Introduction to Computation and Programming Using Python

With Application to Understanding Data

second edition

John V. Guttag

Introduction to Computation and Programming Using Python

with Application to Understanding Data

Introduction to Computation and Programming Using Python
with Application to Understanding Data

Second Edition

John V. Guttag

The MIT Press
Cambridge, Massachusetts
London, England

Printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Names: Guttag, John, author.
Title: Introduction to computation and programming using Python : with application to understanding data / John V. Guttag.
Description: Second edition. | Cambridge, MA : The MIT Press, [2017] | Includes index.
Identifiers: LCCN 2016019367 | ISBN 9780262529624 (pbk. : alk. paper)
Subjects: LCSH: Python (Computer program language)--Textbooks. | Computer programming--Textbooks.
Classification: LCC QA76.73.P98 G88 2017 | DDC 005.13/3--dc23 LC record available at https://lccn.loc.gov/2016019367

10 9 8 7 6 5 4 3 2 1

To my family:

Olga
David
Andrea
Michael
Mark
Addie

CONTENTS

PREFACE .... **xiii**

ACKNOWLEDGMENTS .... **xvii**

1 GETTING STARTED **1**

2 INTRODUCTION TO PYTHON **7**

2.1 The Basic Elements of Python 9

2.1.1 Objects, Expressions, and Numerical Types 9

2.1.2 Variables and Assignment 12

2.1.3 Python IDE's 14

2.2 Branching Programs 15

2.3 Strings and Input 18

2.3.1 Input 20

2.3.2 A Digression About Character Encoding 21

2.4 Iteration 22

3 SOME SIMPLE NUMERICAL PROGRAMS **25**

3.1 Exhaustive Enumeration 25

3.2 For Loops 27

3.3 Approximate Solutions and Bisection Search 30

3.4 A Few Words About Using Floats 34

3.5 Newton-Raphson 37

4 FUNCTIONS, SCOPING, AND ABSTRACTION **39**

4.1 Functions and Scoping 40

4.1.1 Function Definitions 40

4.1.2 Keyword Arguments and Default Values 42

4.1.3 Scoping 43

4.2 Specifications 47

4.3 Recursion 50

4.3.1 Fibonacci Numbers 52

4.3.2 Palindromes 54

4.4 Global Variables 57

4.5 Modules 59

4.6 Files 61

5 STRUCTURED TYPES, MUTABILITY, AND HIGHER-ORDER FUNCTIONS 65
5.1 Tuples 65
5.1.1 Sequences and Multiple Assignment 67
5.2 Ranges 67
5.3 Lists and Mutability 68
5.3.1 Cloning 73
5.3.2 List Comprehension 74
5.4 Functions as Objects 75
5.5 Strings, Tuples, Ranges, and Lists 77
5.6 Dictionaries 79
6 TESTING AND DEBUGGING 85
6.1 Testing 86
6.1.1 Black-Box Testing 87
6.1.2 Glass-box Testing 88
6.1.3 Conducting Tests 90
6.2 Debugging 92
6.2.1 Learning to Debug 94
6.2.2 Designing the Experiment 95
6.2.3 When the Going Gets Tough 98
6.2.4 When You Have Found "The" Bug 99
7 EXCEPTIONS AND ASSERTIONS 101
7.1 Handling Exceptions 101
7.2 Exceptions as a Control Flow Mechanism 105
7.3 Assertions 108
8 CLASSES AND OBJECT-ORIENTED PROGRAMMING 109
8.1 Abstract Data Types and Classes 109
8.1.1 Designing Programs Using Abstract Data Types 114
8.1.2 Using Classes to Keep Track of Students and Faculty 115
8.2 Inheritance 118
8.2.1 Multiple Levels of Inheritance 121
8.2.2 The Substitution Principle 123
8.3 Encapsulation and Information Hiding 123
8.3.1 Generators 128
8.4 Mortgages, an Extended Example 130
9 A SIMPLISTIC INTRODUCTION TO ALGORITHMIC COMPLEXITY 135
9.1 Thinking About Computational Complexity 135
9.2 Asymptotic Notation 139

9.3 Some Important Complexity Classes 141
9.3.1 Constant Complexity 141
9.3.2 Logarithmic Complexity 141
9.3.3 Linear Complexity 142
9.3.4 Log-Linear Complexity 144
9.3.5 Polynomial Complexity 144
9.3.6 Exponential Complexity 145
9.3.7 Comparisons of Complexity Classes 147
10 SOME SIMPLE ALGORITHMS AND DATA STRUCTURES 151
10.1 Search Algorithms 152
10.1.1 Linear Search and Using Indirection to Access Elements 153
10.1.2 Binary Search and Exploiting Assumptions 154
10.2 Sorting Algorithms 158
10.2.1 Merge Sort 159
10.2.2 Exploiting Functions as Parameters 162
10.2.3 Sorting in Python 162
10.3 Hash Tables 164
11 PLOTTING AND MORE ABOUT CLASSES 169
11.1 Plotting Using PyLab 169
11.2 Plotting Mortgages, an Extended Example 175
12 KNAPSACK AND GRAPH OPTIMIZATION PROBLEMS 183
12.1 Knapsack Problems 184
12.1.1 Greedy Algorithms 184
12.1.2 An Optimal Solution to the 0/1 Knapsack Problem 188
12.2 Graph Optimization Problems 190
12.2.1 Some Classic Graph-Theoretic Problems 195
12.2.2 Shortest Path: Depth-First Search and Breadth-First Search 196
13 DYNAMIC PROGRAMMING 203
13.1 Fibonacci Sequences, Revisited 203
13.2 Dynamic Programming and the 0/1 Knapsack Problem 205
13.3 Dynamic Programming and Divide-and-Conquer 213
14 RANDOM WALKS AND MORE ABOUT DATA VISUALIZATION 215
14.1 Random Walks 216
14.2 The Drunkard's Walk 217
14.3 Biased Random Walks 224
14.4 Treacherous Fields 231

15 STOCHASTIC PROGRAMS, PROBABILITY, AND DISTRIBUTIONS 235
15.1 Stochastic Programs 236
15.2 Calculating Simple Probabilities 238
15.3 Inferential Statistics 239
15.4 Distributions 254
15.4.1 Probability Distributions 256
15.4.2 Normal Distributions 258
15.4.3 Continuous and Discrete Uniform Distributions 263
15.4.4 Binomial and Multinomial Distributions 264
15.4.5 Exponential and Geometric Distributions 265
15.4.6 Benford's Distribution 269
15.5 Hashing and Collisions 269
15.6 How Often Does the Better Team Win? 272
16 MONTE CARLO SIMULATION 275
16.1 Pascal's Problem 276
16.2 Pass or Don't Pass? 277
16.3 Using Table Lookup to Improve Performance 282
16.4 Finding π 283
16.5 Some Closing Remarks About Simulation Models 288
17 SAMPLING AND CONFIDENCE INTERVALS 291
17.1 Sampling the Boston Marathon 292
17.2 The Central Limit Theorem 298
17.3 Standard Error of the Mean 302
18 UNDERSTANDING EXPERIMENTAL DATA 305
18.1 The Behavior of Springs 305
18.1.1 Using Linear Regression to Find a Fit 309
18.2 The Behavior of Projectiles 314
18.2.1 Coefficient of Determination 317
18.2.2 Using a Computational Model 319
18.3 Fitting Exponentially Distributed Data 320
18.4 When Theory Is Missing 324
19 RANDOMIZED TRIALS AND HYPOTHESIS CHECKING 327
19.1 Checking Significance 328
19.2 Beware of P-values 334
19.3 One-tail and One-sample Tests 336
19.4 Significant or Not? 338
19.5 Which N? 340

19.6 Multiple Hypotheses....342
20 CONDITIONAL PROBABILITY AND BAYESIAN STATISTICS 345
20.1 Conditional Probabilities....346
20.2 Bayes' Theorem....348
20.3 Bayesian Updating....350
21 LIES, DAMNED LIES, AND STATISTICS....355
21.1 Garbage In Garbage Out (GIGO)....355
21.2 Tests Are Imperfect....356
21.3 Pictures Can Be Deceiving....357
21.4 Cum Hoc Ergo Propter Hoc....359
21.5 Statistical Measures Don't Tell the Whole Story....361
21.6 Sampling Bias....362
21.7 Context Matters....363
21.8 Beware of Extrapolation....364
21.9 The Texas Sharpshooter Fallacy....364
21.10 Percentages Can Confuse....367
21.11 Statistically Significant Differences Can Be Insignificant....368
21.12 The Regressive Fallacy....369
21.13 Just Beware....370
22 A QUICK LOOK AT MACHINE LEARNING....371
22.1 Feature Vectors....374
22.2 Distance Metrics....377
23 CLUSTERING....383
23.1 Class Cluster....385
23.2 K-means Clustering....387
23.3 A Contrived Example....390
23.4 A Less Contrived Example....395
24 CLASSIFICATION METHODS....403
24.1 Evaluating Classifiers....403
24.2 Predicting the Gender of Runners....408
24.3 K-nearest Neighbors....408
24.4 Regression-based Classifiers....415
24.5 Surviving the Titanic....425
24.6 Wrapping Up....430
PYTHON 3.5 QUICK REFERENCE....431
INDEX....435

PREFACE

This book is based on courses that have been offered at MIT since 2006, and as "Massive Online Open Courses" (MOOCs) through edX and MITx since 2012. The first edition of the book was based on a single one-semester course. However, over time I couldn't resist adding more material than could be fit into a semester. The current edition is suitable for a two-semester introductory computer science sequence.

When I started working on the second edition I thought that I would just add a few chapters, but I ended up doing far more. I reorganized the back half of the book, and converted the entire book from Python 2 to Python 3.

The book is aimed at students with little or no prior programming experience who have a desire to understand computational approaches to problem solving. For some of the students the material in this book will be a stepping stone to more advanced computer science courses. But for many of the students it will be their only formal exposure to computer science.

Because this will be the only formal exposure to computer science for many of the students, we emphasize breadth rather than depth. The goal is to provide students with a brief introduction to many topics, so that they will have an idea of what's possible when the time comes to think about how to use computation to accomplish a goal. That said, this is not a "computation appreciation" book. It is challenging and rigorous. Students who wish to really learn the material will have to spend a lot of time and effort learning to bend the computer to their will.

The main goal of this book is to help students become skillful at making productive use of computational techniques. They should learn to use computational modes of thoughts to frame problems and to guide the process of extracting information from data. The primary knowledge they will take away from this book is the art of computational problem solving.

This book is not easily slotted into a conventional computer science curriculum. Chapters 1-11 contain the kind of material typically included in a computer science course aimed at students with little or no programming experience. Chapters 12-14 contain slightly more advanced material, various subsets of which could be added to the introductory course if the students are more advanced. Chapters 15-24 are about using computation to help understand data.

They cover the material that we think should become the usual second course in a computer science curriculum (replacing the traditional data structures course).

In Chapters 1-11, we braid together four strands of material:

- The basics of programming,
- The Python 3 programming language,
- Computational problem solving techniques,
- Computational complexity, and
- Using plots to present information.

We cover most of Python's features, but the emphasis is on what one can do with a programming language, not on the language itself. For example, by the end of Chapter 3 the book has covered only a small fraction of Python, but it has already introduced the notions of exhaustive enumeration, guess-and-check algorithms, bisection search, and efficient approximation algorithms. We introduce features of Python throughout the book. Similarly, we introduce aspects of programming methods throughout the book. The idea is to help students learn Python and how to be a good programmer in the context of using computation to solve interesting problems.

The examples in this book have been tested using Python 3.5. Python 3 cleaned up many of the inconsistencies in the design of the various releases of Python 2 (often referred to as Python 2.x). However, it is not backward compatible. That meant that most programs written using Python 2 cannot be run using implementations of Python 3. For that reason, Python 2.x continues to be widely used. The first time we use features of Python 3 that differ from Python 2, we point out how the same thing could be accomplished in Python 2. All of the examples in this book are available online in both Python 3.5 and Python 2.7.

Chapters 12-13 provide an introduction to optimization, an important topic not usually covered in introductory courses. Chapters 14-16 provide an introduction to stochastic programs, another important topic not usually covered in introductory courses. Our experience at MIT is that we can can cover either Chapters 12-13 or Chapters 15-16, but not both, in our one-semester introductory course.

Chapters 15-24 are designed to provide a self-contained introduction to using computation to help understand data. They assume no knowledge of mathematics beyond high school algebra, but do assume that the reader is comfortable with rigorous thinking and is not intimidated by mathematical concepts. This part of the book is devoted to topics not found in most introductory texts: data visualization, simulation models, probabilistic and statistical thinking, and machine learning. We believe that this is a far more relevant body of material for most students than what is typically covered in the second computer science course.

