOVE`, and `RAISED`.

Here is the resulting GUI:



## Getting Information from the User with the Entry Type

Our next GUI uses text input. The simplest one is `Entry`, which allows for a single line of text. If we associate a `StringVar` with the `Entry`, then whenever a user types anything into that `Entry`, the `StringVar`'s value will automatically be updated to the contents of the `Entry`.

Here's an example that associates a single `StringVar` with both a label and an `Entry`. When the user enters text in the `Entry`, the `StringVar`'s contents will change. This will cause the label to be updated, and so the label will display whatever is currently in the `Entry`.

```

import tkinter
window = tkinter.Tk()

frame = tkinter.Frame(window)
frame.pack()
var = tkinter.StringVar()
label = tkinter.Label(frame, textvariable=var)
label.pack()
entry = tkinter.Entry(frame, textvariable=var)
entry.pack()
window.mainloop()

```

Here is the resulting GUI:



## 10.3 Models, Views, and Controllers, Oh My!

Using a string to connect a text-entry box and a label is the first step toward separating models. How do we represent the data? views (How do we display the data?), and controllers (How do we modify the data?). This is the key to building better GUIs (as well as many other kinds of applications). The MVC design helps separate the parts of an application, which will make the application easier to implement and modify. The main goal of this design is to keep the representation of the data separate from the parts of the program that the user interacts with. This way, it's easier to make changes to the GUI code without affecting the code that manipulates the data.

As its name suggests, a view is something that displays information to the user, like Label. Many views, like Entry, also accept input, which gets displayed immediately. The key is that they don't do anything else; they don't calculate average temperature, move robot arms, or do any other calculations.

Models, on the other hand, store data like a piece of text or the current orientation of a microscope. They also don't do calculations; their job is simply to keep track of the application's current state (and, in some cases, to write that state to a file or database and reload it later).

Controllers are the pieces that convert user input into calls on functions in the model that manipulate the data. The controller is what decides whether two file sequences match well enough to be colored green or whether someone is allowed to overwrite an old version file. Controllers may update an application's models, which in turn can output changes back to a view.

The following code shows what all of this looks like in practice. Here the model is kept track of by variable counter, which refers to an integer so that the view will update itself automatically. The controller is function click(), which updates the model whenever a button is clicked. Four objects make up the view: the root window, a Frame, a Label that shows the current value of counter, and a button that the user can click to increment the counter's value:

```
import tkinter

# The controller
def click():
    counter = tkinter.Counter.get() + 1

    if __name__ == '__main__':
        window = tkWindow.Tk()
        # The model
        counter = tkCounter.Counter()
        counter.set(0)

        # The view
        frame = tkFrame.Frame(window)
        frame.pack()

        label = tkLabel.Label(frame, text=counter)
        label.pack()

        button = tkButton.Button(frame, text='Click me!', command=click)
        button.pack()

        window.mainloop()
```

```

# The class
class CounterFrame(Frame):
    def __init__(self, master):
        Frame.__init__(self, master)

        self.button = tkinter.Button(self, text="Click!", command=self.click)
        self.button.pack()

        self.label = tkinter.Label(self, text="0")
        self.label.pack()

    # Start the necessary
    # window.mainloop()

```

The first two arguments used to construct the `Button` should be familiar by now. The third argument tells it to call `function` each time the user presses the button. This makes use of the fact that in Python a function is just another kind of object and can be passed as an argument like anything else.

Function `click` in the previous code does not have any parameters but uses variable `counter`, which is defined outside the function. Variables like this are called *global variables*, and their use should be avoided, since they make programs hard to understand. It would be better to pass any variables the function needs into it as parameters. We can't do this using the tools we have seen so far, because the functions that our buttons can call must not have any parameters. We will show you one way to avoid using global variables in the next section.

## Using Lambdas

The simple counter GUI shown earlier does what it's supposed to, but there is room for improvement. For example, suppose we want to be able to know the counter's value as well as reset it.

Using only the tools we have seen so far, we could add another button and another controller function like this:

```

import tkinter

window = tkinter.Tk()

# The code!
counter = tkinter.IntVar()
counter.set(0)

# Two controller func.
def click_up():
    counter.set(counter.get() + 1)

def click_down():
    counter.set(counter.get() - 1)

```

This seems a little cleaner though. Functions `click_x` and `click_y` are doing almost the same thing; maybe we might be able to combine them even more. While we're at it, we'll pass `click_x` into the function `expit` like rather than using it as a global variable:

```
def calculate():
    number = 0
    for i in range(1, 1000000):
        number += i * i

    return number
```

The problem with this is that you can't tell the `push` function what arguments to pass into the function, since we can't provide any arguments for the functions assigned to the buttons' `onPress` keyword arguments when creating those buttons. (We cannot read variables—so we don't know how many arguments our functions require or what values to pass in for them.) For that reason, it requires that the controller functions triggered by buttons and other widgets take zero arguments so they can all be called the same way. It is much better here to take the two-argument function we want to use and turn it into one that needs no arguments at all.

You could do this by writing a simple `__str__` function:

```
def click_up():
    clickcount = 1
```

This will get us back to two nearly identical functions that only use global variables. A better way is to use a `Symbol` function, which allows us to create a `Symbol` function without ever touching `Symbol`. Here's a very simple example:

```
>>> lambda: 3
<function <lambda> at 0x1000000000000000>
>>> f = lambda: x + 1
3
```

The expression `lambda: 3` on the first line creates a named or function that always returns the number 3. The second expression creates this function and immediately calls it, which has the same effect as this:

```
>>> def f():
...     return 3
...
>>> f()
3
```

However, the lambda form does not create a new variable or change an existing one. Finally, lambda functions can take arguments, just like other functions:

```
>>> f = lambda: x * 2 + 1(2)
6
```

### Why Lambdas?

The name `lambda` refers to a system developed by Alonzo Church, a mathematician, logician, and philosopher, for specifying functions in the definition and application of type theory developed in the 1930s by Alonzo Church and Stephen Kleene.

So how does this help us with GUIs? The answer is that it lets us write the callback function to handle different buttons in a general way and then simply call to that function when needed as needed. Here's the two-button GUI code again using lambda functions:

```
import tkinter

window = tkinter.Tk()

# The mode!
counter = tkinter.IntVar()
counter.set(0)

# General controller
def onClick(button):
    var = button.get() + counter.get()
    counter.set(var)

# The screen
frame = tkinter.Frame(window)
frame.pack()
```

```

button = tkinter.Button(frame, text='Up', command=button_up, clickcounter=0)
button.pack()

button = tkinter.Button(frame, text='Down', command=button_down, clickcounter=0)
button.pack()

label = tkinter.Label(frame, textvariable=counter)
label.pack()

window.mainloop()

```

This code creates a zero argument `button` function to pass into each button just where it's needed. These functions themselves then pass the right values into `click`. This is cleaner than the preceding code, because the function definitions are enclosed in the cell that uses them—there is no need to clutter the GUI with little functions that are used only in one place.

Now, however, there is a very bad idea to repeat the same function several times in different places—if you do this, the odds are very high that you will one day want to change them all but will take ten or more. If you find yourself wanting to do this, remember: do `def`, so that the function is defined only once:

## 16.4 Customizing the Visual Style

Every windowing system has its own look-and-feel—superiority in corners, particular colors and so on. In this section, we'll see how to change the appearance of GUI widgets to make applications look more distinctive.

A note of caution before we begin: the default style on most windowing systems has been designed especially for trained graphic design and human-computer interaction. The odds are that any visual changes you make will make things worse, not better. In particular, be careful about color because most of the main population has some degree of color blindness and font size: many people, particularly the elderly, cannot read small text.

### Changing Fonts

Let's start by changing the size, weight (font), and family of the font used to display text. To specify the size, we provide the `weight` as an integer in pixels. We can also set the `weight` to either bold (or normal) and the `slant` to either italic (italicized) or roman (not italicized).

The font families we can use depend on what system the program is running on. Common families include Times, Courier, and Helvetica, but dozens of others are usually available. One note of caution though: if you choose an

unusual font, people running your program on other computers might not have it, so your GUI might appear different than you'd like to. Every operating system has a default font that will be used if the requested font isn't installed.

The following sets the font of a button to be 14 point, bold, black, and Courier:

```
import tkinter
```

```
window = tkinter.Tk()
button = tkinter.Button(window, "Press me!")
button['font'] = ('courier', 14, 'bold', 'black')
button.pack()
window.mainloop()
```

Here is the resulting GUI:



Using this technique, you can set the font of any widget that displays text.

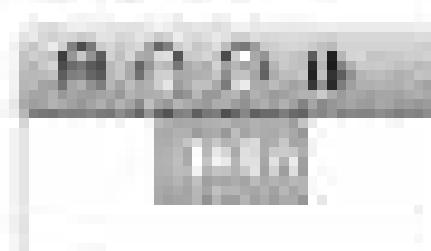
### Changing Colors

Almost all foreground colors can be set using the `fg` and `b` keyword arguments, respectively. As the following code shows, we can set either of these to a standard color by specifying the color's name, such as white, black, red, green, blue, cyan, yellow, or magenta.

```
import tkinter
```

```
window = tkinter.Tk()
button = tkinter.Button(window, fg='red', bg='green')
button['font'] = ('courier', 14, 'bold', 'black')
button.pack()
window.mainloop()
```

Here is the resulting GUI:



As you can see, white text on a bright green background is not particularly readable.

We can choose more colors by specifying them using the RGB color model. RGB is an abbreviation for “red, green, blue”; it turns out that every color

can be created using different amounts of those three colors. The amount of each color is usually specified by a number between 0 and 255 (inclusive).

These numbers are conventionally written in hexadecimal (base 16) notation; the best way to understand them is to play with them. Base 10 (or the digits 0 through 9) base 16 uses those ten digits plus another six: A, B, C, D, E, and F. In base 16, the number 255 is written FF.

The following code picks up this task by updating a pair of text boxes with the color specified by the red, green, and blue values entered in the text boxes; then, any two hex-digitals for the RGB value and click the Update button:

```
import tkinter
def change(widget, colors):
    # Update the foreground color of a widget to show the RGB color value
    # stored in a dictionary with keys 'red', 'green', and 'blue'. Some
    # 'not' check the color value.
    pass
```

```
new_val = '#'
for name in ['red', 'green', 'blue']:
    new_val += colorvar.get()
    entry1['text'] = new_val

# Create the application
window = tkinter.Tk()
frame = tkinter.Frame(window)
frame.pack()

# Set up text entry widgets for red, green, and blue, using the
# given initial colors from a dictionary for defaults.
colors = {}
for name, val in [('red', '#FFFF00'),
                  ('green', '#00FFFF'),
                  ('blue', '#0000FF')]:
    colors[name] = tkColorChooser.askcolor()
    colors[name].add('00')
    entry = tkinter.Entry(frame, textvariable=colors[name], bg='white',
                          fg='black')
    entry.pack()

# Display the current color
current = tkinter.Label(frame, text='          ', bg='white')
current.pack()

# Click the button to trigger an update.
update = tkinter.Button(frame, text='Update')
update.pack()
update.bind('<Button-1>', change)

# Run the application
window.mainloop()
```

This is the most complicated GUI we have seen so far, but it can be understood by breaking it down into a model, some views, and a controller. The model is three `StringVar`s that store the hexdecimal strings representing the current red, green, and blue components of the color to display. These three variables are kept in a dictionary indexed by name for easy access. The controller is `function_change`, which concatenates the strings to create an RGB code and applies that color to the background of a `widget`. The views are the entry boxes for the value components, the label that displays the current color, and the button that tells the GUI to update itself.

This program works, but notice the GUI has very *stealthy*: It's annoying to have to click the update button, and if a user ever types anything other than a two-digit hexdecimal value into any of the text boxes, it results in an error. The exercises will ask you to modify both the appearance and the structure of this program.

## Laying Out the Widgets

One of the things that makes the color picker GUI ugly is the fact that everything is arranged top to bottom. *After all*, this *isn't* by default how we can usually count up with something like this:

To see how, let's revisit the example from *Getting Information from the User with the Entry Type*, on page 302, placing the label and button horizontally. We'll start by doing the preceding `__init__` method back:

```
import tkinter

window = tkinter.Tk()
frame = tkinter.Frame(window)
frame.pack()
label = tkinter.Label(frame, text="Name:")
label.pack(side='left')
entry = tkinter.Entry(frame)
entry.pack(side='left')

window.mainloop()
```

Here is the resulting GUI:



**Sizing and Layout** In Tk's `grid` layout, the leftmost part of the label is to be placed next to the left edge of the frame, and then the leftmost part of the entry field is placed next to the right edge of the label—in short, that width is to be

packed instead (but left-align). We could equally well pack, but the right, top, or bottom edges, as we could mix packages. (though that can quickly become confusing)

For even more control of our window layout, we can use a different layout manager called `grid`. As its name implies, it creates windows and frames as grids of rows and columns. To add the `entry` to the `window`, we call `grid` instead of `pack`. Do not call both on the same widget; this conflicts with each other. The `grid` will take several parameters, as shown in Table 27, `grid` Parameters.

| Parameter               | Description                                                                   |
|-------------------------|-------------------------------------------------------------------------------|
| <code>row</code>        | The number of the row to insert the widget into—row numbers begin at 0.       |
| <code>column</code>     | The number of the column to insert the widget into—column numbers begin at 0. |
| <code>rowspan</code>    | The number of rows the widget occupies. The default value is 1.               |
| <code>columnspan</code> | The number of columns the widget occupies. The default value is 1.            |

Table 27—`grid` Parameters

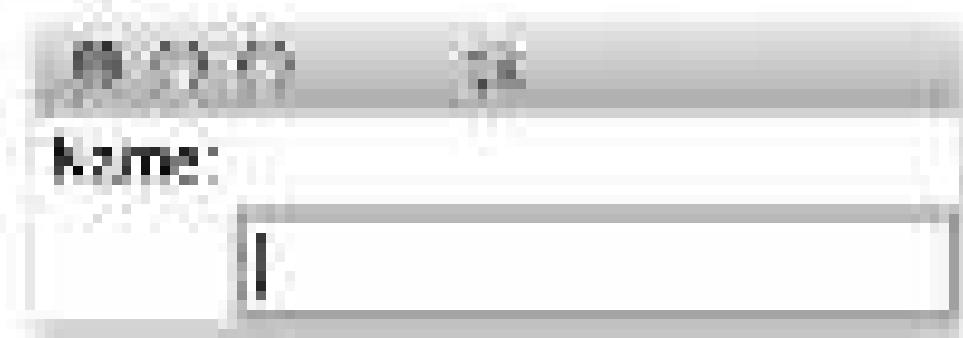
In the following code, we place the label in the upper-left (row 0, column 0) and the entry field in the lower-right (row 1, column 1).

```
import tkinter
```

```
window = tkinter.Tk()
frame = tkinter.Frame(window)
frame.pack()
label = tkinter.Label(frame, text="Name:")
label.grid(row=0, column=0)
entry = tkinter.Entry(frame)
entry.grid(row=1, column=1)
```

```
window.mainloop()
```

Here is the resulting GUI; as you can see, this leaves the bottom-left and upper-right corners empty:



## 16.5 Introducing a Few More Widgets

To end this chapter, we will look at a few more commonly used widgets.