We chose not to include problems at the end of chapters. Instead we inserted "finger exercises" at opportune points within the chapters. Some are quite short, and are intended to allow readers to confirm that they understood the material they just read. Some are a bit more challenging, and are suitable for exam questions. And others are challenging enough to be useful as homework assignments.

The book has three pervasive themes: systematic problem solving, the power of abstraction, and computation as a way of thinking about the world. When you have finished this book you should have:

- Learned a language, Python, for expressing computations,
- Learned a systematic approach to organizing, writing, and debugging medium-sized programs,
- Developed an informal understanding of computational complexity,
- Developed some insight into the process of moving from an ambiguous problem statement to a computational formulation of a method for solving the problem,
- Learned a useful set of algorithmic and problem reduction techniques,
- Learned how to use randomness and simulations to shed light on problems that don't easily succumb to closed-form solutions, and
- Learned how to use computational tools (including simple statistical, visualization, and machine learning tools) to model and understand data.

Programming is an intrinsically difficult activity. Just as "there is no royal road to geometry,"[1] there is no royal road to programming. If you really want to learn the material, reading the book will not be enough. At the very least you should try running some of the code in the book. Various versions of the courses from which this book has been derived have been available on MIT's OpenCourseWare (OCW) Web site since 2008. The site includes video recordings of lectures and a complete set of problem sets and exams. Since the fall of 2012, edX and MITx have offered online courses that cover much of the material in this book. We strongly recommend that you do the problem sets associated with one of the OCW or edX offerings.

[1] This was Euclid's purported response, circa 300 BCE, to King Ptolemy's request for an easier way to learn mathematics.

ACKNOWLEDGMENTS

The first edition of this book grew out of a set of lecture notes that I prepared while teaching an undergraduate course at MIT. The course, and therefore this book, benefited from suggestions from faculty colleagues (especially Ana Bell, Eric Grimson, Srinivas Devadas, Fredo Durand, Ron Rivest, and Chris Terman), teaching assistants, and the students who took the course. David Guttag overcame his aversion to computer science, and proofread multiple chapters.

Like all successful professors, I owe a great deal to my graduate students. The photo on the back cover of this book depicts me supporting some of my current students. In the lab, however, it is they who support me. In addition to doing great research (and letting me take some of the credit for it), Guha Balakrishnan, David Blalock, Joel Brooks, Ganeshapillai Gartheeban, Jen Gong, Yun Liu, Anima Singh, Jenna Wiens, and Amy Zhao all provided useful comments on various versions of this manuscript.

I owe a special debt of gratitude to Julie Sussman, P.P.A. Until I started working with Julie, I had no idea how much difference an editor could make. I had worked with capable copy editors on previous books, and thought that was what I needed for this book. I was wrong. I needed a collaborator who could read the book with the eyes of a student, and tell me what needed to be done, what should be done, and what could be done if I had the time and energy to do it. Julie buried me in “suggestions” that were too good to ignore. Her combined command of both the English language and programming is quite remarkable.

Finally, thanks to my wife, Olga, for pushing me to finish and for excusing me from various household duties so that I could work on the book.

1 GETTING STARTED

A computer does two things, and two things only: it performs calculations and it remembers the results of those calculations. But it does those two things extremely well. The typical computer that sits on a desk or in a briefcase performs a billion or so calculations a second. It's hard to image how truly fast that is. Think about holding a ball a meter above the floor, and letting it go. By the time it reaches the floor, your computer could have executed over a billion instructions. As for memory, a small computer might have hundreds of gigabytes of storage. How big is that? If a byte (the number of bits, typically eight, required to represent one character) weighed one gram (which it doesn't), 100 gigabytes would weigh 10,000 metric tons. For comparison, that's roughly the combined weight of 15,000 African elephants.

For most of human history, computation was limited by the speed of calculation of the human brain and the ability to record computational results with the human hand. This meant that only the smallest problems could be attacked computationally. Even with the speed of modern computers, there are still problems that are beyond modern computational models (e.g., understanding climate change), but more and more problems are proving amenable to computational solution. It is our hope that by the time you finish this book, you will feel comfortable bringing computational thinking to bear on solving many of the problems you encounter during your studies, work, and even everyday life.

What do we mean by computational thinking?

All knowledge can be thought of as either declarative or imperative. **Declarative knowledge** is composed of statements of fact. For example, "the square root of x is a number y such that y*y = x." This is a statement of fact. Unfortunately, it doesn't tell us anything about how to find a square root.

Imperative knowledge is "how to" knowledge, or recipes for deducing information. Heron of Alexandria was the first to document a way to compute the square root of a number.[2] His method for finding the square root of a number, call it x, can be summarized as:

[2] Many believe that Heron was not the inventor of this method, and indeed there is some evidence that it was well known to the ancient Babylonians.

1. Start with a guess, g.
2. If g*g is close enough to x, stop and say that g is the answer.
3. Otherwise create a new guess by averaging g and x/g, i.e., (g + x/g)/2.
4. Using this new guess, which we again call g, repeat the process until g*g is close enough to x.

Consider, for example, finding the square root of 25.

1. Set g to some arbitrary value, e.g., 3.
2. We decide that 3*3 = 9 is not close enough to 25.
3. Set g to (3 + 25/3)/2 = 5.67.[3]
4. We decide that 5.67*5.67 = 32.15 is still not close enough to 25.
5. Set g to (5.67 + 25/5.67)/2 = 5.04
6. We decide that 5.04*5.04 = 25.4 is close enough, so we stop and declare 5.04 to be an adequate approximation to the square root of 25.

Note that the description of the method is a sequence of simple steps, together with a flow of control that specifies when each step is to be executed. Such a description is called an **algorithm**.[4] This algorithm is an example of a guess-and-check algorithm. It is based on the fact that it is easy to check whether or not a guess is a good one.

A bit more formally, an algorithm is a finite list of instructions that describe a **computation** that when executed on a set of inputs will proceed through a set of well-defined states and eventually produce an output.

An algorithm is a bit like a recipe from a cookbook:

1. Put custard mixture over heat.
2. Stir.
3. Dip spoon in custard.
4. Remove spoon and run finger across back of spoon.
5. If clear path is left, remove custard from heat and let cool.
6. Otherwise repeat.

It includes some tests for deciding when the process is complete, as well as instructions about the order in which to execute instructions, sometimes jumping to a specific instruction based on a test.

So how does one capture this idea of a recipe in a mechanical process? One way would be to design a machine specifically intended to compute square roots.

[3] For simplicity, we are rounding results.

[4] The word "algorithm" is derived from the name of the Persian mathematician Muhammad ibn Musa al-Khwarizmi.

Odd as this may sound, the earliest computing machines were, in fact, **fixed-program computers**, meaning they were designed to do very specific things, and were mostly tools to solve a specific mathematical problem, e.g., to compute the trajectory of an artillery shell. One of the first computers (built in 1941 by Atanasoff and Berry) solved systems of linear equations, but could do nothing else. Alan Turing's bombe machine, developed during World War II, was designed strictly for the purpose of breaking German Enigma codes. Some very simple computers still use this approach. For example, a simple handheld calculator[5] is a fixed-program computer. It can do basic arithmetic, but it cannot be used as a word processor or to run video games. To change the program of such a machine, one has to replace the circuitry.

The first truly modern computer was the Manchester Mark 1.[6] It was distinguished from its predecessors by the fact that it was a **stored-program computer**. Such a computer stores (and manipulates) a sequence of instructions, and has components that will execute any instruction in that sequence. By creating an instruction-set architecture and detailing the computation as a sequence of instructions (i.e., a program), we get a highly flexible machine. By treating those instructions in the same way as data, a stored-program machine can easily change the program, and can do so under program control. Indeed, the heart of the computer then becomes a program (called an **interpreter**) that can execute any legal set of instructions, and thus can be used to compute anything that one can describe using some basic set of instructions.

Both the program and the data it manipulates reside in memory. Typically, there is a program counter that points to a particular location in memory, and computation starts by executing the instruction at that point. Most often, the interpreter simply goes to the next instruction in the sequence, but not always. In some cases, it performs a test, and on the basis of that test, execution may jump to some other point in the sequence of instructions. This is called **flow of control,** and is essential to allowing us to write programs that perform complex tasks.

Returning to the recipe metaphor, given a fixed set of ingredients a good chef can make an unbounded number of tasty dishes by combining them in different ways. Similarly, given a small fixed set of primitive features a good programmer can produce an unbounded number of useful programs. This is what makes programming such an amazing endeavor.

[5] It's hard to believe, but once upon a time phones did not provide computational facilities. People actually used small devices that could be used only for calculation.

[6] This computer was built at the University of Manchester, and ran its first program in 1949. It implemented ideas previously described by John von Neumann and was anticipated by the theoretical concept of the Universal Turing Machine described by Alan Turing in 1936.

To create recipes, or sequences of instructions, we need a **programming language** in which to describe them, a way to give the computer its marching orders.

In 1936, the British mathematician Alan Turing described a hypothetical computing device that has come to be called a **Universal Turing Machine**. The machine had an unbounded memory in the form of "tape" on which one could write zeroes and ones, and some very simple primitive instructions for moving, reading, and writing to the tape. The **Church-Turing thesis** states that if a function is computable, a Turing Machine can be programmed to compute it.

The "if" in the Church-Turing thesis is important. Not all problems have computational solutions. Turing showed, for example, that it is impossible to write a program that given an arbitrary program, call it P, prints `true` if and only if P will run forever. This is known as the **halting problem**.

The Church-Turing thesis leads directly to the notion of **Turing completeness**. A programming language is said to be Turing complete if it can be used to simulate a universal Turing Machine. All modern programming languages are Turing complete. As a consequence, anything that can be programmed in one programming language (e.g., Python) can be programmed in any other programming language (e.g., Java). Of course, some things may be easier to program in a particular language, but all languages are fundamentally equal with respect to computational power.

Fortunately, no programmer has to build programs out of Turing's primitive instructions. Instead, modern programming languages offer a larger, more convenient set of primitives. However, the fundamental idea of programming as the process of assembling a sequence of operations remains central.

Whatever set of primitives one has, and whatever methods one has for using them, the best thing and the worst thing about programming are the same: the computer will do exactly what you tell it to do. This is a good thing because it means that you can make it do all sorts of fun and useful things. It is a bad thing because when it doesn't do what you want it to do, you usually have nobody to blame but yourself.

There are hundreds of programming languages in the world. There is no best language (though one could nominate some candidates for worst). Different languages are better or worse for different kinds of applications. MATLAB, for example, is a good language for manipulating vectors and matrices. C is a good language for writing programs that control data networks. PHP is a good language for building Web sites. And Python is a good general-purpose language.

Each programming language has a set of primitive constructs, a syntax, a static semantics, and a semantics. By analogy with a natural language, e.g., English, the primitive constructs are words, the syntax describes which strings of words constitute well-formed sentences, the static semantics defines which sentences are meaningful, and the semantics defines the meaning of those sentences. The primitive constructs in Python include **literals** (e.g., the number `3.2` and the string `'abc'`) and **infix operators** (e.g., `+` and `/`).

The **syntax** of a language defines which strings of characters and symbols are well formed. For example, in English the string "Cat dog boy." is not a syntactically valid sentence, because the syntax of English does not accept sentences of the form <noun> <noun> <noun>. In Python, the sequence of primitives `3.2 + 3.2` is syntactically well formed, but the sequence `3.2 3.2` is not.

The **static semantics** defines which syntactically valid strings have a meaning. In English, for example, the string "I runs fast," is of the form <pronoun> < verb> <adverb>, which is a syntactically acceptable sequence. Nevertheless, it is not valid English, because the noun "I" is singular and the verb "runs" is plural. This is an example of a static semantic error. In Python, the sequence `3.2/'abc'` is syntactically well formed (<literal> <operator> <literal>), but produces a static semantic error since it is not meaningful to divide a number by a string of characters.

The **semantics** of a language associates a meaning with each syntactically correct string of symbols that has no static semantic errors. In natural languages, the semantics of a sentence can be ambiguous. For example, the sentence "I cannot praise this student too highly," can be either flattering or damning. Programming languages are designed so that each legal program has exactly one meaning.

Though syntax errors are the most common kind of error (especially for those learning a new programming language), they are the least dangerous kind of error. Every serious programming language does a complete job of detecting syntactic errors, and will not allow users to execute a program with even one syntactic error. Furthermore, in most cases the language system gives a sufficiently clear indication of the location of the error that it is obvious what needs to be done to fix it.