### Using Text

The `Entry` widget that we have been using since the start of this chapter allows for only a single line of text. If we want multiple lines of text, we use the `Text` widget instead, as shown here:

```
import tkinter

def createText():
    text.insert('1.0', 'INSERT\n')

window = tkinter.Tk()
frame = tkinter.Frame(window)
frame.pack()

text = tkinter.Text(frame, height=2, width=20)
text.pack()

button = tkinter.Button(frame, text='Add', command=lambda: createText())
button.pack()

window.mainloop()
```

Here's the resulting GUI:



This provides a much richer set of methods than the other widgets we have seen so far. We can embed images in the text area, put images, select particular lines, and so on. The exercises will give you a chance to explore its capabilities.

### Using Checkbuttons

`Checkbutton`, often called `checkbox`, has two states: on and off. When a user clicks a `Checkbutton`, the state changes. We use a `Boolean` variable to keep track of the user's selection. Typically, an `int` variable is used, and the values 1 and 0 indicate on and off, respectively. In the following code, we

use. These class definitions provide a simpler value picker, and we are interested only in changing the configuration of a widget after it has been created.

```
import tkinter
```

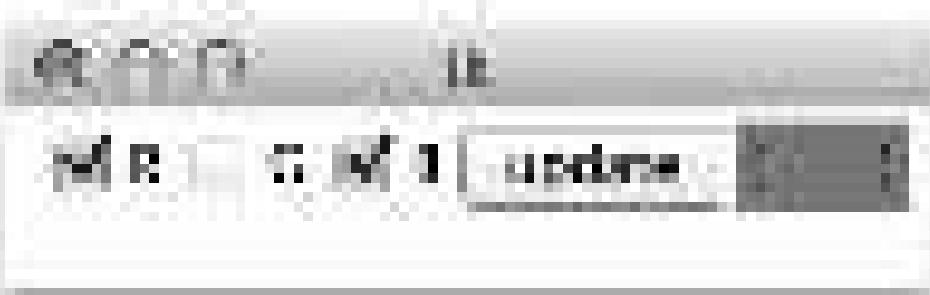
```
master = tkinter.Tk()
frame = tkinter.Frame(master)
frame.pack()
red = tkinter.IntVar()
green = tkinter.IntVar()
blue = tkinter.IntVar()

for name, var in [('R', red), ('G', green), ('B', blue)]:
    check = tkinter.Checkbutton(frame, text=name, variable=var)
    check.pack(side='left')

def recolor(widget, r, g, b):
    color = '#%02x%02x%02x' % (r.get(), g.get(), b.get())
    widget.config(bg=color)

label = tkinter.Label(frame, text='Color')
button = tkinter.Button(frame, text='Update',
                        command=lambda: recolor(label, red, green, blue))
button.pack(side='left')
label.pack(side='left')
master.mainloop()
```

Run this mailing GUI:



## Using Menu

The best candidate will look at its menu.

The following code uses this to create a simple new editor:

```
import tkinter
import tkinter.ttk as tk
import tkinter.dialog as dialog

def new(root, text):
    data = text.get('0.0', '1.0')
    filename = dialog.asksaveasfilename()
    parent=root
    filer=tkFileDialog('Text', 'Text')
```

```

    title='Save as...')

root = tk.Tk()
root.title('TextEditor')
root.minsize(300, 200)

def quit():
    root.destroy()

window = tk.Toplevel(root)
text = tk.Text(window)
text.pack()

menubar = tk.Menu(root)
filemenu = tk.Menu(menubar)
filemenu.add_command(label='Save', command=lambda: treeview.write('test.txt'))
filemenu.add_command(label='Quit', command=lambda: root.destroy())

menubar.add_cascade(label='File', menu=filemenu)
root.config(menu=menubar)

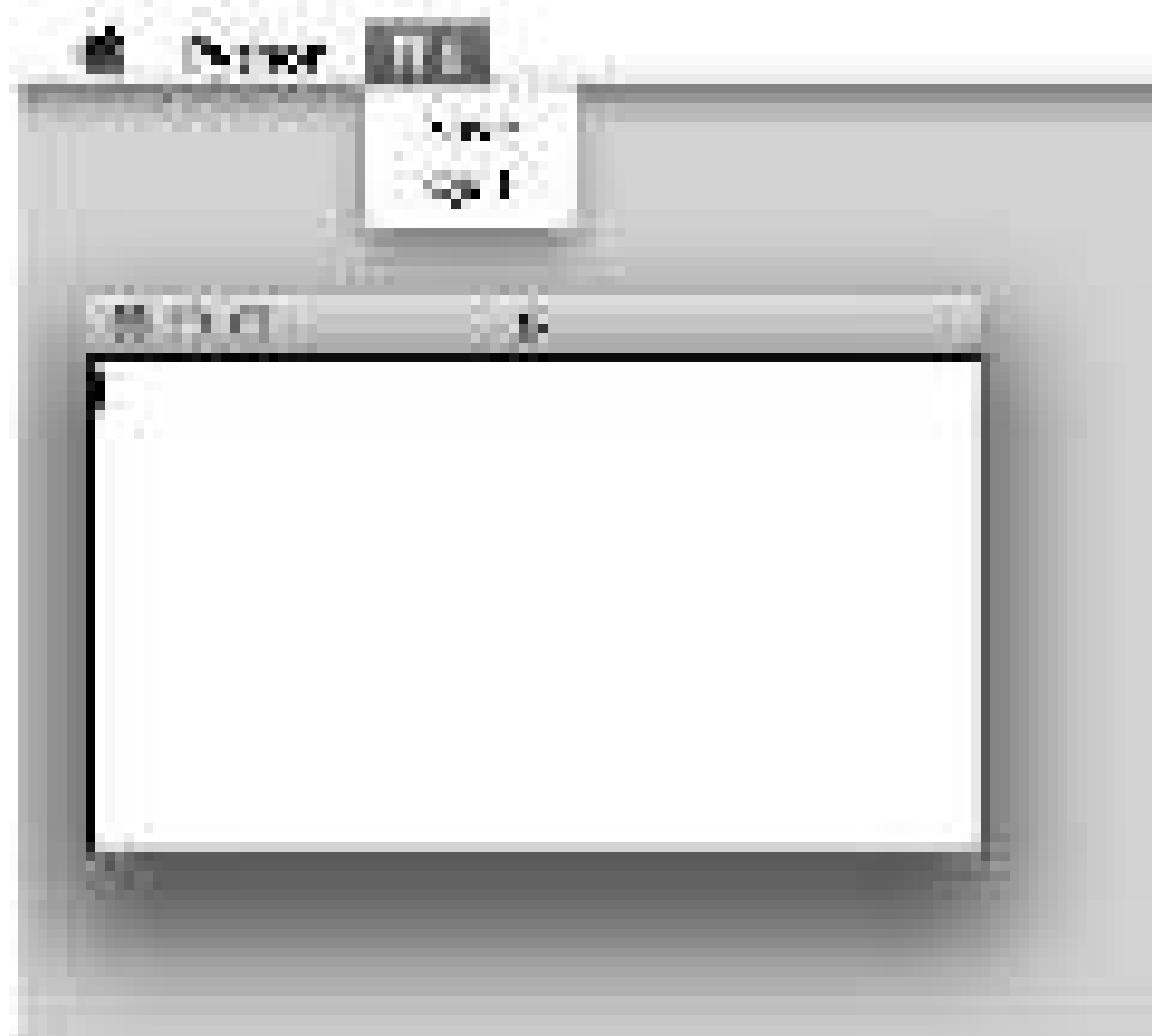
root.mainloop()

```

The program begins by creating `tk.Tk()`, `root`, which gives the container of a text widget and root, which houses the application. Function `quit()` uses `destroy()` to create a standard "Save as..." dialog box, which will prompt the user for the name of a text file.

After creating and packing the text widget, the program creates a menu bar, which is the horizontal bar into which we can put one or more menus. It then creates a file menu and adds two menu items: Save and Quit. We then add the file menu to the menu bar and run `mainloop()`.

Here is the resulting GUI:



## 10.6 Object-Oriented GUIs

The GUIs we have built so far have not been particularly well-structured. Most of the code to construct them has not been modularized in functions, and they have relied on global variables. We can get away with this for very small examples, but, if we try to build larger applications this way, they will be difficult to understand and debug.

For this reason, almost all real GUIs are built using classes and objects that combine models, views, and controllers together in one tidy package. In the example shown next, for example, the application's model is a member variable of class Counter, created using `self.var`, and its controllers are the methods `up()` and `down()`.

```
import tkinter

class Counter:
    """A simple counter GUI using object-oriented programming."""
    def __init__(self, parent):
        """Create the GUI.

        @param parent: parent window
        """
        self.parent = parent
        self.frame = Frame(parent)
        self.frame.pack()

        # Model
        self.var = StringVar()
        self.var.set('0')

        # Label displaying current state.
        self.label = tkinter.Label(self.frame, textvariable=self.var)
        self.label.pack()

        # Buttons to control application.
        self.up = tkinter.Button(self.frame, text='up', command=self.on_up)
        self.up.pack(side='left')

        self.right = tkinter.Button(self.frame, text='right', command=self.on_right)
        self.right.pack(side='left')

    def up(self):
        """Handle click on 'up' button."""
        self.var.set(str(int(self.var.get()) + 1))

    def right(self):
        """Handle click on 'right' button."""
        self.var.set(str(int(self.var.get()) + 1))
```

```

def quit_if_clicked():
    """Handle a click on a quit button.

    self.parent.destroy()
    if root == self.root:
        window = tkinter.Tk()
        app = CounterWindow()
        window.mainloop()

```

## 16.7 Keeping the Concepts from Being a GUI Mess

In this chapter, you learned the following:

- Most modern programs provide a graphical user interface (GUI) for displaying information and interacting with users. GUIs are built out of widgets, such as buttons, sliders, and text panels; all modern programming languages provide an easy-to-use GUI toolkit.
- Unlike command-line programs, GUI applications are usually event-driven. In other words, they react to events such as keystrokes and mouse clicks when and as they occur.
- Experience shows that GUIs should be built using the model-view-controller pattern. The model is the data being manipulated; the view displays the current state of the data and gathers input from the user, while the controller decides what to do next.
- Lambda expressions create functions that have no names. These are often used to define the actions other widgets should take when users provide input without requiring global variables.
- Designing usable GUIs is as challenging a task as designing software. Being good at the latter doesn't guarantee that you can do the former, but dozens of good books can help you get started.

## 16.8 Exercises

Here are some exercises for you to try on your own. Solutions are available at <http://programcrazed.com/16/exercises.html>.

1. Write a GUI application with a button labeled “Quit!”. When the button is clicked, the window closes.
2. Write a GUI application with a single button. Initially the button is labeled 0, but each time it is clicked, the value on the button increases by 1.
3. What is a more readable way to write the following?
 

```

s = lambda p:

```

4. A DNA sequence is a string made up of As, Ts, Cs, and Gs. With a GUI application in which a DNA sequence is entered, and when the Count button is clicked, the number of As, Ts, Cs, and Gs are counted and displayed in the window (see the image below):



8. In Section 3.4, Using Agent Variables for Temporary Variables on page 118, we wrote a function to convert `deg` into `Fahrenheit` to the two variables. Write a GUI application that looks like the image below.



When a value is entered in the text field and the current button is clicked, the value should be converted from Fahrenheit to Celsius and displayed in the window, as shown in the figure below.



- C. Rewrite the text below with Counting Month on page 238 as an object referred to it.

# Databases

In earlier chapters, we used files to store data. This is fine for small problems, but as our data sets become larger and more complex, we need something that will let us search for data in many different ways, control who can view and modify the data, and ensure that the data is correctly formatted. In short, we need a database.

Many different kinds of databases exist. Some are like a dictionary that automatically saves terms on disk, while others store backup copies of the objects in a program. The most popular by far, however, are relational databases, which are at the heart of most large commercial and scientific software systems. In this chapter you will learn about the key concepts behind relational databases and how to perform a few common operations.

## 17.1 Overview

A relational database is a collection of tables, each of which has a fixed number of columns and a variable number of rows. Each column in a table has a name and contains values of the same data type, such as integers or strings. Each row in a table contains values that are related to each other, such as a particular patient's name, date of birth, and blood type.

| Patients |            |            |
|----------|------------|------------|
| name     | birthday   | Blood type |
| Alice    | 1978-04-02 | A          |
| Bob      | 1954-03-15 | AB         |
| Clem     | 1961-04-05 | A          |
| Liz      | 1972-12-15 | A          |
| Marty    | 1948-03-15 | O          |

COLUMNS

Superficially, each table looks like a spreadsheet or a file with entries and just like the Readline Technique on page 179, but behind the scenes, the database does a lot of work to keep track of which values are where and how different tables relate to one another.

Many different kinds of databases are available to choose from, including commercial systems like Oracle, MySQL, PostgreSQL, and Microsoft Access and open source databases like MySQL, and PostgreSQL. One example we can recall is SQLite. It isn't fast enough to handle the heavy loads that sites like Amazon.com experience, but it is free, it is simple to use, and as of Python 3.3.0, the standard library (`sqlite3`) is included, making life easier with it.

A database is usually stored in a file or in a collection of files. These files aren't formatted as plain text—if you open them in an editor, they will look like garbage, and any changes you make will probably corrupt the data and make the database unusable. Instead, you must interact with the database in one of two ways:

- By issuing commands to a database GUI, just as you do with spreadsheets or a PostgreSQL interface. This is good for simple tasks but not for writing applications to your own.
- By writing programs in Python or some other language. These programs import a library that knows how to work with the kind of database you are using and use that library to create tables, insert records, and fetch the data you want. Your code can then format the results in a table, calculate statistics, or do whatever else you like.

In the examples in this chapter, our programs all start with this line:

```
conn = sqlite3.connect('test.db')
```

To put data into a database or to get information out, we'll write commands in a special-purpose language called SQL, which stands for Structured Query Language and is pronounced either as the three letters "S-Q-L" or as the word "sequel."

## 17.2 Creating and Populating

As a running example, we will use the predictions for regional populations in the year 2050, which is available at [www.un.org](http://www.un.org). The first table that we'll work with is Table 2B. Estimated World Population in 2050, on page 641. It has one column that contains the names of regions and another that contains the populations of regions, so each row of the table represents a region and its population.

| Region              | Population (in thousands) |
|---------------------|---------------------------|
| Central Africa      | 330,600                   |
| Southeastern Africa | 743,112                   |
| Northern Africa     | 1,337,460                 |
| Southern Asia       | 2,031,641                 |
| Asia Pacific        | 785,469                   |
| Middle East         | 587,690                   |
| Eastern Asia        | 1,002,655                 |
| South America       | 393,121                   |
| Eastern Europe      | 223,427                   |
| North America       | 601,157                   |
| Western Europe      | 364,813                   |
| <b>Total</b>        | <b>100,569</b>            |

Table 25—Estimated World Population In 2005

If the countries were sized by their estimated populations, they would look like this:



An additional exercise, mentioned by Trifunovic earlier in this chapter, is to do the same thing with the 2005 data from Table 25.

Next we need make a connection to our database by calling the database module's connect method. This method takes one string as a parameter, which identifies the database to connect to. Since SQLite stores each entire database in a single file on disk, this is just the path to the file. Because the database population2005 already exists, it will be created:

```
my_conn = sqlite3.connect('population2005.db')
```

Once we have a connection, we need to get a cursor, like the cursor in an editor, this keeps track of where we are in the database so that if several programs are accessing the database at the same time, the database can keep track of who is trying to do what:

```
>>> cur = con.cursor()
```

We can now actually start working with the database. The first step is to create a database table to store the population data. To do this, we have to know that the operators used are Structured Query Language (SQL). The general form of an SQL statement for table creation is as follows:

```
CREATE TABLE TableName (Column1Type, Column2Type, ...)
```

The types of the data in each of the table's columns are chosen from the types that databases supports:

| Type    | Python Equivalent | Use                            |
|---------|-------------------|--------------------------------|
| NULL    | NoneType          | Missing or unknown value.      |
| INTEGER | int               | Integers.                      |
| BIGINT  | long              | 8 byte floating point numbers. |
| TEXT    | str               | String of characters.          |
| BLOB    | bytes             | Binary data.                   |

Table 17—SQL Data Types

To create a two-column table named `PopByRegion` to store region names as strings in the `Region` column and projected populations as integers in the `Population` column, we use this SQL statement:

```
CREATE TABLE PopByRegion(Region INT, Population INT);
```

Now we put that SQL statement in a string and pass it as an argument to a Python method that will execute the SQL command:

```
>>> cur.execute('CREATE TABLE PopByRegion(Region INT, Population INT)')  
sqlite3.Cursor object at 0x102e2e400
```

When method `execute` is called, it returns the cursor object that it was called on. Since `execute` is that same cursor object, we don't need to do anything with the value returned by `execute`.

The most commonly used data types in SQLite databases are listed in Table 18, [SQLite Data Types](#), along with the corresponding Python data types. The `BLOB` type needs more explanation. The term `BLOB` stands for Binary Large Object, which in a database means a picture, an MP3, or any other kind of

below that isn't of a more specific type. The Python equivalent is a type we haven't seen before called `bytes`, which also stores a sequence of bytes that have no particular predefined meaning. We won't use `bytes` as an example, b/c the examples will likely run a chance to interact with them.

After we create a table, our next task is to insert data into it. We do this one record at a time using the **INSERT** command, whose general form is as follows:

so with the arguments to a function all the values are matched left to right against the parameters. For example, we insert data into the Register table like this:

```
>>> cur.execute("INSERT INTO PopByRegion VALUES('Central Africa', 100000000)  
>sqlitedb.Cursor object at 0x1000044C0>  
>>> cur.execute("INSERT INTO PopByRegion VALUES('Southeastern Africa',  
...  
'7431121')  
>sqlitedb.Cursor object at 0x1000044C0>  
>>>  
>>> cur.execute("INSERT INTO PopByRegion VALUES('Japan', 100000000)  
>sqlitedb.Cursor object at 0x1000044C0>
```

Remember the number and type of values in the JSON statements matching the number and type of columns in the database table. If you try to insert a value of a different type than the one declared for the column, the library will try to convert it, just as it converts the Integer 5 to a floating point number when width:12+5. For example, if we insert the Integer 32 into a TEXT column, it will automatically be converted to "32". Similarly, if we insert a string into an INTEGER column, it is passed to see whether it represents a number. It is, the string is ignored.

If the number of values being inserted doesn't match the number of columns in the table, the database reports an error and the data is not inserted. Our `petroliny` function, however, is smart enough to convert each value to the correct type, such as the string "Honda" into an `INTEGER` field. SQLite will actually do it although other databases will not.

Another useful feature is the `SETF` macro command uses placeholders for the values to be inserted. When using this function, instead create two `SETF` arguments: the first is the `SETF` command with question marks as placeholders for the values to be inserted, and the second is a tuple. When the command is evaluated, the values from the tuple are substituted for the placeholders from left to right. For example, the `coerce` method call to insert a row with "open" and 100% can be written like this:

```
:cc: cur.execute("INSERT INTO PepeFoglio VALUES (1, 1), ('Inizio', 18552)");
```

In this example, `java!` is used in place of the first question mark, and `100000` is placed of the second. This placeholder notation can come in handy when using a loop to insert data from a file or a file into a database, as shown in Section 17.6, [Using Java to Compose Tables](#), on page 340.

## Saving Changes

After we've inserted data into the database, or made any other changes, we must commit those changes using the connection's `commit` method:

```
conn.commit()
```

Committing to a database is like saving the changes made to a file to a text editor. Until we do it, our changes are not actually stored and are not visible to anyone else who is using the database at the same time. Requiring programs to commit is a form of **insurance**. If a program crashes partway through a long sequence of database operations and `commit` is never called, then the database will appear as it did before any of those operations were executed.

## 17.3 Retrieving Data

Now that our database has been created and populated, we can run queries to search for data that meets specific criteria. The general form of a query is as follows:

```
SELECT <columns> FROM <table>
```

The `<table>` is the name of the table to get the data from and the `<column>`s specify which columns to get values from. For example, this query retrieves all the data in the table `Region`:

```
conn.createStatement("SELECT * FROM Region")
```

Once the database has evaluated this query for us, we can access the results one record at a time by calling the cursor's `next` method, just as we can read one line at a time from a file using `readLine`:

```
conn.createStatement("SELECT * FROM Region").  
next();
```

The `next` method returns each record as a tuple (see [Section 11.9, "Getting Data from Tables](#), on page 204) whose elements are in the order specified in the query. If there are no more records, `next` returns `false`.

Just as files have a `readAll` method to get all the lines in a file at once, databases currently have a `fetchAll` method that returns all the data produced by a query that has not yet been fetched as a list of tuples:

```
(cc: cur.fetchall())
[('Northwestern Africa', 100100), ('Northern Africa', 100100), ('Southern
Africa', 100100), ('Asia Pacific', 100100), ('Middle East', 100100),
('Eastern Asia', 100100), ('South America', 100100), ('Central Europe',
200200), ('North America', 600100), ('Western Europe', 300300), ('East',
100100)]
```

Once all of the data has been loaded, the query has been issued and subsequent calls on `fetchone` and `fetchmany` return `None` and the empty list, respectively:

```
(cc: cur.fetchone())
cc: cur.fetchmany()
[]
```

Like a dictionary or a set ([Chapter 11: Storing Data Using Cache-Control](#)), a database stores records in whatever order it thinks is most efficient. To put the data in a particular order, we could sort the list returned by `fetch`. However, it's more efficient to get the database to do the sorting for us by adding an `ORDER BY` clause to the query like this:

```
(cc: cur.execute('SELECT Region, Population FROM PopByRegion ORDER BY Region')
cc: cur.fetchall())
[(('Asia Pacific', 783460), ('Central Africa', 200500), ('Eastern Asia',
12962760), ('Western Europe', 200420), ('Japan', 100500), ('Middle East',
6076980), ('North America', 600100), ('Northeast Africa', 1002400), ('South
America', 100100), ('Southeastern Africa', 200300), ('Southern Asia',
200100), ('Western Europe', 300300)])]
```

By changing the minimum value after the `ORDER BY` to, we can change the way the database sorts. As the following code demonstrates, we can also specify whether we want values sorted in increasing (ASC) or decreasing (DESC) order:

```
(cc: cur.execute(''''SELECT Region, Population FROM PopByRegion
        ORDER BY Population DESC'''')
cc: cursor object at 0x0000000000000000
cc: cur.fetchall())
[(('Southern Asia', 200100), ('Eastern Asia', 100500), ('Northern Africa',
1007400), ('Asia Pacific', 783460), ('Southeastern Africa', 200300),
('Middle East', 6076980), ('North America', 600100), ('South America',
200100), ('Western Europe', 200420), ('Central Africa', 100500), ('Eastern
Europe', 200200), ('Japan', 100500)])]
```

As we've seen, we can specify one or more columns by name in a query. We can also use `*` to indicate we want all columns:

```
(cc: cur.execute('SELECT Region, Population FROM PopByRegion')
cc: cursor object at 0x0000000000000000
cc: cur.fetchall())
[(('Central Africa', 100500), ('Southeastern Africa', 200300),
('Northern Africa', 1007400), ('Asia Pacific', 783460), ('Middle East',
6076980), ('South America', 200100), ('North America', 600100),
('Eastern Europe', 200200), ('Western Europe', 200420), ('Japan', 100500)])]
```

```
[('Southern Asia', 1), ('Asia Pacific', 2), ('Middle East', 3), ('Central
Africa', 4), ('North America', 5), ('Eastern Europe', 6), ('Latin America', 7),
('Western Europe', 8), ('Japan', 9)]
<== cur.fetchall() <-- SQL SELECT * FROM Regions>
<== cur.close() <-- object at 0x102e0450>
<== cur.fetchmany(1)
[('Central Africa', 200000), ('Southeastern Africa', 710000),
('Northern Africa', 1607400), ('Southern Asia', 2253243), ('Asia
Pacific', 705400), ('Middle East', 207000), ('Eastern Europe', 1355000),
('South America', 300121), ('Eastern Europe', 332437), ('Latin America', 10
00157), ('Western Europe', 207500), ('Japan', 1000000)]
```

## Query Conditions

Much of the time, we want only some of the data in the database. (Think about what would happen if you asked Google for all of the web pages it had stored.) We can extract a subset of the data by using the keyword WHERE to specify conditions that the rows we want must satisfy. For example, we can get the regions with a population greater than one million using the greater-than operator:

```
<== cur.execute('SELECT Region FROM Regions WHERE Population > 1000000')
<== cur.fetchone() <-- object at 0x102e0450>
<== cur.fetchmany(1)
[('Northern Africa', 1), ('Southern Asia', 2), ('Eastern Europe', 3)]
```

These are the relational operators that may be used with WHERE:

| Operator | Description              |
|----------|--------------------------|
| =        | Equal to                 |
| !=       | Not equal to             |
| >        | Greater than             |
| <        | Less than                |
| >=       | Greater than or equal to |
| <=       | Less than or equal to    |

Table 17—SQL Relational Operators

Not surprisingly, they are the same as the ones that Python and other programming languages provide. As well as these relational operators, we can also use the AND, OR, and NOT operators. To get all the regions with populations greater than one million that have names that come before the letter L in the alphabet, we would use this (we are using a triple quoted string for the SQL statement so that it can span multiple lines):

```
>>> cur.execute('''
    SELECT Region, SUM(Population)
    WHERE Population > 1000000000
    GROUP BY Region
    ORDER BY Region ASC
    ''')
>>> cur.fetchone()
[('Eastern Asia', 1)]
```

Where conditions are always applied row by row, they cannot be used to compare two or more rows. We will see how to do that in Section 17.6, Using Joins to Compare Tables, in [this book](#).

## 17.4 Updating and Deleting

Now often than not over time, we want to be able to change the information stored in a database. To do that, we can use the UPDATE command, as shown in the following code:

```
>>> cur.execute('SELECT * FROM PopByRegion WHERE Region = "Japan"')
>>> cur.fetchone()
['Japan', 100000]
>>> cur.execute('''
    UPDATE PopByRegion SET Population = 100000
    WHERE Region = "Japan"
    ''')
>>> cur.execute('SELECT * FROM PopByRegion WHERE Region = "Japan"')
>>> cur.fetchone()
['Japan', 100000]
```

We can also delete records from the database:

```
>>> cur.execute('DELETE FROM PopByRegion WHERE Region < "A"')
>>> cur.fetchone()
None
>>> cur.execute('SELECT * FROM PopByRegion')
>>> cur.fetchmany(10)
>>> cur.fetchall()
[('Northwestern Africa', 100000), ('Northern Africa', 100000),
 ('Southern Asia', 100000), ('Middle East', 100000), ('South America',
 100000), ('North America', 100000), ('Eastern Europe', 100000)]
```

In both cases, all records that meet the WHERE condition are affected. If we don't include a WHERE condition, then all rows in the database are updated or removed. However, we can always put records back into the database:

```
>>> cur.execute('INSERT INTO PopByRegion VALUES ("Japan", 100000)')
```

To remove an entire table from the database, we can use the DROP command:

```
>>> cur.execute('DROP TABLE PopByRegion')
```

For example, if we no longer want the table PopByRegion, we can do something like:

```
>>> cur.execute('DROP TABLE PopByRegion')
```

When a table is dropped, all the data contained is lost. You should be very, very sure you want to do this (and even then, it's probably a good idea to make a backup copy of the database before deleting any sizable tables).