The situation with respect to static semantic errors is a bit more complex. Some programming languages, e.g., Java, do a lot of static semantic checking before allowing a program to be executed. Others, e.g., C and Python (alas), do relatively less static semantic checking before a program is executed. Python does do a considerable amount of semantic checking while running a program.

One doesn't usually speak of a program as having a semantic error. If a program has no syntactic errors and no static semantic errors, it has a meaning, i.e., it has semantics. Of course, that isn't to say that it has the semantics that its creator intended it to have. When a program means something other than what its creator thinks it means, bad things can happen.

What might happen if the program has an error, and behaves in an unintended way?

- It might crash, i.e., stop running and produce some sort of obvious indication that it has done so. In a properly designed computing system, when a program crashes it does not do damage to the overall system. Of course, some very popular computer systems don't have this nice property. Almost everyone who uses a personal computer has run a program that has managed to make it necessary to restart the whole computer.
- Or it might keep running, and running, and running, and never stop. If one has no idea of approximately how long the program is supposed to take to do its job, this situation can be hard to recognize.
- Or it might run to completion and produce an answer that might, or might not, be correct.

Each of these is bad, but the last of them is certainly the worst. When a program appears to be doing the right thing but isn't, bad things can follow: fortunes can be lost, patients can receive fatal doses of radiation therapy, airplanes can crash.

Whenever possible, programs should be written in such a way that when they don't work properly, it is self-evident. We will discuss how to do this throughout the book.

Finger exercise: Computers can be annoyingly literal. If you don't tell them exactly what you want them to do, they are likely to do the wrong thing. Try writing an algorithm for driving between two destinations. Write it the way you would for a person, and then imagine what would happen if that person were as stupid as a computer, and executed the algorithm exactly as written. How many traffic tickets might that person get?

2 INTRODUCTION TO PYTHON

Though each programming language is different (though not as different as their designers would have us believe), there are some dimensions along which they can be related.

- **Low-level versus high-level** refers to whether we program using instructions and data objects at the level of the machine (e.g., move 64 bits of data from this location to that location) or whether we program using more abstract operations (e.g., pop up a menu on the screen) that have been provided by the language designer.
- **General versus targeted to an application domain** refers to whether the primitive operations of the programming language are widely applicable or are fine-tuned to a domain. For example, SQL is designed to facilitate extracting information from relational databases, but you wouldn't want to use it build an operating system.
- **Interpreted versus compiled** refers to whether the sequence of instructions written by the programmer, called **source code**, is executed directly (by an interpreter) or whether it is first converted (by a compiler) into a sequence of machine-level primitive operations. (In the early days of computers, people had to write source code in a language that was very close to the **machine code** that could be directly interpreted by the computer hardware.) There are advantages to both approaches. It is often easier to debug programs written in languages that are designed to be interpreted, because the interpreter can produce error messages that are easy to correlate with the source code. Compiled languages usually produce programs that run more quickly and use less space.

In this book, we use **Python**. However, this book is not about Python. It will certainly help readers learn Python, and that's a good thing. What is much more important, however, is that careful readers will learn something about how to write programs that solve problems. This skill can be transferred to any programming language.

Python is a general-purpose programming language that can be used effectively to build almost any kind of program that does not need direct access to the computer's hardware. Python is not optimal for programs that have high reliability constraints (because of its weak static semantic checking) or that are built and maintained by many people or over a long period of time (again because of the weak static semantic checking).

However, Python does have several advantages over many other languages. It is a relatively simple language that is easy to learn. Because Python is designed to be interpreted, it can provide the kind of runtime feedback that is especially helpful to novice programmers. There are also a large number of freely available libraries that interface to Python and provide useful extended functionality. Several of those are used in this book.

Now we are ready to start learning some of the basic elements of Python. These are common to almost all programming languages in concept, though not necessarily in detail.

The reader should be forewarned that this book is by no means a comprehensive introduction to Python. We use Python as a vehicle to present concepts related to computational problem solving and thinking. The language is presented in dribs and drabs, as needed for this ulterior purpose. Python features that we don't need for that purpose are not presented at all. We feel comfortable about not covering the entire language because there are excellent online resources describing almost every aspect of the language. When we teach the course on which this book is based, we suggest to the students that they rely on these free online resources for Python reference material.

Python is a living language. Since its introduction by Guido von Rossum in 1990, it has undergone many changes. For the first decade of its life, Python was a little known and little used language. That changed with the arrival of Python 2.0 in 2000. In addition to incorporating a number of important improvements to the language itself, it marked a shift in the evolutionary path of the language. A large number of people began developing libraries that interfaced seamlessly with Python, and continuing support and development of the Python ecosystem became a community-based activity. Python 3.0 was released at the end of 2008. This version of Python cleaned up many of the inconsistencies in the design of the various releases of Python 2 (often referred to as Python 2.x). However, it was not backward compatible. That meant that most programs written for earlier versions of Python could not be run using implementations of Python 3.

Over the last few years, most of the important public domain Python libraries have been ported to Python 3 and thoroughly tested using Python 3.5—the version of Python we use in this book.

2.1 The Basic Elements of Python

A Python **program**, sometimes called a **script**, is a sequence of definitions and commands. These definitions are evaluated and the commands are executed by the Python interpreter in something called the **shell**. Typically, a new shell is created whenever execution of a program begins. Usually a window is associated with the shell.

We recommend that you start a Python shell now, and use it to try the examples contained in the remainder of this chapter. And, for that matter, later in the book as well.

A **command**, often called a **statement**, instructs the interpreter to do something. For example, the statement `print('Yankees rule!')` instructs the interpreter to call the function[7] `print`, which will output the string `Yankees rule!` to the window associated with the shell.

The sequence of commands

```
print('Yankees rule!')
print('But not in Boston!')
print('Yankees rule,', 'but not in Boston!')
```

causes the interpreter to produce the output

```
Yankees rule!
But not in Boston!
Yankees rule, but not in Boston!
```

Notice that two values were passed to `print` in the third statement. The `print` function takes a variable number of arguments separated by commas, and prints them, separated by a space character, in the order in which they appear.[8]

2.1.1 Objects, Expressions, and Numerical Types

Objects are the core things that Python programs manipulate. Every object has a **type** that defines the kinds of things that programs can do with that object.

Types are either scalar or non-scalar. **Scalar** objects are indivisible. Think of them as the atoms of the language.[9] **Non-scalar** objects, for example strings, have internal structure.

[7] Functions are discussed in Section 4.1.

[8] In Python 2, `print` is a command rather than a function. One would therefore write the line `print 'Yankees rule!', 'but not in Boston'`.

[9] Yes, atoms are not truly indivisible. However, splitting them is not easy, and doing so can have consequences that are not always desirable.

Many types of objects can be denoted by **literals** in the text of a program. For example, the text `2` is a literal representing a number and the text `'abc'` a literal representing a string.

Python has four types of scalar objects:

- `int` is used to represent integers. Literals of type `int` are written in the way we typically denote integers (e.g., `-3` or `5` or `10002`).
- `float` is used to represent real numbers. Literals of type `float` always include a decimal point (e.g., `3.0` or `3.17` or `-28.72`). (It is also possible to write literals of type `float` using scientific notation. For example, the literal `1.6E3` stands for $1.6*10^3$, i.e., it is the same as 1600.0.) You might wonder why this type is not called `real`. Within the computer, values of type `float` are stored in the computer as **floating point numbers**. This representation, which is used by all modern programming languages, has many advantages. However, under some situations it causes floating point arithmetic to behave in ways that are slightly different from arithmetic on real numbers. We discuss this in Section 3.4.
- `bool` is used to represent the Boolean values `True` and `False`.
- `None` is a type with a single value. We will say more about this in Section 4.1.

Objects and **operators** can be combined to form **expressions**, each of which evaluates to an object of some type. We will refer to this as the **value** of the expression. For example, the expression `3 + 2` denotes the object `5` of type `int`, and the expression `3.0 + 2.0` denotes the object `5.0` of type `float`.

The == operator is used to test whether two expressions evaluate to the same value, and the != operator is used to test whether two expressions evaluate to different values. A single = means something quite different, as we will see in Section 2.1.2. Be forewarned, you will make the mistake of typing "=" when you meant to type "==". Keep an eye out for this error.

The symbol >>> is a **shell prompt** indicating that the interpreter is expecting the user to type some Python code into the shell. The line below the line with the prompt is produced when the interpreter evaluates the Python code entered at the prompt, as illustrated by the following interaction with the interpreter:

```
>>> 3 + 2
5
>>> 3.0 + 2.0
5.0
>>> 3 != 2
True
```

The built-in Python function `type` can be used to find out the type of an object:

```
>>> type(3)
<type 'int'>
>>> type(3.0)
<type 'float'>
```

Operators on objects of type `int` and `float` are listed in Figure 2.1.

`i+j` is the sum of `i` and `j`. If `i` and `j` are both of type `int`, the result is an `int`. If either of them is a `float`, the result is a `float`.

`i–j` is `i` minus `j`. If `i` and `j` are both of type `int`, the result is an `int`. If either of them is a `float`, the result is a `float`.

`i*j` is the product of `i` and `j`. If `i` and `j` are both of type `int`, the result is an `int`. If either of them is a `float`, the result is a `float`.

`i//j` is integer division. For example, the value of `6//2` is the `int 3` and the value of `6//4` is the `int 1`. The value is `1` because integer division returns the quotient and ignores the remainder. If `j == 0`, an error occurs.

`i/j` is `i` divided by `j`. In Python 3, the / operator, performs floating point division. For example, the value of `6/4` is `1.5`. If `j == 0`, an error occurs. (In Python 2, when `i` and `j` are both of type `int`, the / operator behaves the same way as // and returns an `int`. If either i or j is a `float`, it behaves like the Python 3 / operator.)

`i%j` is the remainder when the `int i` is divided by the `int j`. It is typically pronounced "i mod j," which is short for "i modulo j."

`i**j` is `i` raised to the power `j`. If `i` and `j` are both of type `int`, the result is an `int`. If either of them is a `float`, the result is a `float`.

The comparison operators are == (equal), != (not equal), **>** (greater), **>=** (at least), **<**, (less) and **<=** (at most).

Figure 2.1 Operators on types `int` and `float`

The arithmetic operators have the usual precedence. For example, `*` binds more tightly than `+`, so the expression `x+y*2` is evaluated by first multiplying `y` by `2` and then adding the result to `x`. The order of evaluation can be changed by using parentheses to group subexpressions, e.g., `(x+y)*2` first adds `x` and `y`, and then multiplies the result by `2`.

The primitive operators on type `bool` are `and`, `or`, and `not`:

- **`a and b`** is `True` if both `a` and `b` are `True`, and `False` otherwise.
- **`a or b`** is `True` if at least one of `a` or `b` is `True`, and `False` otherwise.
- **`not a`** is `True` if `a` is `False`, and `False` if `a` is `True`.

2.1.2 Variables and Assignment

Variables provide a way to associate names with objects. Consider the code

```
pi = 3
radius = 11
area = pi * (radius**2)
radius = 14
```

It first **binds** the names `pi` and `radius` to different objects of type `int`.[10] It then binds the name `area` to a third object of type `int`. This is depicted in the left panel of Figure 2.2.

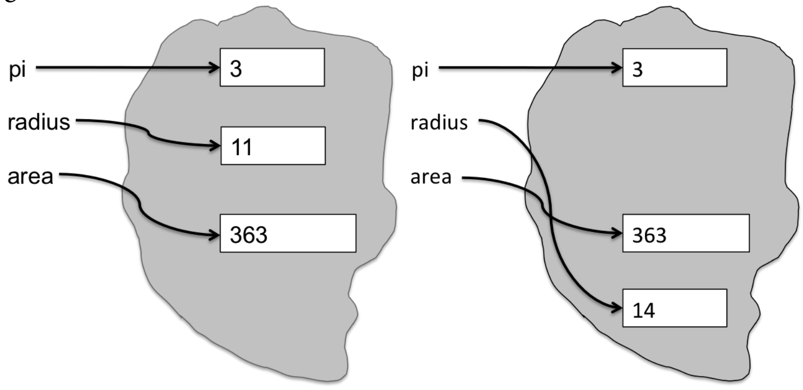


Figure 2.2 Binding of variables to objects

If the program then executes `radius = 14`, the name `radius` is rebound to a different object of type `int`, as shown in the right panel of Figure 2.2. Note that this assignment has no effect on the value to which `area` is bound. It is still bound to the object denoted by the expression `3*(11**2)`.