## 17.5 Using NULL for Missing Data

In the real world, we often don't have all the data we want. We might be missing the time at which an experiment was performed or the postal code of a patient being given a new kind of treatment. Rather than know what we do know out of the database, we may choose to insert it and use the value `NULL` to represent the missing values. For example, if there is a region whose population we don't know, we could insert this into our database:

```
(cc: cur,conn,cur) | INSERT INTO Region VALUES ('East', NULL)
```

On the other hand, we probably don't ever want a record in the database that has a `NULL` region name. We can prevent this from ever happening, stating that the column is `NOT NULL` when the table is created:

```
(cc: cur,conn,cur) | CREATE TABLE Test (Region NOT NULL,
...                           Population INTEGER)
```

Now when we try to insert a `NULL` region into our new test table, let's get an error message:

```
(cc: cur,conn,cur) | INSERT INTO Test VALUES ('East', NULL)
ERROR:  must specify value for column "Population"
```

Rule 1: `NOT NULL`

```
(cc: cur,conn,cur) | INSERT INTO Test VALUES ('West', 45678)
```

Population integrity constraint violated: key ("Region") is null.

So far, this behavior may not be `NULL` is not always necessary, and imposing such a constraint may not be reasonable in some cases. Rather than using `NULL`, it may sometimes be more appropriate to use the value `zero`, an empty string, or `None`. You should do so in cases where you know everything about the data and use `NULL` only in cases where you know nothing at all about it.

In fact, some experts recommend not using `NULL` at all because the behavior is counterintuitive (at least until you've learned your lesson). The general rule of thumb is that operations involving `NULL` produce `NULL` as a result. The reasoning is that if the computer doesn't know what one of the operation's inputs is, it can't know what the output is either. Adding a number to `NULL` just gives `NULL` no matter what the number was, and multiplying by `NULL` also produces `NULL`.

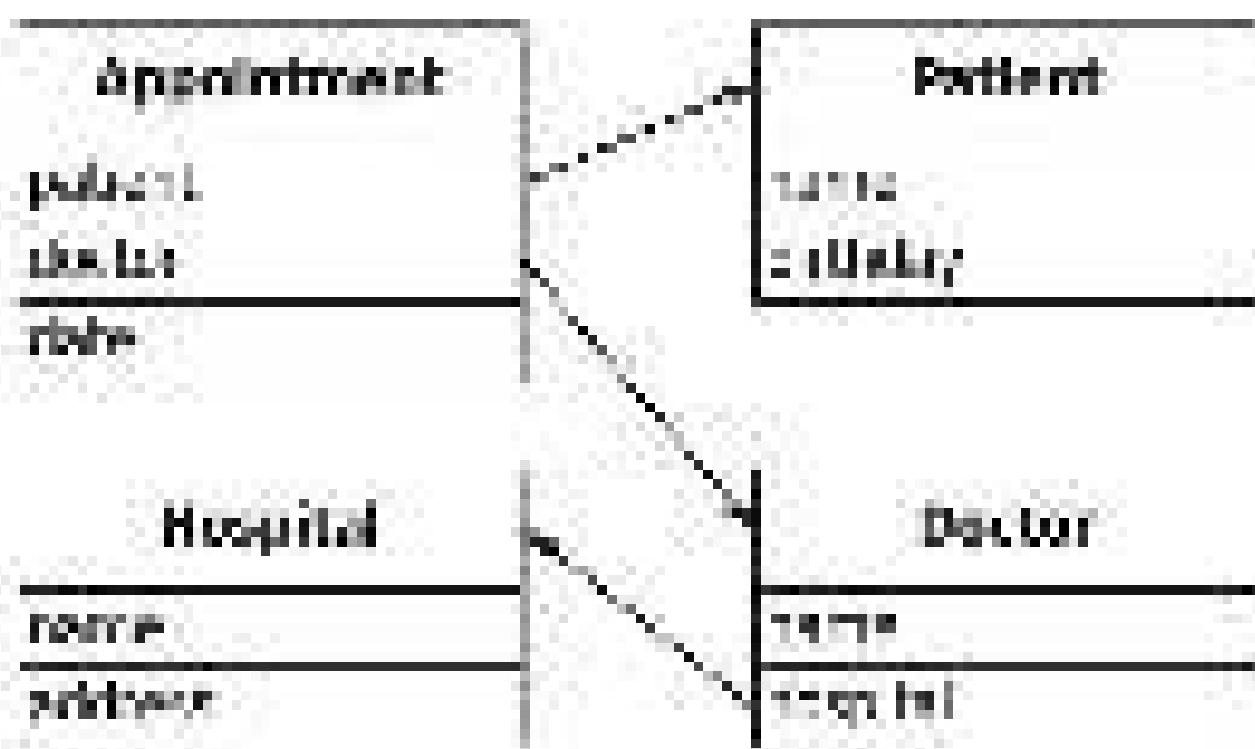
This is not what's compatibility with logical operators. The expression `NULL || 1` evaluates to 1, rather than `NULL`, because of the following:

- If the first argument was false (or 0 or an empty string, or some equivalent value), the result would be 1.
- If the first argument was true (or nonzero, or a nonempty string), the result would also be 1.

The technical term for this is *three-valued logic*. In SQL's view of the world, things aren't just true or false—they can be true, false, or unknown, and NULL represents the last. Unfortunately, different databases & languages implement this in the SQL standard in different ways, so their handling of NULL is not consistent. NULL should therefore be used with caution and only when other approaches won't work.

## 17.6. Using Joins to Combine Tables

When developing a database, it often makes sense to divide data between two or more tables. For example, if you are maintaining a database of patient records, you would probably want at least four tables: one for the patient's personal information (such as name and date of birth), a second to keep track of appointments, a third for information about the doctors who are treating the patient, and a fourth for information about the hospitals or clinics those doctors work at.



We could store all of this in one table, but then a lot of information would be needlessly duplicated:

| Patient-Doctor-Appointment-Hospital |            |           |            |            |              |           |
|-------------------------------------|------------|-----------|------------|------------|--------------|-----------|
| patient                             | birthday   | doctor    | date       | name       | address      | specialty |
| John Doe                            | 1970-01-01 | Dr. Smith | 2010-01-01 | Hospital A | 123 Main St. | General   |
| Jane Doe                            | 1970-01-01 | Dr. Brown | 2010-01-01 | Hospital A | 123 Main St. | General   |
| John Doe                            | 1970-01-01 | Dr. Brown | 2010-01-01 | Hospital B | 456 Elm St.  | General   |
| Jane Doe                            | 1970-01-01 | Dr. Brown | 2010-01-01 | Hospital B | 456 Elm St.  | General   |
| John Doe                            | 1970-01-01 | Dr. Brown | 2010-01-01 | Hospital C | 789 Oak St.  | General   |
| Jane Doe                            | 1970-01-01 | Dr. Brown | 2010-01-01 | Hospital C | 789 Oak St.  | General   |

If we divide information between tables, though, we need some way to pull that information back together. For example, if we want to know the hospital at which a patient has had appointments, we need to combine data from all four tables to find out the following:

- Which appointments the patient has had
- Which doctor each appointment was with
- Which hospital/clinic the doctor works at

The right way to do this in a relational database is to use a **join**. As the name suggests, a join combines information from two or more tables to create a new set of records, each of which can contain some or all of the information in the tables involved.

To begin, let's add another table that contains the names of countries, the regions that they are in, and their populations:

```
sql> create table PopByCountry(Region INT, Country TEXT,
    Population INT);
```

Then let's insert data into the new table:

```
sql> insert into PopByCountry values('Eastern Asia', 'China', 1340000000);
sql> insert into PopByCountry values('Eastern Asia', 'India', 1130000000);
...;
```

Inserting data one row at a time like this requires a lot of typing. To exemplify, to make a list of tuples to be inserted and write a loop that inserts the values from rows, begin one by one using the placeholder notation from [Section 17.2](#), Creating and Populating, in just 7 lines:

```
sql> countries = [ ("Eastern Asia", "PRR Korea", 24000), ("Eastern Asia",
    "Hong Kong (China)", 6764), ("Eastern Asia", "Mongolia", 2402), ("Eastern
    Asia", "Republic of Korea", 43400), ("Eastern Asia", "Taiwan", 2422),
    ("North America", "Bahamas", 366), ("North America", "Canada", 35322),
    ("North America", "Greenland", 43), ("North America", "Mexico", 123279),
    ("North America", "United States", 455888)];
sql> for c in countries:
    sql>     cur.execute("insert into PopByCountry values(%s, %s, %s)", (c[0], c[1], c[2]));
...
sql> conn.commit()
```

Now that we have two tables in our database, we can use joins to combine the information they contain. Several types of joins exist; you'll learn about them, joins and self-joins.

We'll begin with inner joins, which involve the following. (Note that the numbers in this list correspond to the last numbers in the following diagram.)

1. Constructing the cross product of the tables.
2. Discarding rows that do not meet the selection criteria.
3. Selecting columns from the remaining rows.



① Cross product of PopByRegion and PopByCountry

| PopByRegion                     | PopByCountry     |
|---------------------------------|------------------|
| Eastern Asia<br>40.0M People    | 1362955<br>China |
| North America<br>66.357M People | 66.357M People   |
| Eastern Asia<br>40.0M People    | 66.357M People   |
| North America<br>66.357M People | 66.357M People   |

② Discard rows where Region != Country

| PopByRegion   | PopByCountry | PopByRegion   | PopByCountry |
|---------------|--------------|---------------|--------------|
| Eastern Asia  | 1362955      | Eastern Asia  | Vietnam 340M |
| North America | 66.357M      | North America | Greenland 42 |
| Eastern Asia  | 66.357M      | North America | Uruguay 4M   |
| North America | 66.357M      | Eastern Asia  | Vietnam 340M |

③ Select columns Region, Population, and Country

| Region        | Population | Country       |
|---------------|------------|---------------|
| Eastern Asia  | 1362955    | Eastern Asia  |
| North America | 66.357M    | North America |

First, all combinations of all rows in the tables are combined, which makes the cross product. Second, the selection criteria specified by WHERE is applied, and rows that don't match are removed. Finally, the selected columns are kept, and all others are discarded.

In our earlier query, we retrieved the names of regions with projected populations greater than one million. Using an INNER JOIN, we can get the names of the countries that are in those nations. The query and its results look like this:

```
--> run.succeful()
SELECT PopByRegion.Region, PopByCountry.Country
FROM PopByRegion INNER JOIN PopByCountry
WHERE PopByRegion.Region = PopByCountry.Region
AND PopByRegion.Population > 1000000
-->
```

```
sqlplusS.Curso> object at 0x182e3e458
--> run.failed()
-->
```

```
1) 'Eastern Asia', 'China', 'Eastern Asia', 'PR China'
1) 'Eastern Asia', 'Hong Kong (Kwun Tong)', 'Eastern Asia', 'Hong Kong'
1) 'Eastern Asia', 'Republic of Korea', 'Eastern Asia', 'Korea'
```

To understand what this query is doing, we can analyze it in terms of the three steps outlined earlier:

- Containerizing raw data alongside with a summary table of proportions. `tbl_df` has 2 columns and 12 rows while `tbl_summary` has 8 columns and 11 rows, so this generates a temporary table with 9 columns and 130 rows:

|                     |         |              |                  |        |
|---------------------|---------|--------------|------------------|--------|
| North America       | 222222  | Eastern Asia | CPR, China       | 220000 |
| Southeastern Africa | 242222  | Eastern Asia | CPR, China       | 240000 |
| Central Africa      | 2222422 | Eastern Asia | CPR, China       | 240000 |
| Eastern Asia        | 2222512 | Eastern Asia | CPR, China       | 240000 |
| South Europe        | 222272  | Eastern Asia | CPR, China       | 220000 |
| Middle East         | 222273  | Eastern Asia | CPR, China       | 220000 |
| Eastern Asia        | 222255  | Eastern Asia | CPR, China       | 220000 |
| South America       | 222222  | Eastern Asia | CPR, China       | 220000 |
| Central Europe      | 2222422 | Eastern Asia | CPR, China       | 220000 |
| South America       | 222223  | Eastern Asia | CPR, China       | 220000 |
| Central America     | 222224  | Eastern Asia | CPR, China       | 220000 |
| Modern Europe       | 222252  | Eastern Asia | CPR, China       | 220000 |
| 22227               | 222272  | Eastern Asia | CPR, China       | 220000 |
| Central Africa      | 2222532 | Eastern Asia | Hong Kong, China | 220000 |
| Southeastern Africa | 2222532 | Eastern Asia | Hong Kong, China | 220000 |
| Central Africa      | 2222422 | Eastern Asia | Hong Kong, China | 220000 |
| South America       | 2222542 | Eastern Asia | Hong Kong, China | 220000 |
| Asia Pacific        | 2222422 | Eastern Asia | Hong Kong, China | 220000 |
| 22227               | 2222512 | Eastern Asia | Hong Kong, China | 220000 |
| Central Africa      | 2222522 | Eastern Asia | Hong Kong, China | 220000 |
| Central Asia        | 2222532 | Eastern Asia | Hong Kong, China | 220000 |
| South America       | 2222532 | Eastern Asia | Hong Kong, China | 220000 |
| South America       | 2222422 | Eastern Asia | Hong Kong, China | 220000 |
| South America       | 2222542 | Eastern Asia | Hong Kong, China | 220000 |
| South America       | 2222422 | Eastern Asia | Hong Kong, China | 220000 |
| Central Europe      | 2222522 | Eastern Asia | Hong Kong, China | 220000 |
| 22227               | 2222532 | Eastern Asia | Hong Kong, China | 220000 |

2. Discontinues that do not meet the selection criteria. The rules WHERE clause specifies DISCONTINUED. The region taken from Page 292 must be the same as the region taken from PopbyCountry, and the region's population must be greater than one million. This first collection contains four we don't need, which each has a unique ID. We'll remove with regard to population in Step 2a; the second filter removes information about countries in regions whose populations are less than that threshold.
  3. Finally, select the region and country names from the rows that have survived.