In Python, **a variable is just a name,** nothing more. Remember this—it is important. An **assignment** statement associates the name to the left of the `=` symbol with the object denoted by the expression to the right of the `=`. Remember this too. An object can have one, more than one, or no name associated with it.

[10] If you believe that the actual value of π is not 3, you're right. We even demonstrate that fact in Section 16.4.

Perhaps we shouldn't have said, "a variable is just a name." Despite what Juliet said,[11] names matter. Programming languages let us describe computations in a way that allows machines to execute them. This does not mean that only computers read programs.

As you will soon discover, it's not always easy to write programs that work correctly. Experienced programmers will confirm that they spend a great deal of time reading programs in an attempt to understand why they behave as they do. It is therefore of critical importance to write programs in such way that they are easy to read. Apt choice of variable names plays an important role in enhancing readability.

Consider the two code fragments

```
a = 3.14159               pi = 3.14159
b = 11.2                  diameter = 11.2
c = a*(b**2)              area = pi*(diameter**2)
```

As far as Python is concerned, they are not different. When executed, they will do the same thing. To a human reader, however, they are quite different. When we read the fragment on the left, there is no *a priori* reason to suspect that anything is amiss. However, a quick glance at the code on the right should prompt us to be suspicious that something is wrong. Either the variable should have been named `radius` rather than `diameter`, or `diameter` should have been divided by `2.0` in the calculation of the area.

In Python, variable names can contain uppercase and lowercase letters, digits (but they cannot start with a digit), and the special character `_`. Python variable names are case-sensitive e.g., `Julie` and `julie` are different names. Finally, there are a small number of **reserved words** (sometimes called **keywords**) in Python that have built-in meanings and cannot be used as variable names. Different versions of Python have slightly different lists of reserved words. The reserved words in Python 3 are `and`, `as`, `assert`, `break`, `class`, `continue`, `def`, `del`, `elif`, `else`, `except`, `False`, `finally`, `for`, `from`, `global`, `if`, `import`, `in`, `is`, `lambda`, `nonlocal`, `None`, `not`, `or`, `pass`, `raise`, `return`, `True`, `try`, `while`, `with`, and `yield`.

Another good way to enhance the readability of code is to add **comments**. Text following the symbol `#` is not interpreted by Python. For example, one might write

[11] "What's in a name? That which we call a rose by any other name would smell as sweet."

```
side = 1 #length of sides of a unit square
radius = 1 #radius of a unit circle
#subtract area of unit circle from area of unit square
areaC = pi*radius**2
areaS = side*side
difference = areaS - areaC
```

Python allows multiple assignment. The statement

```
x, y = 2, 3
```

binds `x` to `2` and `y` to `3`. All of the expressions on the right-hand side of the assignment are evaluated before any bindings are changed. This is convenient since it allows you to use multiple assignment to swap the bindings of two variables.

For example, the code

```
x, y = 2, 3
x, y = y, x
print('x =', x)
print('y =', y)
```

will print

```
x = 3
y = 2
```

2.1.3 Python IDE's

Typing programs directly into the shell is highly inconvenient. Most programmers prefer to use some sort of text editor that is part of an **integrated development environment** (**IDE**).

One IDE, **IDLE**,[12] comes as part of the standard Python installation package. As Python has grown in popularity, other IDE's have sprung up. These newer IDE's often incorporate some of the more popular Python libraries and provide facilities not provided by IDLE. **Anaconda** and **Canopy** are among the more popular of these IDE's. The code appearing in this book was created and tested using Anaconda.

IDE's are applications, just like any other application on your computer. Start one the same way you would start any other application, e.g., by double-clicking on an icon.

[12] Allegedly, the name Python was chosen as a tribute to the British comedy troupe Monty Python. This leads one to think that the name IDLE is a pun on Eric Idle, a member of the troupe.

All of the Python IDE's provide

- A text editor with syntax highlighting, auto completion, and smart indentation,
- a shell with syntax highlighting, and
- an integrated debugger, which you can safely ignore for now.

When the IDE starts it will open a shell window into which you can type Python commands. It will also provide you with a file menu and an edit menu (as well as some other menus that make it convenient to do things such as printing your program).

The **file menu** includes commands to

- create a new editing window into which you can type a Python program,
- open a file containing an existing Python program, and
- save the contents of the current editing window into a file (with the file extension `.py`).

The **edit menu** includes standard text-editing commands (e.g., copy, paste, and find) plus some commands specifically designed to make it easy to edit Python code (e.g., indent region and comment out region).

For more information about some popular IDE's see

```
http://docs.python.org/library/idle.html/
https://store.continuum.io/cshop/anaconda/
https://www.enthought.com/products/canopy/
```

2.2 Branching Programs

The kinds of computations we have been looking at thus far are called **straight-line programs**. They execute one statement after another in the order in which they appear, and stop when they run out of statements. The kinds of computations we can describe with straight-line programs are not very interesting. In fact, they are downright boring.

Branching programs are more interesting. The simplest branching statement is a **conditional**. As shown in the boxed-in part of Figure 2.3, a conditional statement has three parts:

- a test, i.e., an expression that evaluates to either `True` or `False`;
- a block of code that is executed if the test evaluates to `True`; and
- an optional block of code that is executed if the test evaluates to `False`.

After a conditional statement is executed, execution resumes at the code following the statement.

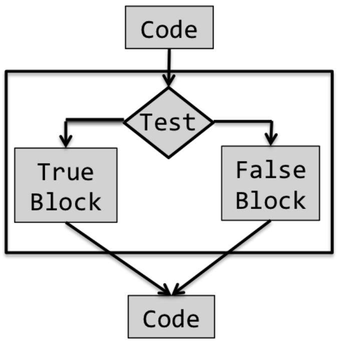


Figure 2.3 Flow chart for conditional statement

In Python, a conditional statement has the form

```
if Boolean expression:
    block of code
else:
    block of code
```

or

```
if Boolean expression:
    block of code
```

In describing the form of Python statements we use italics to describe the kinds of code that could occur at that point in a program. For example, *`Boolean expression`* indicates that any expression that evaluates to `True` or `False` can follow the reserved word `if`, and *`block of code`* indicates that any sequence of Python statements can follow `else:`.

Consider the following program that prints "Even" if the value of the variable `x` is even and "Odd" otherwise:

```
if x%2 == 0:
    print('Even')
else:
    print('Odd')
print('Done with conditional')
```

The expression `x%2 == 0` evaluates to `True` when the remainder of `x` divided by `2` is `0`, and `False` otherwise. Remember that `==` is used for comparison, since `=` is reserved for assignment.

Indentation is semantically meaningful in Python. For example, if the last statement in the above code were indented it would be part of the block of code associated with the `else`, rather than the block of code following the conditional statement.

Python is unusual in using indentation this way. Most other programming languages use some sort of bracketing symbols to delineate blocks of code, e.g., C encloses blocks in braces, { }. An advantage of the Python approach is that it ensures that the visual structure of a program is an accurate representation of the semantic structure of that program. Because indentation is semantically important, the notion of a line is important.

When either the true block or the false block of a conditional contains another conditional, the conditional statements are said to be **nested**. In the code below, there are nested conditionals in both branches of the top-level `if` statement.

```
if x%2 == 0:
    if x%3 == 0:
        print('Divisible by 2 and 3')
    else:
        print('Divisible by 2 and not by 3')
elif x%3 == 0:
    print('Divisible by 3 and not by 2')
```

The `elif` in the above code stands for "else if."

It is often convenient to use a **compound Boolean expression** in the test of a conditional, for example,

```
if x < y and x < z:
    print('x is least')
elif y < z:
    print('y is least')
else:
    print('z is least')
```

Conditionals allow us to write programs that are more interesting than straight-line programs, but the class of branching programs is still quite limited. One way to think about the power of a class of programs is in terms of how long they can take to run. Assume that each line of code takes one unit of time to execute. If a straight-line program has n lines of code, it will take n units of time to run. What about a branching program with n lines of code? It might take less than n units of time to run, but it cannot take more, since each line of code is executed at most once.

A program for which the maximum running time is bounded by the length of the program is said to run in **constant time**. This does not mean that each time it is run it executes the same number of steps. It means that there exists a constant, k, such that the program is guaranteed to take no more than k steps to run.

This implies that the running time does not grow with the size of the input to the program.

Constant-time programs are quite limited in what they can do. Consider, for example, writing a program to tally the votes in an election. It would be truly surprising if one could write a program that could do this in a time that was independent of the number of votes cast. In fact, one can prove that it is impossible to do so. The study of the intrinsic difficulty of problems is the topic of **computational complexity**. We will return to this topic several times in this book.

Fortunately, we need only one more programming language construct, iteration, to be able to write programs of arbitrary complexity. We get to that in Section 2.4.

Finger exercise: Write a program that examines three variables—`x`, `y`, and `z`—and prints the largest odd number among them. If none of them are odd, it should print a message to that effect.

2.3 Strings and Input

Objects of type `str` are used to represent strings of characters.[13] Literals of type `str` can be written using either single or double quotes, e.g., `'abc'` or `"abc"`. The literal `'123'` denotes a string of three characters, not the number one hundred twenty-three.

Try typing the following expressions in to the Python interpreter (remember that the >>> is a prompt, not something that you type):

```
>>> 'a'
>>> 3*4
>>> 3*'a'
>>> 3+4
>>> 'a'+'a'
```

The operator `+` is said to be **overloaded**: It has different meanings depending upon the types of the objects to which it is applied. For example, it means addition when applied to two numbers and concatenation when applied to two strings. The operator `*` is also overloaded. It means what you expect it to mean when its operands are both numbers. When applied to an `int` and a `str`, it a **repetition operator**—the expression `n*s`, where `n` is an `int` and `s` is a `str`, evaluates to a

[13] Unlike many programming languages, Python has no type corresponding to a character. Instead, it uses strings of length 1.

`str` with `n` repeats of `s`. For example, the expression `2*'John'` has the value `'JohnJohn'`. There is a logic to this. Just as the mathematical expression 3*2 is equivalent to 2+2+2, the expression `3*'a'` is equivalent to `'a'+'a'+'a'`.

Now try typing

```
>>> a
>>> 'a'*'a'
```

Each of these lines generates an error message. The first line produces the message

```
NameError: name 'a' is not defined
```

Because `a` is not a literal of any type, the interpreter treats it as a name. However, since that name is not bound to any object, attempting to use it causes a runtime error. The code `'a'*'a'` produces the error message

```
TypeError: can't multiply sequence by non-int of type 'str'
```

That **type checking** exists is a good thing. It turns careless (and sometimes subtle) mistakes into errors that stop execution, rather than errors that lead programs to behave in mysterious ways. The type checking in Python is not as strong as in some other programming languages (e.g., Java), but it is better in Python 3 than in Python 2. For example, it is pretty clear what `<` should mean when it is used to compare two strings or two numbers. But what should the value of `'4' < 3` be? Rather arbitrarily, the designers of Python 2 decided that it should be `False`, because all numeric values should be less than all values of type `str`. The designers of Python 3 and most other modern languages decided that since such expressions don't have an obvious meaning, they should generate an error message.

Strings are one of several sequence types in Python. They share the following operations with all sequence types.

- The **length** of a string can be found using the `len` function. For example, the value of `len('abc')` is `3`.
- **Indexing** can be used to extract individual characters from a string. In Python, all indexing is zero-based. For example, typing `'abc'[0]` into the interpreter will cause it to display the string `'a'`. Typing `'abc'[3]` will produce the error message `IndexError: string index out of range`. Since Python uses `0` to indicate the first element of a string, the last element of a string of length 3 is accessed using the index `2`. Negative numbers are used to index from the end of a string. For example, the value of `'abc'[-1]` is `'c'`.
- **Slicing** is used to extract substrings of arbitrary length. If `s` is a string, the expression `s[start:end]` denotes the substring of `s` that starts at index `start` and

ends at index end-1. For example, 'abc'[1:3] = 'bc'. Why does it end at index end-1 rather than end? So that expressions such as 'abc'[0:len('abc')] have the value one might expect. If the value before the colon is omitted, it defaults to 0. If the value after the colon is omitted, it defaults to the length of the string. Consequently, the expression 'abc'[:] is semantically equivalent to the more verbose 'abc'[0:len('abc')].

2.3.1 Input

Python 3 has a function, input, that can be used to get input directly from a user.[14] It takes a string as an argument and displays it as a prompt in the shell. It then waits for the user to type something, followed by hitting the enter key. The line typed by the user is treated as a string and becomes the value returned by the function.