## Removing Duplicates

To find the regions where company accounts for more than 10 percent of the region's overall population, we would also need to join the two tables.

www.gutenberg.org

W.H.T. Hsieh, Region Region

PHD Prog, Majlis, Makk. 1010 Proficiency

ISSN 1062-1024 • Volume 29 Number 10 • November 2007 • \$5.95

```
AND ((PopByCountry.Population > 100) & PopByRegion.Population < 200); -- |
--> result.fetchall();
```

[1]: execute('select r.Region, c.Country, p.Population from PopByRegion p
inner join PopByCountry c on p.Region = c.Region
where ((PopByCountry.Population > 100) & PopByRegion.Population < 200);');
--> result.fetchall();

We now multiply region and division in our WHERE condition to calculate the percentage of the region's population by country as a floating-point number. The resulting list contains duplicates, since more than one North American country accounts for more than 100 percent of its region's population. To remove the duplicates, we add the keyword DISTINCT to the query:

```
--> result.fetchall();
```

```
SELECT DISTINCT PopByRegion.Region
FROM PopByRegion INNER JOIN PopByCountry
WHERE (PopByRegion.Region = PopByCountry.Region)
AND ((PopByCountry.Population > 100) & PopByRegion.Population < 200); -- |
--> result.fetchall();
```

[1]: execute('select r.Region, c.Country, p.Population from PopByRegion p
inner join PopByCountry c on p.Region = c.Region
where ((PopByCountry.Population > 100) & PopByRegion.Population < 200);');
--> result.fetchall();

Now in the results above, North America appears once.

## 12.2 Keys and Constraints

Our query in the previous section relied on the fact that our regions and countries were uniquely identified by their names. A column in a table that is used this way is called a key. Ideally, a key's values should be unique, just like the keys in a dictionary. We can tell the database to enforce this constraint by adding a PRIMARY KEY clause when we create the table. For example, when we created the PopByRegion table, we should have specified the primary key:

```
--> cur.execute('CREATE TABLE PopByRegion (
Region TEXT NOT NULL,
Population INTEGER NOT NULL,
PRIMARY KEY (Region));')
```

Just as a key in a dictionary can't be made up of multiple values, the primary key for a database table can consist of multiple columns.

The following code uses the CONSTRAINT keyword to specify that no two entries in the table being created will ever have the same values for region and country:

```
--> cur.execute('
CREATE TABLE PopByCountry (
Region TEXT NOT NULL,
Country TEXT NOT NULL,
Population INTEGER NOT NULL,
CONSTRAINT CountryKey PRIMARY KEY (Region, Country));')
```

In practice, most database designers don't use real names as primary keys. Instead, they usually create a unique integer ID for each "thing" in the database, such as a driver's license number or a patient ID. This is partly done for efficiency reasons—keys are faster to sort and compare than strings—but the real reason is that it is a simple way to deal with things that have the same name. There are a lot of Jane Smiths in the world; using that name as a primary key is a disaster! It almost guarantees a key collision. Giving each person a unique ID, on the other hand, ensures that they can be told apart.

## 17.8 Advanced Features

The SQL we have seen so far is powerful enough for many everyday tasks, but other questions require more powerful tools. This section introduces a handful and shows when and how they are useful.

### Aggregation

Our next task is to calculate the total projected world population for this year (2011). We will do this by adding up the values in `PopEst::Projected` column using the SQL aggregate function `SUM`:

```
--> cur.execute('SELECT SUM([Population]) FROM PopEst')
cursor2 = cursor.execute('SELECT SUM([Population]) FROM PopEst')
--> cur.fetchone()
(1255062,)
```

`SUM` provides several other aggregate functions. All of them are associative that is, the result doesn't depend on the order of operations. This means that the result doesn't depend on the order in which records are pulled out of tables.

| Aggregate Function | Description              |
|--------------------|--------------------------|
| AVG                | Average of the values    |
| MIN                | Minimum value            |
| MAX                | Maximum value            |
| COUNT              | Number of nonnull values |
| SUM                | Sum of the values        |

Table 31—Aggregate Functions

Addition and multiplication are associative, since  $1 + 2 + 3$  produces the same results as  $(1 + 2) + 3$ , and  $4 \cdot 5 \cdot 6$  produces the same result as  $4 \cdot (5 \cdot 6)$ . By contrast, subtraction isn't associative.  $1 - 2 - 3$  is not the same thing as  $1 - (2 - 3)$ . Notice that there isn't a subtraction aggregate function.

## Grouping

What if we only had the table `PopCountry` and wanted to find the projected population for each region? We could go the table's contents into a Python program using `SELECT *` and then loop over them to add them up by region, but again, it is simpler and more efficient to have the database do the work for us. In this case, we use `SQL's GROUP BY` to collect records into subtotals:

```
>>> cur.execute('SELECT sum(Population) FROM PopCountry GROUP BY Region')
>>> cursor = conn.cursor()
>>> cur.fetchall()
[(1264268, ), (661200, )]
```

Since we have asked the database to construct groups by region and there are two distinct values in this column in the table, the database divides the records into two subtotals. It then applies the `SUM` function to each group separately to give us the projected populations of Eastern Asia and North America:

| PopCountry    |                   |            |
|---------------|-------------------|------------|
| Region        | Country           | Population |
| North America | Bahamas           | 368        |
| North America | Mexico            | 1264268    |
| Eastern Asia  | Kongo Is.         | 3607       |
| Eastern Asia  | Republic of Korea | 41491      |

➊ Group by region and sum the total population:

|                |         |         |
|----------------|---------|---------|
| North America  | Bahamas | 368     |
| North America  | Mexico  | 1264268 |
| <b>1267243</b> |         |         |

|              |                   |       |
|--------------|-------------------|-------|
| Eastern Asia | Monrovia          | 3607  |
| Eastern Asia | Republic of Korea | 41491 |
| <b>44898</b> |                   |       |

➋ Keep selected columns:

|               |         |
|---------------|---------|
| North America | 1267243 |
| Eastern Asia  | 44898   |

## Self-Joins

Let's consider the problem of comparing a table's values to themselves. Suppose that we want to find pairs of countries whose populations are close to each other—say, within 1,000 of each other. Our query might look like this:

```
>>> cur.execute("""SELECT Country FROM PopByCountry
    WHERE ABS(A.Population - B.Population) < 1000""")  

>>> results.Cursor object at 0x102a0a4cc  

>>> cur.fetchall()  

[('China',), ('DPR Korea',), ('Hong Kong (China)',), ('Mongolia',),
 ('Republic of Korea',), ('Tajikistan',), ('Ukraine',), ('Canada',),
 ('Greenland',), ('Mexico',), ('United States',)]
```

The output is definitely not what we want for two reasons. First, the plain SELECT query is giving us return only one country per record, while we want pairs of countries. Second, the expression ABS(A.Population - B.Population) is always going to return zero because we are subtracting each country's population from itself. Since every difference will be less than 1,000, the names of all the countries in the table will be returned by the query.

What we actually want to do is compare the populations in one row with the populations in each of the other rows. To do this, we need to JOIN PopByCountry with itself except for the first row:

| PopByCountry Cross Join Example |               |              |              |               |               |       |
|---------------------------------|---------------|--------------|--------------|---------------|---------------|-------|
| A.Country                       | B.Country     | A.Population | B.Population | A.Country     | B.Country     |       |
| North America                   | Canada        | 36776        | 36776        | North America | United States | 36776 |
| North America                   | United States | 36776        | 36776        | North America | United States | 36776 |
| Eastern Asia                    | Taiwan        | 232          | 232          | Eastern Asia  | Taiwan        | 232   |
| North America                   | United States | 36776        | 36776        | North America | United States | 36776 |
| North America                   | United States | 36776        | 36776        | Eastern Asia  | Taiwan        | 232   |
| Eastern Asia                    | Taiwan        | 232          | 232          | North America | Canada        | 36776 |
| North America                   | Canada        | 36776        | 36776        | Eastern Asia  | Taiwan        | 232   |
| North America                   | United States | 36776        | 36776        | North America | United States | 36776 |
| Eastern Asia                    | Taiwan        | 232          | 232          | North America | United States | 36776 |

This will result in the rows for each pair of countries being combined into a single row with six columns: two regions, two countries, and two populations. To tell them apart, we have to give the two instances of the PopByCountry table temporary names (e.g., this uses A and B):

```
>>> cur.execute("""  

    SELECT A.Country, B.Country  

    FROM PopByCountry A INNER JOIN PopByCountry B  

    WHERE ABS(A.Population - B.Population) <= 1000  

    AND A.Country != B.Country""")  

>>> results.Cursor object at 0x102a0a4cc  

>>> cur.fetchall()  

[('Republic of Korea', 'Canada'), ('Belarus', 'Russia'), ('Russia', 'Canada'),
 ('Russia', 'Belarus')]
```

Note that we used `NOT IN` to get the desired result of the population difference. Let's consider what would happen without `NOT IN`:

`1A.Population - B.Population >= 1000`

Omitting `NOT IN` would result in getting the `“Greater”` filter being included, because every nation's difference is less than ...`000`. If we want each pair of countries to appear only once (in my output), we could rewrite the second half of the condition as follows:

`A.Country < B.Country`

By changing the condition above, each pair of countries only appears once.

## Nested Queries

Up to now, our queries have involved only one `SELECT` command. Since the result of every query looks exactly like a table with a fixed number of columns and some number of rows, we can run a second query on the result—that is, run a `SELECT` on the result of another `SELECT`, rather than directly on the database's tables. Such queries are called nested queries and are analogous to having one function called on the value returned by another function call.

To see why we would want to do this, let's write a query on the `PopByCountry` table to get the regions that do not have a country with a population of  $\geq 700,000$ . Our first attempt looks like this (remember that the units are in thousands of people):

```
>>> cur.execute(''SELECT DISTINCT Region
    FROM PopByCountry
    WHERE (PopByCountry.Population >= 700)'')
```

`<sqlite3.Cursor object at 0x100a2e45c>`

`>>> cur.fetchall()`

`[('Eastern Asia', 'North America')]`

This result is wrong. Hong Kong has a projected population of 8,704,000, so `cur.fetchall` shouldn't have been returned. This means other countries in eastern Asia have populations that exceed  $\geq 700,000$ . Though, `cur.fetchall` was included in the final results.

Let's think our strategy. What we need to do is find out which regions include countries with a population of  $\geq 700,000$  and then exclude those regions from our final result. Basically, find the regions that follow our condition and subtract them from the set of all countries (`Region`: 18, `NextGen_regions`, in part: `ASIA`).

The last step is to get those regions that have countries with a population of  $\geq 700,000$ , as shown in the following code:

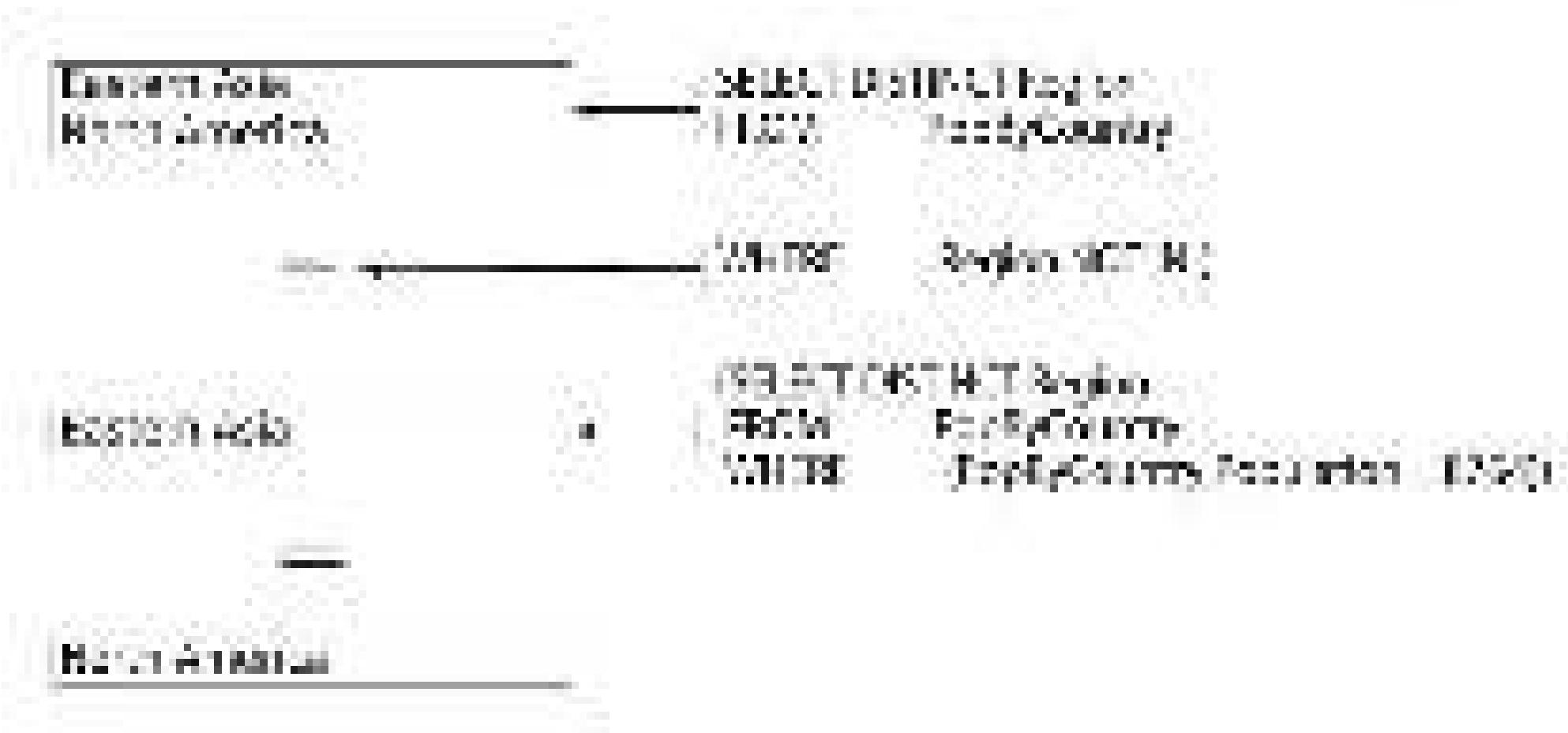


Figure 16—Nested negation

```

--> cur.execute()
SELECT DISTINCT Region
FROM PopByCountry
WHERE (PopByCountry.Population > 8000)
AND
PopByCountry.Population <= 8000
--> cur.fetchall()
[] ('Western Asia')
  
```

Now we want to get the names of regions that were not in the result of our first query. To do this, we will use a WHERE clause and NOT IN:

```

--> cur.execute()
SELECT DISTINCT Region
FROM PopByCountry
WHERE Region NOT IN
  (SELECT DISTINCT Region
   FROM PopByCountry
   WHERE (PopByCountry.Population > 8000)
   AND
   PopByCountry.Population <= 8000)
--> cur.fetchall()
[] ('North America')
  
```

This time we got what we were looking for. Nested queries are often used for anomalies like this one, where mutation is involved.