Consider the code

```
>>> name = input('Enter your name: ')
Enter your name: George Washington
>>> print('Are you really', name, '?')
Are you really George Washington ?
>>> print('Are you really ' + name + '?')
Are you really George Washington?
```

Notice that the first print statement introduces a blank before the "?". It does this because when print is given multiple arguments it places a blank space between the values associated with the arguments. The second print statement uses concatenation to produce a string that does not contain the superfluous blank and passes this as the only argument to print.

Now consider

```
>>> n = input('Enter an int: ')
Enter an int: 3
>>> print(type(n))
<type 'str'>
```

Notice that the variable n is bound to the str '3' not the int 3. So, for example, the value of the expression n*4 is '3333' rather than 12. The good news is that whenever a string is a valid literal of some type, a type conversion can be applied to it.

[14] Python 2 has two functions, input and raw_input, that are used to get input from users. Somewhat confusingly, raw_input in Python 2 has the same semantics as input in Python 3. The Python 2 function input treats the typed line as a Python expression and infers a type. Python 2 programmers would be well advised to use only raw_input.

Type conversions (also called **type casts**) are used often in Python code. We use the name of a type to convert values to that type. So, for example, the value of `int('3')*4` is `12`. When a `float` is converted to an `int`, the number is truncated (not rounded), e.g., the value of `int(3.9)` is the `int` `3`.

2.3.2 A Digression About Character Encoding

For many years most programming languages used a standard called ASCII for the internal representation of characters. This standard included 128 characters, plenty for representing the usual set of characters appearing in English-language text—but not enough to cover the characters and accents appearing in all the world's languages.

In recent years, there has been a shift to Unicode. The Unicode standard is a character coding system designed to support the digital processing and display of the written texts of all languages. The standard contains more than 120,000 different characters—covering 129 modern and historic scripts and multiple symbol sets. The Unicode standard can be implemented using different internal character encodings. You can tell Python which encoding to use by inserting a comment of the form

```
# -*- coding: encoding name -*-
```

as the first or second line of your program. For example,

```
# -*- coding: utf-8 -*-
```

instructs Python to use UTF-8, the most frequently used character encoding for World Wide Web pages.[15] If you don't have such a comment in your program, most Python implementations will default to UTF-8.

When using UTF-8, you can, text editor permitting, directly enter code like

```
print('Mluvíš anglicky?')
print('क्या आप अंग्रेज़ी बोलते हैं?')
```

which will print

```
Mluvíš anglicky?
क्या आप अंग्रेज़ी बोलते हैं?
```

You might be wondering how I managed to type the string `'क्या आप अंग्रेज़ी बोलते हैं?'`. I didn't. Because most the World Wide Web uses UTF-8, I was able to cut the string from a Web page and paste it directly into my program.

[15] In 2016, over 85% of the pages on the World Wide Web were encoded using UTF-8.

2.4 Iteration

We closed Section 2.2 with the observation that most computational tasks cannot be accomplished using branching programs. Consider, for example, writing a program that asks the user how many time he wants to print the letter X, and then prints a string with that number of X's. We might think about writing something like

```
numXs = int(input('How many times should I print the letter X? '))
toPrint = ''
if numXs == 1:
    toPrint = 'X'
elif numXs == 2:
    toPrint = 'XX'
elif numXs == 3:
    toPrint = 'XXX'
#...
print(toPrint)
```

But it would quickly become apparent that we would need as many conditionals as there are positive integers—and there are an infinite number of those. What we need is a program that looks like

```
numXs = int(input('How many times should I print the letter X? '))
toPrint = ''
concatenate X to toPrint numXs times
print(toPrint)
```

When we want a program to do the same thing many times, we can use **iteration**. A generic iteration (also called **looping**) mechanism is shown in the boxed-in part of Figure 2.4. Like a conditional statement, it begins with a test. If the test evaluates to `True`, the program executes the **loop** body once, and then goes back to reevaluate the test. This process is repeated until the test evaluates to `False`, after which control passes to the code following the iteration statement.

We can write the kind of loop depicted in Figure 2.4 using a **while** statement. Consider the following example:

```
# Square an integer, the hard way
x = 3
ans = 0
itersLeft = x
while (itersLeft != 0):
    ans = ans + x
    itersLeft = itersLeft - 1
print(str(x) + '*' + str(x) + ' = ' + str(ans))
```

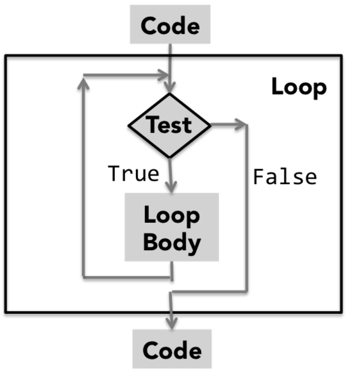


Figure 2.4 Flow chart for iteration

The code starts by binding the variable `x` to the integer `3`. It then proceeds to square `x` by using repetitive addition. The table in Figure 2.5 shows the value associated with each variable each time the test at the start of the loop is reached. We constructed it by **hand-simulating** the code, i.e., we pretended to be a Python interpreter and executed the program using pencil and paper. Using pencil and paper might seem kind of quaint, but it is an excellent way to understand how a program behaves.[16]

Test #	x	ans	itersLeft
1	3	0	3
2	3	3	2
3	3	6	1
4	3	9	0

Figure 2.5 Hand simulation of a small program

The fourth time the test is reached, it evaluates to `False` and flow of control proceeds to the `print` statement following the loop. For what values of `x` will this program terminate? There are three cases to consider: `x == 0`, `x > 0`, and `x < 0`.

Suppose `x == 0`. The initial value of `itersLeft` will also be `0`, and the loop body will never be executed.

Suppose `x > 0`. The initial value of `itersLeft` will be greater than `0`, and the loop body will be executed at least once. Each time the loop body is executed, the value of `itersLeft` is decreased by exactly `1`. This means that if `itersLeft` started out greater than `0`, after some finite number of iterations of the loop, `itersLeft`

[16] It is also possible to hand-simulate a program using pen and paper, or even a text editor.

will equal `0`. At this point the loop test evaluates to `False`, and control proceeds to the code following the `while` statement.

Suppose `x < 0`. Something very bad happens. Control will enter the loop, and each iteration will move `itersLeft` farther from `0` rather than closer to it. The program will therefore continue executing the loop forever (or until something else bad, e.g., an overflow error, occurs). How might we remove this flaw in the program? Initializing `itersLeft` to the absolute value of `x` almost works. The loop terminates, but it prints a negative value. If the assignment statement inside the loop is also changed, to `ans = ans + abs(x)`, the code works properly.

Finger exercise: Replace the comment in the following code with a `while` loop.

```
numXs = int(input('How many times should I print the letter X? '))
toPrint = ''
#concatenate X to toPrint numXs times
print(toPrint)
```

It is sometimes convenient to exit a loop without testing the loop condition. Executing a **break** statement terminates the loop in which it is contained, and transfers control to the code immediately following the loop. For example, the code

```
#Find a positive integer that is divisible by both 11 and 12
x = 1
while True:
    if x%11 == 0 and x%12 == 0:
        break
    x = x + 1
print(x, 'is divisible by 11 and 12')
```

prints

```
132 is divisible by 11 and 12
```

If a `break` statement is executed inside a nested loop (a loop inside another loop), the break will terminate the inner loop.

We have now covered pretty much everything about Python that we need to know to start writing interesting programs that deal with numbers and strings. In the next chapter, we take a short break from learning Python, and use what we have already learned to solve some simple problems.

Finger exercise: Write a program that asks the user to input 10 integers, and then prints the largest odd number that was entered. If no odd number was entered, it should print a message to that effect.

3 SOME SIMPLE NUMERICAL PROGRAMS

Now that we have covered some basic Python constructs, it is time to start thinking about how we can combine those constructs to write some simple programs. Along the way, we'll sneak in a few more language constructs and some algorithmic techniques.

3.1 Exhaustive Enumeration

The code in Figure 3.1 prints the integer cube root, if it exists, of an integer. If the input is not a perfect cube, it prints a message to that effect.

```
#Find the cube root of a perfect cube
x = int(input('Enter an integer: '))
ans = 0
while ans**3 < abs(x):
    ans = ans + 1
if ans**3 != abs(x):
    print(x, 'is not a perfect cube')
else:
    if x < 0:
        ans = -ans
    print('Cube root of', x,'is', ans)
```

Figure 3.1 Using exhaustive enumeration to find the cube root

For what values of `x` will this program terminate? The answer is, "all integers." This can be argued quite simply.

- The value of the expression `ans**3` starts at `0`, and gets larger each time through the loop.
- When it reaches or exceeds `abs(x)`, the loop terminates.
- Since `abs(x)` is always positive there are only a finite number of iterations before the loop must terminate.

Whenever you write a loop, you should think about an appropriate **decrementing function**. This is a function that has the following properties:

- It maps a set of program variables into an integer.
- When the loop is entered, its value is nonnegative.
- When its value is ≤ 0, the loop terminates.
- Its value is decreased every time through the loop.

What is the decrementing function for the `while` loop in Figure 3.1? It is `abs(x) - ans**3`.

Now, let's insert some errors and see what happens. First, try commenting out the statement `ans = 0`. The Python interpreter prints the error message

```
NameError: name 'ans' is not defined
```

because the interpreter attempts to find the value to which `ans` is bound before it has been bound to anything. Now, restore the initialization of `ans`, replace the statement `ans = ans + 1` by `ans = ans`, and try finding the cube root of 8. After you get tired of waiting, enter "control c" (hold down the control key and the c key simultaneously). This will return you to the user prompt in the shell.

Now, add the statement

```
print('Value of the decrementing function abs(x) - ans**3 is',
      abs(x) - ans**3)
```

at the start of the loop, and try running it again. This time it will print

```
Value of the decrementing function abs(x) - ans**3 is 8
```

over and over again.

The program would have run forever because the loop body is no longer reducing the distance between `ans**3` and `abs(x)`. When confronted with a program that seems not to be terminating, experienced programmers often insert print statements, such as the one here, to test whether the decrementing function is indeed being decremented.

The algorithmic technique used in this program is a variant of **guess and check** called **exhaustive enumeration**. We enumerate all possibilities until we get to the right answer or exhaust the space of possibilities. At first blush, this may seem like an incredibly stupid way to solve a problem. Surprisingly, however, exhaustive enumeration algorithms are often the most practical way to solve a problem. They are typically easy to implement and easy to understand. And, in many cases, they run fast enough for all practical purposes. Make sure to remove or comment out the print statement that you inserted and to reinsert the statement `ans = ans + 1`, and then try finding the cube root of 1957816251. The program will finish almost instantaneously. Now, try 7406961012236344616.

As you can see, even if millions of guesses are required, it's not usually a problem. Modern computers are amazingly fast. It takes on the order of one nanosecond—one billionth of a second—to execute an instruction. It's a bit hard to appreciate how fast that is. For perspective, it takes slightly more than a nanosecond for light to travel a single foot (0.3 meters). Another way to think about this is that in the time it takes for the sound of your voice to travel a hundred feet, a modern computer can execute millions of instructions.

Just for fun, try executing the code

```
maxVal = int(input('Enter a postive integer: '))
i = 0
while i < maxVal:
    i = i + 1
print(i)
```

See how large an integer you need to enter before there is a perceptible pause before the result is printed.

Finger exercise: Write a program that asks the user to enter an integer and prints two integers, `root` and `pwr`, such that `0 < pwr < 6` and `root**pwr` is equal to the integer entered by the user. If no such pair of integers exists, it should print a message to that effect.

3.2 For Loops

The `while` loops we have used so far are highly stylized. Each iterates over a sequence of integers. Python provides a language mechanism, the **for loop**, that can be used to simplify programs containing this kind of iteration.

The general form of a `for` statement is (recall that the words in italics are descriptions of what can appear, not actual code):

```
for variable in sequence:
    code block
```

The variable following `for` is bound to the first value in the sequence, and the code block is executed. The variable is then assigned the second value in the sequence, and the code block is executed again. The process continues until the sequence is exhausted or a `break` statement is executed within the code block.