## Transactions

A transaction is a sequence of database operations that are interdependent. No operation in a transaction can be committed unless every single one can be successfully committed in sequence. For example, if an employee is paying an employee, there are two independent operations: withdraw of funds

from the employer's account and depositing funds in the employee's account. By grouping the operations into a single transaction, it is guaranteed that either both operations occur or neither operation occurs. When executing the operations in a transaction, if one operation fails, the transaction must be rolled back. That causes all the operations in the transaction to be undone. Using transactions ensures the database doesn't end up in an inconsistent state (such as having funds withdrawn from the employer's account and deposited in the employee's account).

Databases create transactions automatically. As soon as you try to start an operation (such as by calling the `open()` method) it becomes part of a transaction. When you commit the transaction successfully, the changes become permanent. At that point, a new transaction begins.

Imagine a library that may have multiple copies of the same book. It uses a computerized system to track its books by their ISBN numbers. Whenever a patron signs out a book, a query is executed on the Books table to find out how many copies of that book are currently signed out, and then the table is updated to indicate that one more copy has been signed out:

```
cur.execute('SELECT Signout FROM Books WHERE ISBN = ?;', [isbn])
signedOut = cur.fetchone()[0]
cur.execute('UPDATE Books SET Signout = ? WHERE ISBN = ?; Signout + 1, [isbn])
cur.commit()
```

Given a patron who wants to have the reverse happen:

```
cur.execute('SELECT Signout FROM Books WHERE ISBN = ?; signOut)
signedOut = cur.fetchone()[0]
cur.execute('UPDATE Books SET Signout = ? WHERE ISBN = ?; signOut - 1, [isbn])
cur.commit()
```

What if the library had two computers that handled book signings and returns? Both computers connect to the same database. What happens if one patron tries to return a copy of *Gray's Anatomy* while another was signing out a different copy of the same book at the exact same time?

One possibility is that computers A and B would each execute queries to determine how many copies of the book have been signed out, thus computer A would add one to the number of copies signed out and update the table without computer B knowing. Computer B would decrease the number of copies listed on the query result and update the table.

Here's the code for that scenario:

```

Computer-A: cur.execute("SELECT Segmented FROM Books WHERE ISBN = ? , :book")
Computer-B: assignedOut = cur.fetchone()[0]
Computer-B: cur.execute("SELECT Segmented FROM Books WHERE ISBN = ? , :book")
Computer-B: assignedOut = cur.fetchone()[0]
Computer-A: cur.execute("""UPDATE Books SET Segmented = ?
    WHERE ISBN = ?""", assignedOut + 1, :book)
Computer-A: cur.commit()
Computer-B: cur.execute("""UPDATE Books SET Segmented = ?
    WHERE ISBN = ?""", assignedOut - 1, :book)
Computer-B: cur.commit()

```

Notice that computer B counts the number of cloned out copies before computer A registers the database. After computer A commits its changes, the value that computer B has just increased. If computer B were allowed to commit its changes, the Library Database would account for more books than the Library actually had!

Fortunately, databases can detect such a situation and would prevent computer B from committing its transaction.

## 17.9 Some Data Based On What You Learned

In this chapter, you learned the following:

- Most large applications store information in relational databases. A database is made up of tables, each of which stores logically related information. A table has one or more columns—each of which has a name and a type—and zero or more rows, or records. In most tables, each row can be identified by a unique key, which consists of one or more of the values in the row.
- Commands to put data into databases, or to get data out, can be written in a specialized language called SQL.
- SQL commands can be sent to databases interactively from GUIs or command line tools, but for larger jobs, it is more common to write programs that create SQL and process the results.
- Changes made to a database don't actually take effect until they are committed. This ensures that if two or more programs are working with a database at the same time, it will always be in a consistent state. However, it also means that operations in one program can fail because of something that another program is doing.
- SQL queries must specify the table(s) and column(s) that values are to be taken from. They may also specify Boolean conditions, their values must satisfy and the ordering of results.

- Simple queries work on one row at a time, but we can use joins to combine values from different rows. Queries can also group and aggregate rows to calculate sums, averages, and other values.
- NaN values can use the special value `NaN` to represent missing information. However, it must be used with caution, since operations on `NaN` values don't behave in the same way that operations on float values do.

## 13.10 Exercises

Here are some exercises for you to try on your own. Solutions are available at <https://www.pythontutor.com/problems/canada>.

- In this exercise, you will create a table to store the population and land area of the Canadian provinces and territories according to the 2001 census. Our data is taken from <https://open.canada.ca/datacatalogue/13004.html>.

| Province/Territory        | Population | Land Area  |
|---------------------------|------------|------------|
| Newfoundland and Labrador | 5129301    | 570521.83  |
| Prairie Edward Island     | 1062294    | 2644.33    |
| Nova Scotia               | 9090117    | 521914.43  |
| New Brunswick             | 7203985    | 71356.87   |
| Quebec                    | 7237479    | 1937743.03 |
| Ontario                   | 11410040   | 907655.53  |
| Manitoba                  | 1119689    | 531987.97  |
| Saskatchewan              | 978693     | 586461.33  |
| Alberta                   | 2974807    | 939567.12  |
| British Columbia          | 3907799    | 926492.43  |
| Yukon Territory           | 38674      | 474706.97  |
| Northwest Territories     | 57260      | 1141108.37 |
| Nunavut                   | 26745      | 1920460.13 |

Table 3.2 2001 Canadian Census Data

Write Python code that does the following:

- Create a new database called `canada`.
- Makes a database table called `territory` that will hold the name of the province or territory (TERR), the population (INSPR), and the land area (LTSA).

- i. Insert the data from Table 32, 2001 Canadian Census Data, on page 331.
- ii. Retrieve the *provinces* of the table.
- iii. Retrieve the populations.
- iv. Retrieve the provinces that have populations of less than one million.
- v. Retrieve the provinces that have populations of less than one million or greater than five million.
- vi. Retrieve the provinces that do not have populations of less than one million or greater than five million.
- vii. Retrieve the populations of provinces that have a land area greater than 200,000 square kilometers.
- viii. Retrieve the provinces along with their population densities (population divided by land area).
- ix. For this exercise, add a new table called *Capital* to the database. Capital has three columns—name (territory) (*lt<sup>2</sup>*), capital (*lt<sup>3</sup>*), and population (*lt<sup>10000</sup>*)—and it holds the data shown here:

| Province/Territory        | Capital       | Population |
|---------------------------|---------------|------------|
| Newfoundland and Labrador | St. John's    | 572 915    |
| Prairie: Newfoundland     | Charlottetown | 115 500    |
| Nova Scotia               | Halifax       | 950 100    |
| New Brunswick             | Fredericton   | 811 500    |
| Manitoba                  | Winnipeg      | 882 757    |
| Ontario                   | Toronto       | 4 692 897  |
| Manitoba                  | Winnipeg      | 671 274    |
| Saskatchewan              | Regina        | 1 028 000  |
| Alberta                   | Edmonton      | 937 845    |
| British Columbia          | Vancouver     | 3 119 022  |
| Yukon Territory           | Whitehorse    | 21 400     |
| Northwest Territories     | Yellowknife   | 19 541     |
| Eskimos                   | Iqaluit       | 5 286      |

Table 33 2001 Canadian Census Data: Capital City Populations

With SQL, you can test on the following:

- a. Review the contents of the table
  - b. Retrieve the populations of the provinces and capitals (in a list of tuples of the form [province population, capital population])
  - c. Retrieve the land area of the provinces whose capitals' total populations are greater than 100,000
  - d. Retrieve the provinces with land densities less than two people per square kilometer and capital city populations more than 500,000
  - e. Retrieve the total land area of Canada
  - f. Retrieve the average capital city population
  - g. Retrieve the lowest capital city population
  - h. Retrieve the highest province/territory population
  - i. Retrieve the provinces that have land densities within 0.0 persons per square kilometer of one another—have each pair of provinces reported only once
- A. Write a Python program that creates a new database and executes the following SQL statements. How do the results of the SELECT statements differ from what you would expect Python itself to do? Why?

```
CREATE TABLE Numbers(id INT);
INSERT INTO Numbers VALUES();
INSERT INTO Numbers VALUES();
SELECT * FROM Numbers WHERE id < 100
SELECT * FROM Numbers WHERE id > 100 AND id < 9
SELECT * FROM Numbers WHERE id < 0 AND id > 100
```

# Bibliography

- [DZM02] Alan Danzig, Brad Miller, and Chris Morris. *How to Think Like a Computer Scientist: Learning with Python*. CreateYourPress, Bedford, MA, 2002.
- [GZT9] Mark T. Goodwin and Richard T. Parsons. *Introduction to Cryptology, uses, Principles, and Perspectives*. A Mathematics Approach. Prentice Hall, Englewood Cliffs, NJ, Third, 2001.
- [GL07] Michael H. Goldwasser and Daniel M. Ronner. *Object Oriented Programming in Python*. Prentice Hall, Englewood Cliffs, NJ, 2007.
- [Haa06] Roger R. Hirsch, Marty Kacik, and Chungwu Chang. *Python for Data Intensive Task-Based and Grid Computing*. No, 2006.
- [Hoy05] D. J. Hoydman. *New Secure Document-type Identifier*. <http://www.djhoyle.com>, 2005.
- [Jac76] Ivan Jacobson. *Frogs and Feijoas*. Cambridge University Press, Cambridge, United Kingdom, 1976.
- [Lam04] Mark Turi. *Learn to Python: CRYPTO & AUTOMATION*. Inf. Sci. Develop., CS, Fifth, 2004.
- [Py11] Python ED. J-S G. Python/Maintainer Special Interest Group (J-S). 2011. Python ED. S G. <http://www.python.org/ftp/python/2.7.0/peps/current/already>, 2011.
- [Wat08] Jennifer M. Wing. Computational Thinking. <http://www.csail.mit.edu/~wing/cse6.0008/su08.html>.
- [Zel05] John Zelle. *Python Programming: An Introduction to Computer Science*. Franklin Beale & Associates, Winona, OK, 2005.

## Index

Symbole

- *join* operator, 103  
 - 246  
 - *map* operator, 141  
 - *partition*, 47, 86  
   - *empty string*, 86  
 - *n (number)* operator, 216  
 - *n (number) separator*, 111  
 - *newline character separator*, 22  
 - *padding*, 22  
**3. *operator***  
 - *add*, 5  
   - *function*, 31  
   - *number definition*, 25  
   - *string padding*, 21  
   - *for loops*, 506  
**4. *operators***  
 - *multiplication operator*,  
   - 2, 11  
   - *string repeat operator*, 66  
 - *correlation operator*,  
   - 2, 11  
 - *loop iteration operator*,  
   - 2, 11  
 - *array iteration operator*, 22  
 - *function iteration operator*, 22  
 - *conditional operator*, 12  
 - *(percentage) operator*,  
   - 22, 105  
 - *- (minus sign) operator*, 22  
   - 105  
 - *angle bracket operator*, 22, 14  
 - *bitmasking operator*, 15,  
   - 105



Exhibit 503

- *multiple values*, 230  
*multiple values*, 31  
*natural path*, 175  
*negative value*, 10  
*nonlocal value*, concept of function  
     *imported function*, 230  
*odd number*, 222  
*addition (+) operator*, 12  
*addition (+) operator*  $\Rightarrow$  *operator*, 22  
*adding three numbers*, 221  
*algorithm*  
     *always*, 253, 258, 259  
     *example*, 251  
     *executing the standard*  
         *calculus*, 221  
     *function*, 253  
     *loop*, 253  
     *using* *variables*, 253  
*Checking*, 128  
*International Standard Code for*  
     *Information Interchange*,  
     *ISO 7010*, 63  
*and operator*, 26  
*angle in degrees*, 12  
*arcsine function*, 120  
*approximate value*, 31, 42  
*arbitrary operator*, 10  
*area*, 220, 221  
*ASCII International Standard*  
     *Code for Information Interchange*,  
     *checkup*, 42  
*arithmetic method*, 502