The sequence of values bound to *variable* is most commonly generated using the built-in function **range**, which returns a series of integers. The `range` function takes three integer arguments: `start`, `stop`, and `step`. It produces the progression `start, start + step, start + 2*step`, etc. If `step` is positive, the last element is the largest integer `start + i*step` less than `stop`. If `step` is negative, the last element is the smallest integer `start + i*step` greater than `stop`. For example, the expression `range(5, 40, 10)` yields the sequence `5, 15, 25, 35`, and `range(40, 5, -10)` yields the sequence `40, 30, 20, 10`. If the first argument is omitted it defaults to `0`, and if the last argument (the step size) is omitted it defaults to `1`. For example, `range(0, 3)` and `range(3)` both produce the sequence `0, 1, 2`. The numbers in the progression are generated on an "as needed" basis, so even expressions such as `range(1000000)` consume little memory.[17] We will discuss `range` in more depth in Section 5.2.

Less commonly, we specify the sequence to be iterated over in a `for` loop by using a literal, e.g., `[0, 3, 2]`.

Consider the code

```
x = 4
for i in range(0, x):
    print(i)
```

It prints

```
0
1
2
3
```

Now, think about the code

```
x = 4
for i in range(0, x):
    print(i)
    x = 5
```

It raises the question of whether changing the value of `x` inside the loop affects the number of iterations. It does not. The arguments to the `range` function in the line with `for` are evaluated just before the first iteration of the loop, and not reevaluated for subsequent iterations.

To see how this works, consider

```
x = 4
for j in range(x):
    for i in range(x):
        print(i)
        x = 2
```

[17] In Python 2, `range` generates the entire sequence when invoked. Therefore, expressions such as `range(1000000)` use quite a lot of memory. In Python 2, `xrange` behaves the way `range` behaves in Python 3.

which prints

```
0
1
2
3
0
1
0
1
0
1
```

because the range function in the outer loop is evaluated only once, but the range function in the inner loop is evaluated each time the inner for statement is reached.

The code in Figure 3.2 reimplements the exhaustive enumeration algorithm for finding cube roots. The break statement in the for loop causes the loop to terminate before it has been run on each element in the sequence over which it is iterating.

```
#Find the cube root of a perfect cube
x = int(input('Enter an integer: '))
for ans in range(0, abs(x)+1):
    if ans**3 >= abs(x):
        break
if ans**3 != abs(x):
    print(x, 'is not a perfect cube')
else:
    if x < 0:
        ans = -ans
    print('Cube root of', x,'is', ans)
```

Figure 3.2 Using for and break statements

The for statement can be used in conjunction with the **in operator** to conveniently iterate over characters of a string. For example,

```
total = 0
for c in '12345678':
    total = total + int(c)
print(total)
```

sums the digits in the string denoted by the literal '12345678' and prints the total.

Finger exercise: Let s be a string that contains a sequence of decimal numbers separated by commas, e.g., `s = '1.23,2.4,3.123'`. Write a program that prints the sum of the numbers in s.

3.3 Approximate Solutions and Bisection Search

Imagine that someone asks you to write a program that finds the square root of any nonnegative number. What should you do?

You should probably start by saying that you need a better problem statement. For example, what should the program do if asked to find the square root of 2? The square root of 2 is not a rational number. This means that there is no way to precisely represent its value as a finite string of digits (or as a `float`), so the problem as initially stated cannot be solved.

The right thing to have asked for is a program that finds an **approximation** to the square root—i.e., an answer that is close enough to the actual square root to be useful. We will return to this issue in considerable detail later in the book. But for now, let's think of "close enough" as an answer that lies within some constant, call it `epsilon`, of the actual answer.

The code in Figure 3.3 implements an algorithm that finds an approximation to a square root. It uses an operator, `+=`, that we have not previously used. The assignment statement `ans += step` is semantically equivalent to the more verbose code `ans = ans+step`. The operators `-=` and `*=` work similarly.

```
x = 25
epsilon = 0.01
step = epsilon**2
numGuesses = 0
ans = 0.0
while abs(ans**2 - x) >= epsilon and ans <= x:
    ans += step
    numGuesses += 1
print('numGuesses =', numGuesses)
if abs(ans**2 - x) >= epsilon:
    print('Failed on square root of', x)
else:
    print(ans, 'is close to square root of', x)
```

Figure 3.3 Approximating the square root using exhaustive enumeration

Once again, we are using exhaustive enumeration. Notice that this method for finding the square root has nothing in common with the way of finding square roots using a pencil that you might have learned in middle school. It is often the case that the best way to solve a problem with a computer is quite different from how one would approach the problem by hand.

When the code is run, it prints

```
numGuesses = 49990
4.999000000001688 is close to square root of 25
```

Should we be disappointed that the program didn't figure out that 25 is a perfect square and print 5? No. The program did what it was intended to do. Though it would have been OK to print 5, doing so is no better than printing any value close enough to 5.

What do you think will happen if we set `x = 0.25`? Will it find a root close to 0.5? Nope. It will report

```
numGuesses = 2501
Failed on square root of 0.25
```

Exhaustive enumeration is a search technique that works only if the set of values being searched includes the answer. In this case, we are enumerating the values between 0 and the value of x. When x is between 0 and 1, the square root of x does not lie in this interval. One way to fix this is to change the second operand of `and` in the first line of the `while` loop to get

```
while abs(ans**2 - x) >= epsilon and ans*ans <= x:
```

Now, let's think about how long the program will take to run. The number of iterations depends upon how close the answer is to 0 and on the size of the steps. Roughly speaking, the program will execute the `while` loop at most `x/step` times.

Let's try the code on something bigger, e.g., `x = 123456`. It will run for a bit, and then print

```
numGuesses = 3513631
Failed on square root of 123456
```

What do you think happened? Surely there exists a floating point number that approximates the square root of 123456 to within 0.01. Why didn't our program find it? The problem is that our step size was too large, and the program skipped over all the suitable answers. Try making `step` equal to `epsilon**3` and running the program. It will eventually find a suitable answer, but you might not have the patience to wait for it to do so.

Roughly how many guesses will it have to make? The step size will be 0.000001 and the square root of 123456 is around 351.36. This means that the program will have to make in the neighborhood of 351,000,000 guesses to find a satisfactory answer. We could try to speed it up by starting closer to the answer, but that presumes that we know the answer.

The time has come to look for a different way to attack the problem. We need to choose a better algorithm rather than fine-tune the current one. But before doing so, let's look at a problem that, at first blush, appears to be completely different from root finding.

Consider the problem of discovering whether a word starting with a given sequence of letters appears in some hard-copy dictionary of the English language. Exhaustive enumeration would, in principle, work. You could start at the first word and examine each word until either you found a word starting with the sequence of letters or you ran out of words to examine. If the dictionary contained n words, it would, on average, take n/2 probes to find the word. If the word were not in the dictionary, it would take n probes. Of course, those who have had the pleasure of actually looking a word up in a physical (rather than online) dictionary would never proceed in this way.

Fortunately, the folks who publish hardcopy dictionaries go to the trouble of putting the words in lexicographical order. This allows us to open the book to a page where we think the word might lie (e.g., near the middle for words starting with the letter m). If the sequence of letters lexicographically precedes the first word on the page, we know to go backwards. If the sequence of letters follows the last word on the page, we know to go forwards. Otherwise, we check whether the sequence of letters matches a word on the page.

Now let's take the same idea and apply it the problem of finding the square root of x. Suppose we know that a good approximation to the square root of x lies somewhere between 0 and max. We can exploit the fact that numbers are **totally ordered**. That is, for any pair of distinct numbers, n1 and n2, either n1 < n2 or n1 > n2. So, we can think of the square root of x as lying somewhere on the line

0____max

and start searching that interval. Since we don't necessarily know where to start searching, let's start in the middle.

0__________________________guess__________________________max

If that is not the right answer (and it won't be most of the time), ask whether it is too big or too small. If it is too big, we know that the answer must lie to the left. If it is too small, we know that the answer must lie to the right. We then repeat the process on the smaller interval. Figure 3.4 contains an implementation and test of this algorithm.

```
x = 25
epsilon = 0.01
numGuesses = 0
low = 0.0
high = max(1.0, x)
ans = (high + low)/2.0
while abs(ans**2 - x) >= epsilon:
    print('low =', low, 'high =', high, 'ans =', ans)
    numGuesses += 1
    if ans**2 < x:
        low = ans
    else:
        high = ans
    ans = (high + low)/2.0
print('numGuesses =', numGuesses)
print(ans, 'is close to square root of', x)
```

Figure 3.4 Using bisection search to approximate square root

When run, it prints

```
low = 0.0 high = 25 ans = 12.5
low = 0.0 high = 12.5 ans = 6.25
low = 0.0 high = 6.25 ans = 3.125
low = 3.125 high = 6.25 ans = 4.6875
low = 4.6875 high = 6.25 ans = 5.46875
low = 4.6875 high = 5.46875 ans = 5.078125
low = 4.6875 high = 5.078125 ans = 4.8828125
low = 4.8828125 high = 5.078125 ans = 4.98046875
low = 4.98046875 high = 5.078125 ans = 5.029296875
low = 4.98046875 high = 5.029296875 ans = 5.0048828125
low = 4.98046875 high = 5.0048828125 ans = 4.99267578125
low = 4.99267578125 high = 5.0048828125 ans = 4.998779296875
low = 4.998779296875 high = 5.0048828125 ans = 5.0018310546875
numGuesses = 13
5.00030517578125 is close to square root of 25
```

Notice that it finds a different answer than our earlier algorithm. That is perfectly fine, since it still meets the problem's specification.

More important, notice that at each iteration of the loop the size of the space to be searched is cut in half. Because it divides the search space in half at each step, it is called a **bisection search**. Bisection search is a huge improvement over our earlier algorithm, which reduced the search space by only a small amount at each iteration.

Let us try `x = 123456` again. This time the program takes only thirty guesses to find an acceptable answer. How about `x = 123456789` ? It takes only forty-five guesses.

There is nothing special about the fact that we are using this algorithm to find square roots. For example, by changing a couple of 2's to 3's, we can use it to approximate a cube root of a nonnegative number. In Chapter 4, we introduce a language mechanism that allows us to generalize this code to find any root.

Finger exercise: What would the code in Figure 3.4 do if the statement `x = 25` were replaced by `x = -25`?

Finger exercise: What would have to be changed to make the code in Figure 3.4 work for finding an approximation to the cube root of both negative and positive numbers? (Hint: think about changing `low` to ensure that the answer lies within the region being searched.)

3.4 A Few Words About Using Floats

Most of the time, numbers of type `float` provide a reasonably good approximation to real numbers. But "most of the time" is not all of the time, and when they don't it can lead to surprising consequences. For example, try running the code

```
x = 0.0
for i in range(10):
    x = x + 0.1
if x == 1.0:
    print(x, '= 1.0')
else:
    print(x, 'is not 1.0')s
```

Perhaps you, like most people, find it surprising that it prints,

```
0.9999999999999999 is not 1.0
```

Why does it get to the `else` clause in the first place?

To understand why this happens, we need to understand how floating point numbers are represented in the computer during a computation. To understand that, we need to understand **binary numbers**.

When you first learned about decimal numbers—i.e., numbers base 10—you learned that any decimal number can be represented by a sequence of the digits 0123456789. The rightmost digit is the 10^0 place, the next digit towards the left the 10^1 place, etc. For example, the sequence of decimal digits 302 represents

3*100 + 0*10 + 2*1. How many different numbers can be represented by a sequence of length n? A sequence of length 1 can represent any one of ten numbers (0-9); a sequence of length 2 can represent one hundred numbers (0-99). More generally, with a sequence of length n, one can represent 10^n different numbers.

Binary numbers—numbers base 2—work similarly. A binary number is represented by a sequence of digits each of which is either 0 or 1. These digits are often called **bits**. The rightmost digit is the 2^0 place, the next digit towards the left the 2^1 place, etc. For example, the sequence of binary digits 101 represents 1*4 + 0*2 + 1*1 = 5. How many different numbers can be represented by a sequence of length n? 2^n.

Finger exercise: What is the decimal equivalent of the binary number 10011?

Perhaps because most people have ten fingers, we seem to like to use decimals to represent numbers. On the other hand, all modern computer systems represent numbers in binary. This is not because computers are born with two fingers. It is because it is easy to build hardware switches, i.e., devices that can be in only one of two states, on or off. That the computer uses a binary representation and people a decimal representation can lead to occasional cognitive dissonance.

In almost modern programming languages non-integer numbers are implemented using a representation called **floating point**. For the moment, let's pretend that the internal representation is in decimal. We would represent a number as a pair of integers—the **significant digits** of the number and an **exponent**. For example, the number 1.949 would be represented as the pair (1949, -3), which stands for the product $1949*10^{-3}$.

The number of significant digits determines the **precision** with which numbers can be represented. If for example, there were only two significant digits, the number 1.949 could not be represented exactly. It would have to be converted to some approximation of 1.949, in this case 1.9. That approximation is called the **rounded value**.

Modern computers use binary, not decimal, representations. We represent the significant digits and exponents in binary rather than decimal and raise 2 rather than 10 to the exponent. For example, the number 0.625 (5/8) would be represented as the pair (101, -11); because 5/8 is 0.101 in binary and -11 is the binary representation of -3, the pair (101, -11) stands for $5*2^{-3} = 5/8 = 0.625$.

What about the decimal fraction 1/10, which we write in Python as `0.1`? The best we can do with four significant binary digits is (0011, -101). This is equivalent to 3/32, i.e., 0.09375. If we had five significant binary digits, we would represent 0.1 as (11001, -1000), which is equivalent to 25/256, i.e., 0.09765625. How many significant digits would we need to get an exact floating point representation of 0.1? An infinite number of digits! There do not exist integers sig and exp such that sig * 2^{-exp} equals 0.1. So no matter how many bits Python (or any other language) chooses to use to represent floating point numbers, it will be able to represent only an approximation to 0.1. In most Python implementations, there are 53 bits of precision available for floating point numbers, so the significant digits stored for the decimal number 0.1 will be

```
11001100110011001100110011001100110011001100110011001
```

This is equivalent to the decimal number

```
0.1000000000000000055511151231257827021181583404541015625
```

Pretty close to 1/10, but not exactly 1/10.

Returning to the original mystery, why does

```
x = 0.0
for i in range(10):
    x = x + 0.1
if x == 1.0:
    print(x, '= 1.0')
else:
    print(x, 'is not 1.0')
```

print

```
0.9999999999999999 is not 1.0
```

We now see that the test `x == 1.0` produces the result `False` because the value to which `x` is bound is not exactly `1.0`. What gets printed if we add to the end of the `else` clause the code `print x == 10.0*0.1`? It prints `False` because during at least one iteration of the loop Python ran out of significant digits and did some rounding. It's not what our elementary school teachers taught us, but adding `0.1` ten times does not produce the same value as multiplying `0.1` by 10.[18]

By the way, if you want to explicitly round a floating point number, use the `round` function. The expression `round(x, numDigits)` returns the floating point number equivalent to rounding the value of `x` to `numDigits` decimal digits following the decimal point. For example, print `round(2**0.5, 3)` will print `1.414` as an approximation to the square root of 2.

[18] In Python 2 another strange thing happens. Because the `print` statement does some automatic rounding, the `else` clause would print `1.0 is not 1.0`.

Does the difference between real and floating point numbers really matter? Most of the time, mercifully, it does not. There are few situations where `1.0` is an acceptable answer and `0.9999999999999999` is not. However, one thing that is almost always worth worrying about is tests for equality. As we have seen, using `==` to compare two floating point values can produce a surprising result. It is almost always more appropriate to ask whether two floating point values are close enough to each other, not whether they are identical. So, for example, it is better to write `abs(x-y) < 0.0001` rather than `x == y`.

Another thing to worry about is the accumulation of rounding errors. Most of the time things work out OK, because sometimes the number stored in the computer is a little bigger than intended, and sometimes it is a little smaller than intended. However, in some programs, the errors will all be in the same direction and accumulate over time.

3.5 Newton-Raphson

The most commonly used approximation algorithm is usually attributed to Isaac Newton. It is typically called Newton's method, but is sometimes referred to as the Newton-Raphson method.[19] It can be used to find the real roots of many functions, but we shall look at it only in the context of finding the real roots of a polynomial with one variable. The generalization to polynomials with multiple variables is straightforward both mathematically and algorithmically.

A **polynomial** with one variable (by convention, we will write the variable as x) is either 0 or the sum of a finite number of nonzero terms, e.g., $3x^2 + 2x + 3$. Each term, e.g., $3x^2$, consists of a constant (the **coefficient** of the term, 3 in this case) multiplied by the variable (x in this case) raised to a nonnegative integer exponent (2 in this case). The exponent on a variable in a term is called the **degree** of that term. The degree of a polynomial is the largest degree of any single term. Some examples are, 3 (degree 0), $2.5x + 12$ (degree 1), and $3x^2$ (degree 2). In contrast, $2/x$ and $x^{0.5}$ are not polynomials.

If p is a polynomial and r a real number, we will write p(r) to stand for the value of the polynomial when $x = r$. A **root** of the polynomial p is a solution to the equation $p = 0$, i.e., an r such that $p(r) = 0$. So, for example, the problem of finding an approximation to the square root of 24 can be formulated as finding an x such that $x^2 - 24 \approx 0$.

[19] Joseph Raphson published a similar method about the same time as Newton.

Newton proved a theorem that implies that if a value, call it guess, is an approximation to a root of a polynomial, then guess – p(guess)/p'(guess), where p' is the first derivative of p, is a better approximation.[20]

For any constant k and any coefficient c, the first derivative of the polynomial $cx^2 + k$ is 2cx. For example, the first derivative of $x^2 - k$ is 2x. Therefore, we know that we can improve on the current guess, call it y, by choosing as our next guess $y - (y^2 - k)/2y$. This is called **successive approximation**. Figure 3.5 contains code illustrating how to use this idea to quickly find an approximation to the square root.

```
#Newton-Raphson for square root
#Find x such that x**2 - 24 is within epsilon of 0
epsilon = 0.01
k = 24.0
guess = k/2.0
while abs(guess*guess - k) >= epsilon:
    guess = guess - (((guess**2) - k)/(2*guess))
print('Square root of', k, 'is about', guess)
```

Figure 3.5 Implementation of Newton-Raphson method

Finger exercise: Add some code to the implementation of Newton-Raphson that keeps track of the number of iterations used to find the root. Use that code as part of a program that compares the efficiency of Newton-Raphson and bisection search. (You should discover that Newton-Raphson is more efficient.)

[20] The first derivative of a function `f(x)` can be thought of as expressing how the value of f(x) changes with respect to changes in `x`. If you haven't previously encountered derivatives, don't worry. You don't need to understand them, or for that matter polynomials, to understand the implementation of Newton's method.

4 FUNCTIONS, SCOPING, AND ABSTRACTION

So far, we have introduced numbers, assignments, input/output, comparisons, and looping constructs. How powerful is this subset of Python? In a theoretical sense, it is as powerful as you will ever need, i.e., it is Turing complete. This means that if a problem can be solved via computation, it can be solved using only those statements you have already seen.

Which isn't to say that you should use only these statements. At this point we have covered a lot of language mechanisms, but the code has been a single sequence of instructions, all merged together. For example, in the last chapter we looked at the code in Figure 4.1.

```
x = 25
epsilon = 0.01
numGuesses = 0
low = 0.0
high = max(1.0, x)
ans = (high + low)/2.0
while abs(ans**2 - x) >= epsilon:
    numGuesses += 1
    if ans**2 < x:
        low = ans
    else:
        high = ans
    ans = (high + low)/2.0
print('numGuesses =', numGuesses)
print(ans, 'is close to square root of', x)
```

Figure 4.1 Using bisection search to approximate square root

This is a reasonable piece of code, but it lacks general utility. It works only for values denoted by the variables `x` and `epsilon`. This means that if we want to reuse it, we need to copy the code, possibly edit the variable names, and paste it where we want it. Because of this we cannot easily use this computation inside of some other, more complex, computation.

Furthermore, if we want to compute cube roots rather than square roots, we have to edit the code. If we want a program that computes both square and cube roots (or for that matter square roots in two different places), the program would contain multiple chunks of almost identical code. This is a very bad thing. The more code a program contains, the more chance there is for something to go wrong, and the harder the code is to maintain. Imagine, for example, that there was an error in the initial implementation of square root, and that the error came to light when testing the program. It would be all too easy to fix the implementation of square root in one place and forget that there was similar code elsewhere that was also in need of repair.

Python provides several linguistic features that make it relatively easy to generalize and reuse code. The most important is the function.

4.1 Functions and Scoping

We've already used a number of built-in functions, e.g., `max` and `abs` in Figure 4.1. The ability for programmers to define and then use their own functions, as if they were built-in, is a qualitative leap forward in convenience.

4.1.1 Function Definitions

In Python each **function definition** is of the form[21]

```
def name of function (list of formal parameters):
    body of function
```

For example, we could define the function `maxVal`[22] by the code

```
def maxVal(x, y):
    if x > y:
        return x
    else:
        return y
```

`def` is a reserved word that tells Python that a function is about to be defined. The function name (`maxVal` in this example) is simply a name that is used to refer to the function.

The sequence of names within the parentheses following the function name (`x, y` in this example) are the **formal parameters** of the function. When the function is used, the formal parameters are bound (as in an assignment statement) to

[21] Recall that italic is used to describe kinds of Python code.

[22] In practice, you would probably use the built-in function `max`, rather than define your own function.

the **actual parameters** (often referred to as **arguments**) of the **function invocation** (also referred to as a **function call**). For example, the invocation

```
maxVal(3, 4)
```

binds `x` to `3` and `y` to `4`.

The function body is any piece of Python code.[23] There is, however, a special statement, **`return`**, that can be used only within the body of a function.

A function call is an expression, and like all expressions it has a value. That value is the value returned by the invoked function. For example, the value of the expression `maxVal(3,4)*maxVal(3,2)` is `12`, because the first invocation of `maxVal` returns the `int` `4` and the second returns the `int` `3`. Note that execution of a `return` statement terminates an invocation of the function.

To recapitulate, when a function is called

1. The expressions that make up the actual parameters are evaluated, and the formal parameters of the function are bound to the resulting values. For example, the invocation `maxVal(3+4, z)` will bind the formal parameter `x` to `7` and the formal parameter `y` to whatever value the variable `z` has when the invocation is evaluated.
2. The **point of execution** (the next instruction to be executed) moves from the point of invocation to the first statement in the body of the function.
3. The code in the body of the function is executed until either a `return` statement is encountered, in which case the value of the expression following the `return` becomes the value of the function invocation, or there are no more statements to execute, in which case the function returns the value `None`. (If no expression follows the `return`, the value of the invocation is `None`.)
4. The value of the invocation is the returned value.
5. The point of execution is transferred back to the code immediately following the invocation.

Parameters provide something called **lambda abstraction**,[24] allowing programmers to write code that manipulates not specific objects, but instead whatever objects the caller of the function chooses to use as actual parameters.

[23] As we will see later, this notion of function is more general than what mathematicians call a function. It was first popularized by the programming language Fortran 2 in the late 1950s.

[24] The name "lambda abstraction" is derived from mathematics developed by Alonzo Church in the 1930s and 1940s.

Finger exercise: Write a function isIn that accepts two strings as arguments and returns True if either string occurs anywhere in the other, and False otherwise. Hint: you might want to use the built-in str operation in.

4.1.2 Keyword Arguments and Default Values

In Python, there are two ways that formal parameters get bound to actual parameters. The most common method, which is the only one we have used thus far, is called **positional**—the first formal parameter is bound to the first actual parameter, the second formal to the second actual, etc. Python also supports **keyword arguments**, in which formals are bound to actuals using the name of the formal parameter. Consider the function definition

```
def printName(firstName, lastName, reverse):
    if reverse:
       print(lastName + ', ' + firstName)
    else:
       print(firstName, lastName)
```

The function printName assumes that firstName and lastName are strings and that reverse is a Boolean. If reverse == True, it prints lastName, firstName, otherwise it prints firstName lastName.

Each of the following is an equivalent invocation of printName:

```
printName('Olga', 'Puchmajerova', False)
printName('Olga', 'Puchmajerova', reverse = False)
printName('Olga', lastName = 'Puchmajerova', reverse = False)
printName(lastName = 'Puchmajerova', firstName = ' Olga',
          reverse = False)
```

Though the keyword arguments can appear in any order in the list of actual parameters, it is not legal to follow a keyword argument with a non-keyword argument. Therefore, an error message would be produced by

```
printName('Olga', lastName = 'Puchmajerova', False)
```

Keyword arguments are commonly used in conjunction with **default parameter values**. We can, for example, write

```
def printName(firstName, lastName, reverse = False):
    if reverse:
       print(lastName + ', ' + firstName)
    else:
       print(firstName, lastName)
```

Default values allow programmers to call a function with fewer than the specified number of arguments. For example,

```
printName('Olga', 'Puchmajerova')
printName('Olga', 'Puchmajerova', True)
printName('Olga', 'Puchmajerova', reverse = True)
```

will print

```
Olga Puchmajerova
Puchmajerova, Olga
Puchmajerova, Olga
```

The last two invocations of `printName` are semantically equivalent. The last one has the advantage of providing some documentation for the perhaps mysterious argument `True`.