- redundancy**  
multiple conditions (using  
operator), 209  
multiple conditions, 10  
multiple conditions, 101  
multiple conditions, 43  
  10, 72, 104, 145, 214  
**segmented suggested**, 21  
**sequential treatment**, 104  
selected, 21  
  multiple conditions  
  10, 11  
  multiple conditions, 101  
**sequential**, 212  
**semantics**  
  defined, 222  
  syntax, 221  
**separated assumptions**, 21  
**set function**, 171
- S**
- scheduled** (1), 176
- slice**, 4
- slimy operators**, 12
- smiley search**, 204, 214
- soft errors**, 214
- soft實施**, 245
- softly matched**, 242
- soft data type**, 202
- soft flow rule enforcement**,  
  10, 224
- soft type**, 22
- Soek, George**, 77
- soeken expression analysis**,  
  100, 202
- soeken logic**, 77
- soeken operators**  
  and, 28, 107  
  using conditions and  
  values with, 21
- soeken type**, 77
- soeken-type**, 250
- sorter engine**, 202
- sorted conditions**, 102
- boundary cases**, 200
- operator**, 5
- operator (1, 2, 3, 100)**
- operator (1)**  
  and, 107  
  multiple conditions, 214
- September 1995 (CEI)**
- operator (2)**
- operator (3)**
- operator (4)**
- operator (5)**
- operator (6)**
- operator (7)**
- operator (8)**
- operator (9)**
- operator (10)**
- operator (11)**
- operator (12)**
- operator (13)**
- operator (14)**
- operator (15)**
- operator (16)**
- operator (17)**
- operator (18)**
- operator (19)**
- operator (20)**
- operator (21)**
- operator (22)**
- operator (23)**
- operator (24)**
- operator (25)**
- operator (26)**
- operator (27)**
- operator (28)**
- operator (29)**
- operator (30)**
- operator (31)**
- operator (32)**
- operator (33)**
- operator (34)**
- operator (35)**
- operator (36)**
- operator (37)**
- operator (38)**
- operator (39)**
- operator (40)**
- operator (41)**
- operator (42)**
- operator (43)**
- operator (44)**
- operator (45)**
- operator (46)**
- operator (47)**
- operator (48)**
- operator (49)**
- operator (50)**
- operator (51)**
- operator (52)**
- operator (53)**
- operator (54)**
- operator (55)**
- operator (56)**
- operator (57)**
- operator (58)**
- operator (59)**
- operator (60)**
- operator (61)**
- operator (62)**
- operator (63)**
- operator (64)**
- operator (65)**
- operator (66)**
- operator (67)**
- operator (68)**
- operator (69)**
- operator (70)**
- operator (71)**
- operator (72)**
- operator (73)**
- operator (74)**
- operator (75)**
- operator (76)**
- operator (77)**
- operator (78)**
- operator (79)**
- operator (80)**
- operator (81)**
- operator (82)**
- operator (83)**
- operator (84)**
- operator (85)**
- operator (86)**
- operator (87)**
- operator (88)**
- operator (89)**
- operator (90)**
- operator (91)**
- operator (92)**
- operator (93)**
- operator (94)**
- operator (95)**
- operator (96)**
- operator (97)**
- operator (98)**
- operator (99)**
- operator (100)**
- operator (101)**
- operator (102)**
- operator (103)**
- operator (104)**
- operator (105)**
- operator (106)**
- operator (107)**
- operator (108)**
- operator (109)**
- operator (110)**
- operator (111)**
- operator (112)**
- operator (113)**
- operator (114)**
- operator (115)**
- operator (116)**
- operator (117)**
- operator (118)**
- operator (119)**
- operator (120)**
- operator (121)**
- operator (122)**
- operator (123)**
- operator (124)**
- operator (125)**
- operator (126)**
- operator (127)**
- operator (128)**
- operator (129)**
- operator (130)**
- operator (131)**
- operator (132)**
- operator (133)**
- operator (134)**
- operator (135)**
- operator (136)**
- operator (137)**
- operator (138)**
- operator (139)**
- operator (140)**
- operator (141)**
- operator (142)**
- operator (143)**
- operator (144)**
- operator (145)**
- operator (146)**
- operator (147)**
- operator (148)**
- operator (149)**
- operator (150)**
- operator (151)**
- operator (152)**
- operator (153)**
- operator (154)**
- operator (155)**
- operator (156)**
- operator (157)**
- operator (158)**
- operator (159)**
- operator (160)**
- operator (161)**
- operator (162)**
- operator (163)**
- operator (164)**
- operator (165)**
- operator (166)**
- operator (167)**
- operator (168)**
- operator (169)**
- operator (170)**
- operator (171)**
- operator (172)**
- operator (173)**
- operator (174)**
- operator (175)**
- operator (176)**
- operator (177)**
- operator (178)**
- operator (179)**
- operator (180)**
- operator (181)**
- operator (182)**
- operator (183)**
- operator (184)**
- operator (185)**
- operator (186)**
- operator (187)**
- operator (188)**
- operator (189)**
- operator (190)**
- operator (191)**
- operator (192)**
- operator (193)**
- operator (194)**
- operator (195)**
- operator (196)**
- operator (197)**
- operator (198)**
- operator (199)**
- operator (200)**
- operator (201)**
- operator (202)**
- operator (203)**
- operator (204)**
- operator (205)**
- operator (206)**
- operator (207)**
- operator (208)**
- operator (209)**
- operator (210)**
- operator (211)**
- operator (212)**
- operator (213)**
- operator (214)**
- operator (215)**
- operator (216)**
- operator (217)**
- operator (218)**
- operator (219)**
- operator (220)**
- operator (221)**
- operator (222)**
- operator (223)**
- operator (224)**
- operator (225)**
- operator (226)**
- operator (227)**
- operator (228)**
- operator (229)**
- operator (230)**
- operator (231)**
- operator (232)**
- operator (233)**
- operator (234)**
- operator (235)**
- operator (236)**
- operator (237)**
- operator (238)**
- operator (239)**
- operator (240)**
- operator (241)**
- operator (242)**
- operator (243)**
- operator (244)**
- operator (245)**
- operator (246)**
- operator (247)**
- operator (248)**
- operator (249)**
- operator (250)**
- operator (251)**
- operator (252)**
- operator (253)**
- operator (254)**
- operator (255)**
- operator (256)**
- operator (257)**
- operator (258)**
- operator (259)**
- operator (260)**
- operator (261)**
- operator (262)**
- operator (263)**
- operator (264)**
- operator (265)**
- operator (266)**
- operator (267)**
- operator (268)**
- operator (269)**
- operator (270)**
- operator (271)**
- operator (272)**
- operator (273)**
- operator (274)**
- operator (275)**
- operator (276)**
- operator (277)**
- operator (278)**
- operator (279)**
- operator (280)**
- operator (281)**
- operator (282)**
- operator (283)**
- operator (284)**
- operator (285)**
- operator (286)**
- operator (287)**
- operator (288)**
- operator (289)**
- operator (290)**
- operator (291)**
- operator (292)**
- operator (293)**
- operator (294)**
- operator (295)**
- operator (296)**
- operator (297)**
- operator (298)**
- operator (299)**
- operator (300)**
- operator (301)**
- operator (302)**
- operator (303)**
- operator (304)**
- operator (305)**
- operator (306)**
- operator (307)**
- operator (308)**
- operator (309)**
- operator (310)**
- operator (311)**
- operator (312)**
- operator (313)**
- operator (314)**
- operator (315)**
- operator (316)**
- operator (317)**
- operator (318)**
- operator (319)**
- operator (320)**
- operator (321)**
- operator (322)**
- operator (323)**
- operator (324)**
- operator (325)**
- operator (326)**
- operator (327)**
- operator (328)**
- operator (329)**
- operator (330)**
- operator (331)**
- operator (332)**
- operator (333)**
- operator (334)**
- operator (335)**
- operator (336)**
- operator (337)**
- operator (338)**
- operator (339)**
- operator (340)**
- operator (341)**
- operator (342)**
- operator (343)**
- operator (344)**
- operator (345)**
- operator (346)**
- operator (347)**
- operator (348)**
- operator (349)**
- operator (350)**
- operator (351)**
- operator (352)**
- operator (353)**
- operator (354)**
- operator (355)**
- operator (356)**
- operator (357)**
- operator (358)**
- operator (359)**
- operator (360)**
- operator (361)**
- operator (362)**
- operator (363)**
- operator (364)**
- operator (365)**
- operator (366)**
- operator (367)**
- operator (368)**
- operator (369)**
- operator (370)**
- operator (371)**
- operator (372)**
- operator (373)**
- operator (374)**
- operator (375)**
- operator (376)**
- operator (377)**
- operator (378)**
- operator (379)**
- operator (380)**
- operator (381)**
- operator (382)**
- operator (383)**
- operator (384)**
- operator (385)**
- operator (386)**
- operator (387)**
- operator (388)**
- operator (389)**
- operator (390)**
- operator (391)**
- operator (392)**
- operator (393)**
- operator (394)**
- operator (395)**
- operator (396)**
- operator (397)**
- operator (398)**
- operator (399)**
- operator (400)**
- operator (401)**
- operator (402)**
- operator (403)**
- operator (404)**
- operator (405)**
- operator (406)**
- operator (407)**
- operator (408)**
- operator (409)**
- operator (410)**
- operator (411)**
- operator (412)**
- operator (413)**
- operator (414)**
- operator (415)**
- operator (416)**
- operator (417)**
- operator (418)**
- operator (419)**
- operator (420)**
- operator (421)**
- operator (422)**
- operator (423)**
- operator (424)**
- operator (425)**
- operator (426)**
- operator (427)**
- operator (428)**
- operator (429)**
- operator (430)**
- operator (431)**
- operator (432)**
- operator (433)**
- operator (434)**
- operator (435)**
- operator (436)**
- operator (437)**
- operator (438)**
- operator (439)**
- operator (440)**
- operator (441)**
- operator (442)**
- operator (443)**
- operator (444)**
- operator (445)**
- operator (446)**
- operator (447)**
- operator (448)**
- operator (449)**
- operator (450)**
- operator (451)**
- operator (452)**
- operator (453)**
- operator (454)**
- operator (455)**
- operator (456)**
- operator (457)**
- operator (458)**
- operator (459)**
- operator (460)**
- operator (461)**
- operator (462)**
- operator (463)**
- operator (464)**
- operator (465)**
- operator (466)**
- operator (467)**
- operator (468)**
- operator (469)**
- operator (470)**
- operator (471)**
- operator (472)**
- operator (473)**
- operator (474)**
- operator (475)**
- operator (476)**
- operator (477)**
- operator (478)**
- operator (479)**
- operator (480)**
- operator (481)**
- operator (482)**
- operator (483)**
- operator (484)**
- operator (485)**
- operator (486)**
- operator (487)**
- operator (488)**
- operator (489)**
- operator (490)**
- operator (491)**
- operator (492)**
- operator (493)**
- operator (494)**
- operator (495)**
- operator (496)**
- operator (497)**
- operator (498)**
- operator (499)**
- operator (500)**
- operator (501)**
- operator (502)**
- operator (503)**
- operator (504)**
- operator (505)**
- operator (506)**
- operator (507)**
- operator (508)**
- operator (509)**
- operator (510)**
- operator (511)**
- operator (512)**
- operator (513)**
- operator (514)**
- <p





- J**
- Java.** *Dotfiles operations in*, 52
  - Java8.** 200
  - jboss.** *Using to monitor the host machine*, 118
- K**
- keybinds, adding/removing**, 351
  - keys.** 253
  - keyword arguments (lexical).** 32
  - keybinding Python.** 198
  - keymap, keyboard configuration**, 25
- L**
- lambda.** 310
  - lambda functions.** 324
  - laptops.** 312
  - laptop Linux.** 322
  - laptop/python.** 60, 246
  - last time or equal to last step command.** 30, 246
  - lexicographic ordering.** 40
  - less/grep command.** 261
  - Linux, installing Python on**, 176
  - list search.** 235
  - lists.** 124
  - list comprehension.** 212
  - lists.**
    - append.** 346
    - counting elements.** 129
    - finding.** 126
    - length.** 127
    - map.** 127
    - remove.** 126
    - reversing.** 127
    - slice.** 126
    - sort.** 126
    - update.** 126
  - list comprehension.** 124
  - list of parameters.** 126
  - listened.** 156
  - listening port.** 124
  - parallel.** 154
  - procedure name in.** 145
  - proximity testing indices.** 122
  - ranged.** 157
  - scanning.** 212
  - slicing.** 127
- M**
- maps.** 127
  - match method.** 218
  - maps, and dictionaries.** 218
  - matchable.** 106, 212
  - matchable.** 106
  - map/reduce, map/reduce**, 126
  - memory address.** 16, 64
  - memory model.** 32, 40
  - method.** 121
  - methodicity.** 210
  - method function.** 233
  - merging sorted lists.** 211
  - metaprograms.** 210
  - metaprograms.**
    - code.** 125
    - meta.** 210
    - open.** 125
    - read.** 125
    - readline.** 210
  - metatypes.** 122
  - metaclass.**
    - about.** 12
    - base.** 346
    - \_\_b, \_\_m.** 224

1

overwriting *new*, 113  
expanding operators, 12, 13  
negative operators, 11  
ranged references, 102  
reduced table, 101  
redundant operators, 107  
reading keys, 104  
readline, 20  
readline command (`rl`), 10  
readline, 107  
remaking tables, 70  
set operator, 70  
set operator (operator), 103,  
111  
sort in the reader, 107A  
SQL, 842  
surrogate, 152  
tuples  
    keeping overwrites of,  
        120  
    using with `select` operator  
        102

Journal of Health Politics  
Volume 34 Number 3

1

where it is shown that  
expanding  $\mathcal{E}(\mathbf{z})$ , (7),  
into its expansion around  $\mathbf{z}_0$ , (3),  
and then multiplying by  $\mathbf{z}$ , (1),  
one obtains the particular (13)  
containing integrated methods.  
Q.E.D.

T

**John Throckmorton  
John Throckmorton  
John Throckmorton**

## **ANSWERED** (X) **OPERATOR**, 11 ANSWERED, 111

Jahresbericht 2011

Wiley Database 2000

#### REFERENCES AND NOTES

protection, 14  
protection, 14

John F. Kennedy Library, Boston, Massachusetts

ANSWER SECTION 270  
ANSWER

*Journalism in America.* 19  
Vol. II. No. 347.

1990-1991 Academic Year

APPENDIX B  
AEROSOLIZED AND DROPOUT

## WATER PROGRAM

**ANSWER**

### **REFERENCES AND BIBLIOGRAPHY**

#### REFERENCES

- join condition, 248  
join query, 248
- programs**
- about, 2
  - printing, 528
  - by user, 27
  - running, 7, 32
  - writing, 26
- printing, 2, 57
- Protocol Data Unit (PDU) types, 181
- Python**
- about, 101
  - functions, 2
  - operator overloading, 13–16
  - scripting code, 10
  - stack traces, 31
  - string managers, 14
  - variables, 17
  - operators, 11
  - operations, 4, 27
  - processes, 31
  - printing, 2, 57
  - proxies, 14
  - keywords, 23
  - tracking code readability, 56
  - modules, 11
  - value tower, 17
  - program, 27
  - scripted programs, 10, 7
  - structured query language (SQL), 201
  - statements, 23
  - timestamp values, 10, 101
  - tuples, 17, 18
  - values, 18
  - variables, 10, 11, 23
  - variables, 201
- PyQt (Python port of Qt)
- PyQt stylesheet, 250
- 
- D**
- data (query statements), 207
- dates, 111, 109
- query conditions, 542
- queries, 47, 63
- 
- R**
- random module, 202
- range function, 121
- ranges of numbers, listing, 200, 150
- Read methods, 272
- reading files, 20, 102
- Reduction techniques, 175, 253
- relationships, 107
- SQL data types, 242
- real number precision of, 11
- remote database modules, 18
- records, 129, 191, 229
- relational database.
- SQL database
  - SQL operations, 60–82
  - values path, 173
  - views, 200
  - where clause, 101
- remove\_dups function, 255
- removing code, 207
- removal operator, 107
- remove method, 253
- removing module, 203
- removed data, 207
- reusable code, 101, 104
- reusing module names, 21
- REPL session, 177
- ReLU (rectified linear unit) model, 216
- REDFILE, 207
- related task automation, 222
- remote windows, 212, 282
- remove function, 101
- empty or null value, 101
  - strip method, 121
- removing, processes, 7, 59
- removing noise, measuring, 100–101, 103
- filter search, 205
  - image search, 204
- remove() method, 101
- 
- S**
- saving stamp to database, 204
- Scalable programming (SPr), 204
- scripting
- about, 127, 205
  - binary search, 205
  - functions, 206
  - loop, 127
  - or smallest value, 206
  - using functions, 203
- scripted command, 212
- select function, 201
- SELECT parameter, 274
- self-join, 250, 399
- semantic errors, 52
- semantics, 213
- set type, 199
- sets
- about, 106
  - defined, 199
  - complement, 212
  - intersection, 217
  - operations-on, 100
- shannon, Claude, 77
- short, default, 18
- short lived connection, 54
- size argument, 592
- static classes, 66
- stack overflow, 101
- stackless, 102
- stack method, 250
- stack function, 101
- stacks
- about, 107, 209, 210
  - data flows, 209
  - operations, 209
  - push and pop, 209
  - remove and, 209
  - return stack, 209
- space complexity, 101
- special characters, 205, 210
- stack overflow, 102
- stack method, 127
- SQL Structured Query Language, 207–210, 242
- SQLite, 101
- SQL injection, 101
- signature analysis
- about, 2
  - inconsistency when, 202
  - empty form, 101
- softmax function, 121
- softmaxic
- activation, 12, 13, 72
  - 1, 11, 108, 201, 2
  - bias, 161
  - cheating, which to use, 100
  - cure, 66
  - values, 183
  - weight bias, 77
- SQLITE mode, 242
- defined, 7
- 8, 57, 92
- input, 100
- map, 100, 1, 101
- System, 27
- 14, 15, 16, 91





# Tinker, Tailor, Solder, and DIY!

Get into the DIY spirit with Raspberry Pi or Arduino. Whether you're a total beginner or

The Raspberry Pi is a cool, full-blown mini-computer that costs less than \$50. Its sales, mailing list, and forum of 100,000+ members worldwide make it easy to learn to build hardware—your own robot lab book gives you everything you need to get started.

**Mark Ormanek**

314 pages | ISBN 9781430255040 | \$17  
A hands-on guide with step-by-step projects



Arduino is an open-source platform that makes DIY electronics projects easy for everyone. Even if you have no electronics experience, you'll learn how to put it to work with fun projects, including step-by-step instructions showing you how to build a behavioral mouse, a motion-sensing game controller, and many other fun Arduino projects. This book has now been updated for Arduino 1.0, with revised code, exercises, and coverage of the latest. We've changed all the projects to use the official Arduino IDE's new features.

Published price: approximately \$40.00. ISBN 9781430255057 | \$35  
A hands-on guide with step-by-step projects



**Mark Ormanek**

3572 pages | ISBN 9781430255064 | \$35  
A hands-on guide with step-by-step projects

# Kick your Career up a Notch

Ready to blog or promote yourself online? Does it take knowledge of personal branding? We've got you covered.

Technology gives the feedback to your blog's track progress, so it can quickly and easily inform of your experience has w/ toward successful bloggers. There is no magic to successful blogging; with this book you'll learn the techniques to start, and keep a large audience of local, nearby readers and keep your this popular. Order now and start today!

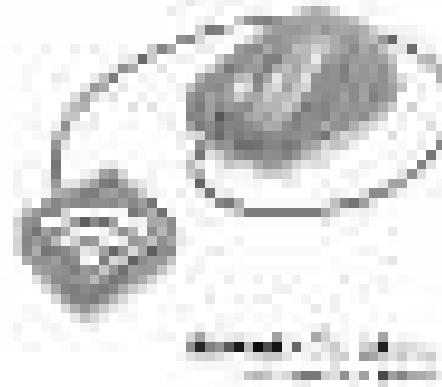
Published in paper format, \$21.95, available for \$16.95 with shipping.

**Author Bio:**

285 pages ISBN 0761224306553, \$22  
http://www.safaribooksonline.com

## Technical Blogging

By: Michael S. Hiltz  
Stern, Michael S. Hiltz



Michael S. Hiltz

Author biography, information, links, and more can be found at [www.michaelshiltz.com](http://www.michaelshiltz.com). You have access to 20+ chapters in addition. You can programmatic techniques for power PC development web applications. New technologies are here's the guide to what you need to be aware of. With this book you'll find the tools to help you succeed fast and easy.

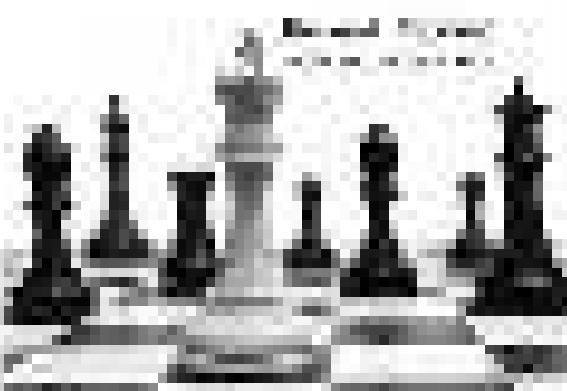
Author Bio: Michael S. Hiltz, Michael S. Hiltz

**Author Bio:**

285 pages ISBN 0761224306553, \$22  
http://www.safaribooksonline.com

## The Developer's Code Book

By: Michael S. Hiltz, Michael S. Hiltz



Michael S. Hiltz

# Seven Databases, Seven Languages

You've got much more to learn with the Java API in MySQL, databases, and lots and lots of languages a year. How about seven?

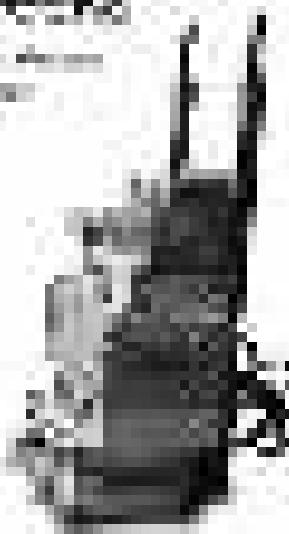
Java, PostgreSQL, and more complete for the day, and you can learn them in half the time. *Seven Databases in Seven Weeks* (O'Reilly MySQL, \$39.99) by Scott Winters takes you on a tour of some of the most popular open source databases today. In the tradition of Timon K. Gehr's *Seven Languages in Seven Weeks*, this book gives you a year's worth of material to study and learn the concepts of the most useful technologies.

**Eric Badger and Jim H. Wilson**  
350 pages | ISBN: 9781449312600 | \$39.99  
<http://oreil.ly/7db7w>



## Seven Databases in Seven Weeks

Open source databases for the  
modern developer



Eric Badger  
Jim H. Wilson

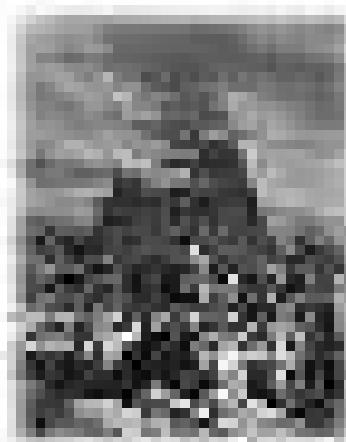
"You should learn a programming language every year, as recommended by the Project on Professional IT Competency. In just one short year, *Seven Languages in Seven Weeks* will get you up-to-speed on C, Python, Haskell, Erlang, Scala, Ruby, and Racket. Whether or not your favorite language is on that list, you'll broaden your perspective of programming by examining these languages, and in so doing, you'll learn how to learn a language quickly."

**Eric S. Raymond**  
350 pages | ISBN: 9781449312600 | \$39.99  
<http://oreil.ly/7l7w>



## Seven Languages in Seven Weeks

Eric S. Raymond  
Haskell  
Erlang  
Python  
Ruby  
Scala  
C



Eric S. Raymond

# Put the "Fun" in Functional

Even pros like Tim, back into functional programming, can say it's not robust, reliable, or industrial-strength enough for C#.

You want to apply functional programming, because you really believe it can help you solve more problems than you've dreamt. You know you need functional applications, but also know there are almost impossible to port over. Meet Elixir, a functional concurrent language built on the rock-solid Erlang VM. Elixir programs start fast, and will never crash, but they're not yet well-taught to produce, and keep you informed for the long haul. This book is the *Introduction to Elixir for experienced programmers*.

**Dave Thomas**

ISBN 978-1-4919-2770-1 \$35  
[pragprog.com/titles/elixir](http://pragprog.com/titles/elixir)



## Programming Elixir

David  
Thomas

Introduction  
to functional  
concurrent  
languages

Pragmatic  
Programmers

http://pragprog.com/titles/elixir

A multi-user game system, cloud application, or networked database can have thousands of users all interacting at the same time. You need a powerful tool of strength to handle them easily but quickly, without crashing or crashing, without losing moments. In *Concurrent Programming with Elixir and Erlang*, you'll learn how to write parallel programs that scale effectively on multicore systems.

**Joe Armstrong**

ISBN 978-1-4919-2817-0 \$35  
[pragprog.com/titles/jaerlang](http://pragprog.com/titles/jaerlang)



## Concurrent Programming with Elixir and Erlang

Joe  
Armstrong

Pragmatic  
Programmers

http://pragprog.com/titles/jaerlang

Prag

# Long live the command line!

Discover just how far you can go toward productivity.

Your mouse is slowing you down. The time you spend context switching between your editor and your terminal costs you time and productivity. That's why tooling your terminal with basic automation capabilities that are specific to your workflow. Learn how to customize, extend, and leverage these unique abilities and keep your fingers on your keyboard forever.

John P. Mueller

<http://www.pragprog.com/titles/jmcl>



Wim is a fast and efficient text editor that will make you a faster and more efficient developer. It's available on almost every OS—if you need the features of the best, well-known editors like Vim or Emacs, but in less than 100 lines of code, you'll quickly know the value a well-intentionally optimized, yet simple editing and writing tools.

Dave Hall

<http://www.pragprog.com/titles/dhllw>





# Be Agile

Don't just read up; you have to be agile. With these, you have.

The best agile book isn't a book: *Agile in a Flash* is a compact deck of cards that fits easily in your pocket. You can flip through them at your desk, or during a meeting, or while you're taking a break from work. These cards are meant to be used, not just read.

Jeff Langr and Tim Cragg

(110 pages) ISBN 978-0321825673, \$15  
<http://www.agileinflash.com>



Here are three simple truths about software development:

1. You can't anticipate all the requirements upfront. The requirements you identify will change. There is always more to do than time and resources allow. This is the nature of software development. By embracing this, you can stay more involved and by learning to live with uncertainty, professional, capable and dispatching one more difficult software project and the toughest delivery schedules, with ease and joy.

Jonathan Edwards

(300 pages) ISBN 978-0984156001  
<http://www.agilesamurai.com>

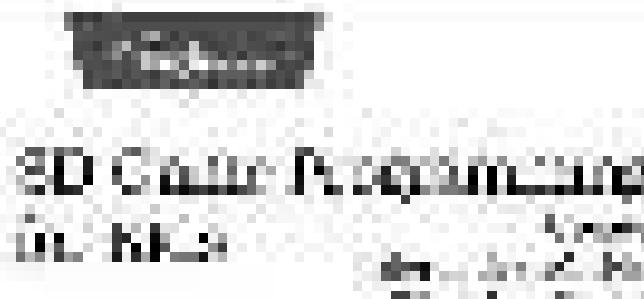


# The Joy of Math and Programming

Discover the joy and rewarding challenges of pure mathematics with your kids' first introduction to JavaScape.

We're all born mathematicians, but as we grow older, we tend to forget that. This book will help you remember. With 50 fun, enlightening topics from my 20+ years of teaching: from Egyptian fractions to Turing machines; from the real meaning of numbers to proofs, group symmetry, and mechanical computation. If you've ever wondered what lies beyond the problems you struggled through in high school geometry, or what limits the capabilities of the computers on your desk, this is the book for you!

Mark C. Chu-Carroll  
2012 paper cover: \$29.99/£17.99/CA\$36.99  
<http://www.mazabooks.com>



You have probably talked more than played games? Creating your own? Even if you're an absolute beginner, this book will teach you how to make your own online games with descriptive examples. You'll learn how to program using existing game engines, and see how others have done it. You'll learn how to build programs using built-in visual programming languages like Scratch, the language of the web. You'll be amazed at what you can do as you build interactive games and fun games.

2012 paper cover  
2012 digital download  
2012 digital download

# The Pragmatic Bookshelf

The Pragmatic Bookshelf offers books written by developers for developers. That's the audience we serve. From the pragmatic solutions you'll find in our books, to our blog posts and reviews, to our events, podcasts, and more, PragPub, the Pragmatic Bookshelf team will be there with more tools and products to help you stay on top of your game.

## Visit Us Online

### [The Book's Home Page](#)

[PragPub.com/books/](#)

Searchable catalog, book reviews, and other resources from our online bookstore and

### [Register for Updates](#)

[PragPub.com/newsletter](#)

Recipients of updates and new book reviews via email

### [Join the Community](#)

[PragPub.com/community](#)

Read our writing, participate in discussions, contribute, and connect with other members of our Pragmatic Bookshelf community.

### [New and Noteworthy](#)

[PragPub.com/news](#)

Keep up the latest, people's reviews, news, and other stuff.

## Buy the Book

Buy the book off us, perhaps you'd like to have paper copy of the book. US residents  
for purchases of more than \$100, we can ship free!

## Contact Us

### [Order Details](#)

[Customer Service](#)

**International Rights:** [Int'lRights@pragprog.com](#)

### [Author Info](#)

[Author's Bio](#)

[About](#)

[My PragProg Account](#)

[My Order History](#)

**Books I've Written:** [mybooks@pragprog.com](#)

[Bookmarks](#)

[Bookmarks by Author](#)

[+1 800 227 7771](#)