4.1.3 Scoping

Let's look at another small example,

```
def f(x): #name x used as formal parameter
    y = 1
    x = x + y
    print('x =', x)
    return x

x = 3
y = 2
z = f(x) #value of x used as actual parameter
print('z =', z)
print('x =', x)
print('y =', y)
```

When run, this code prints,

```
x = 4
z = 4
x = 3
y = 2
```

What is going on here? At the call of `f`, the formal parameter `x` is locally bound to the value of the actual parameter `x`. It is important to note that though the actual and formal parameters have the same name, they are not the same variable. Each function defines a new **name space**, also called a **scope**. The formal parameter `x` and the **local variable** `y` that are used in `f` exist only within the scope of the definition of `f`. The assignment statement `x = x + y` within the function body binds the local name `x` to the object `4`. The assignments in `f` have no effect at all on the bindings of the names `x` and `y` that exist outside the scope of `f`.

Here's one way to think about this:

1. At top level, i.e., the level of the shell, a **symbol table** keeps track of all names defined at that level and their current bindings.
2. When a function is called, a new symbol table (often called a **stack frame**) is created. This table keeps track of all names defined within the function (including the formal parameters) and their current bindings. If a function is called from within the function body, yet another stack frame is created.
3. When the function completes, its stack frame goes away.

In Python, one can always determine the scope of a name by looking at the program text. This is called **static** or **lexical scoping**. Figure 4.2 contains an example illustrating Python's scope rules.

```
def f(x):
   def g():
      x = 'abc'
      print('x =', x)
   def h():
      z = x
      print('z =', z)
   x = x + 1
   print('x =', x)
   h()
   g()
   print('x =', x)
   return g

x = 3
z = f(x)
print('x =', x)
print('z =', z)
z()
```

Figure 4.2 Nested scopes

The history of the stack frames associated with the code is depicted in Figure 4.3. The first column contains the set of names known outside the body of the function `f`, i.e., the variables `x` and `z`, and the function name `f`. The first assignment statement binds `x` to `3`.

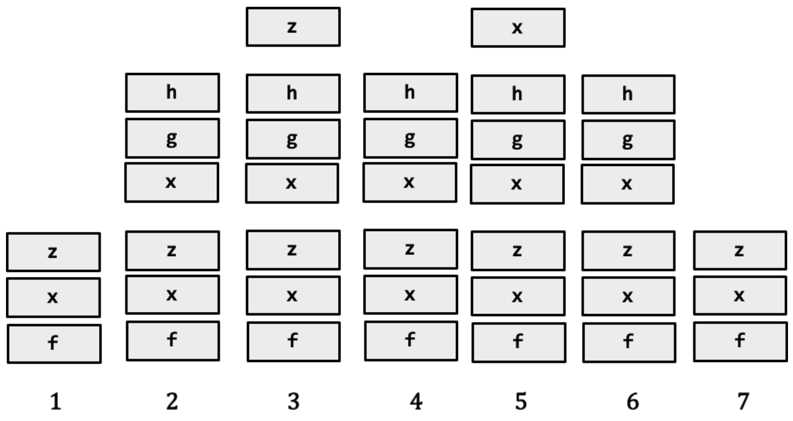


Figure 4.3 Stack frames

The assignment statement `z = f(x)` first evaluates the expression `f(x)` by invoking the function `f` with the value to which `x` is bound. When `f` is entered, a stack frame is created, as shown in column 2. The names in the stack frame are `x` (the formal parameter, not the `x` in the calling context), `g` and `h`. The variables `g` and `h` are bound to objects of type `function`. The properties of these functions are given by the function definitions within `f`.

When `h` is invoked from within `f`, yet another stack frame is created, as shown in column 3. This frame contains only the local variable `z`. Why does it not also contain `x`? A name is added to the scope associated with a function only if that name is either a formal parameter of the function or a variable that is bound to an object within the body of the function. In the body of `h`, `x` occurs only on the right-hand side of an assignment statement. The appearance of a name (`x` in this case) that is not bound to an object anywhere in the function body (the body of `h`) causes the interpreter to search the stack frame associated with the scope within which the function is defined (the stack frame associated with `f`). If the name is found (which it is in this case) the value to which it is bound (`4`) is used. If it is not found there, an error message is produced.

When `h` returns, the stack frame associated with the invocation of `h` goes away (it is **popped** off the top of the stack), as depicted in column 4. Note that we never remove frames from the middle of the stack, but only the most recently added frame. It is because it has this "last in first out" behavior that we refer to it as a **stack** (think of a stack of trays waiting to be taken in a cafeteria).

Next g is invoked, and a stack frame containing g's local variable x is added (column 5). When g returns, that frame is popped (column 6). When f returns, the stack frame containing the names associated with f is popped, getting us back to the original stack frame (column 7).

Notice that when f returns, even though the variable g no longer exists, the object of type function to which that name was once bound still exists. This is because functions are objects, and can be returned just like any other kind of object. So, z can be bound to the value returned by f, and the function call z() can be used to invoke the function that was bound to the name g within f—even though the name g is not known outside the context of f.

So, what does the code in Figure 4.2 print? It prints

```
x = 4
z = 4
x = abc
x = 4
x = 3
z = <function f.<locals>.g at 0x1092a7510>
x = abc
```

The order in which references to a name occur is not germane. If an object is bound to a name anywhere in the function body (even if it occurs in an expression before it appears as the left-hand side of an assignment), it is treated as local to that function.[25] Consider, for example, the code

```
def f():
   print(x)

def g():
   print(x)
   x = 1

x = 3
f()
x = 3
g()
```

It prints 3 when f is invoked, but the error message

```
UnboundLocalError: local variable 'x' referenced before assignment
```

is printed when the print statement in g is encountered. This happens because the assignment statement following the print statement causes x to be local to g. And because x is local to g, it has no value when the print statement is executed.

[25] The wisdom of this language design decision is debatable.

Confused yet? It takes most people a bit of time to get their head around scope rules. Don't let this bother you. For now, charge ahead and start using functions. Most of the time you will find that you only want to use variables that are local to a function, and the subtleties of scoping will be irrelevant.

4.2 Specifications

Figure 4.4 defines a function, `findRoot`, that generalizes the bisection search we used to find square roots in Figure 4.1. It also contains a function, `testFindRoot`, that can be used to test whether or not `findRoot` works as intended.

```
def findRoot(x, power, epsilon):
    """Assumes x and epsilon int or float, power an int,
           epsilon > 0 & power >= 1
       Returns float y such that y**power is within epsilon of x.
           If such a float does not exist, it returns None"""
    if x < 0 and power%2 == 0: #Negative number has no even-powered
                               #roots
        return None
    low = min(-1.0, x)
    high = max(1.0, x)
    ans = (high + low)/2.0
    while abs(ans**power - x) >= epsilon:
        if ans**power < x:
            low = ans
        else:
            high = ans
        ans = (high + low)/2.0
    return ans

def testFindRoot():
    epsilon = 0.0001
    for x in [0.25, -0.25, 2, -2, 8, -8]:
        for power in range(1, 4):
            print('Testing x =', str(x), 'and power = ', power)
            result = findRoot(x, power, epsilon)
            if result == None:
                print('   No root')
            else:
                print('   ', result**power, '~=', x)
```

Figure 4.4 Finding an approximation to a root

The function `testFindRoot` is almost as long as `findRoot` itself. To inexperienced programmers, writing **test functions** such as this often seems to be a waste of effort. Experienced programmers know, however, that an investment in writing testing code often pays big dividends. It certainly beats typing test cases into the shell over and over again during **debugging** (the process of finding out why a program does not work, and then fixing it). It also forces us to think about which tests are likely to be most illuminating.

The text between the triple quotation marks is called a **docstring** in Python. By convention, Python programmers use docstrings to provide specifications of functions. These docstrings can be accessed using the built-in function **`help`**. For example, if we enter the shell and type `help(abs)`, the system will display

```
Help on built-in function abs in module built-ins:

abs(x)
    Return the absolute value of the argument.
```

If the code in Figure 4.4 has been loaded into an IDE, typing `help(findRoot)` in the shell will display

```
findRoot(x, power, epsilon)
    Assumes x and epsilon int or float, power an int,
        epsilon > 0 & power >= 1
    Returns float y such that y**power is within epsilon of x.
        If such a float does not exist, it returns None
```

If we type `findRoot(` in the editor, the list of formal parameters will be displayed.

A **specification** of a function defines a contract between the implementer of a function and those who will be writing programs that use the function. We will refer to the users of a function as its **clients**. This contract can be thought of as containing two parts:

- **Assumptions**: These describe conditions that must be met by clients of the function. Typically, they describe constraints on the actual parameters. Almost always, they specify the acceptable set of types for each parameter, and not infrequently some constraints on the value of one or more of the parameters. For example, the first two lines of the docstring of `findRoot` describe the assumptions that must be satisfied by clients of `findRoot`.
- **Guarantees**: These describe conditions that must be met by the function, provided that it has been called in a way that satisfies the assumptions. The last two lines of the docstring of `findRoot` describe the guarantees that the implementation of the function must meet.

Functions are a way of creating computational elements that we can think of as primitives. Just as we have the built-in functions `max` and `abs`, we would like to have the equivalent of a built-in function for finding roots and for many other complex operations. Functions facilitate this by providing decomposition and abstraction.

Decomposition creates structure. It allows us to break a program into parts that are reasonably self-contained, and that may be reused in different settings.

Abstraction hides detail. It allows us to use a piece of code as if it were a black box—that is, something whose interior details we cannot see, don't need to see, and shouldn't even want to see.[26] The essence of abstraction is preserving information that is relevant in a given context, and forgetting information that is irrelevant in that context. The key to using abstraction effectively in programming is finding a notion of relevance that is appropriate for both the builder of an abstraction and the potential clients of the abstraction. That is the true art of programming.

Abstraction is all about forgetting. There are lots of ways to model this, for example, the auditory apparatus of most teenagers.

> Teenager says: *May I borrow the car tonight?*
>
> Parent says: *Yes, but be back before midnight, and make sure that the gas tank is full.*
>
> Teenager hears: *Yes.*

The teenager has ignored all of those pesky details that he or she considers irrelevant. Abstraction is a many-to-one process. Had the parent said Yes, but be back before 2:00 a.m., and make sure that the car is clean, it would also have been abstracted to Yes.

By way of analogy, imagine that you were asked to produce an introductory computer science course containing twenty-five lectures. One way to do this would be to recruit twenty-five professors and ask each of them to prepare a one-hour lecture on their favorite topic. Though you might get twenty-five wonderful hours, the whole thing is likely to feel like a dramatization of Pirandello's "Six Characters in Search of an Author" (or that political science course you took with fifteen guest lecturers). If each professor worked in isolation, they would have no idea how to relate the material in their lecture to the material covered in other lectures.

Somehow, one needs to let everyone know what everyone else is doing, without generating so much work that nobody is willing to participate. This is where

[26] "Where ignorance is bliss, 'tis folly to be wise."—Thomas Gray

abstraction comes in. You could write twenty-five specifications, each saying what material the students should learn in each lecture, but not giving any detail about how that material should be taught. What you got might not be pedagogically wonderful, but at least it might make sense.

This is the way organizations go about using teams of programmers to get things done. Given a specification of a module, a programmer can work on implementing that module without worrying unduly about what the other programmers on the team are doing. Moreover, the other programmers can use the specification to start writing code that uses that module without worrying unduly about how that module is to be implemented.

The specification of `findRoot` is an abstraction of all the possible implementations that meet the specification. Clients of `findRoot` can assume that the implementation meets the specification, but they should assume nothing more. For example, clients can assume that the call `findRoot(4.0, 2, 0.01)` returns some value whose square is between 3.99 and 4.01. The value returned could be positive or negative, and even though 4.0 is a perfect square, the value returned might not be 2.0 or -2.0.

4.3 Recursion

You may have heard of **recursion**, and in all likelihood think of it as a rather subtle programming technique. That's a charming urban legend spread by computer scientists to make people think that we are smarter than we really are. Recursion is a very important idea, but it's not so subtle, and it is more than a programming technique.

As a descriptive method recursion is widely used, even by people who would never dream of writing a program. Consider part of the legal code of the United States defining the notion of a "natural-born" citizen. Roughly speaking, the definition is as follows

- Any child born inside the United States,
- Any child born in wedlock outside the United States both of whose parents are citizens of the U.S., as long as one parent has lived in the U.S. prior to the birth of the child, and
- Any child born in wedlock outside the United States one of whose parents is a U.S. citizen who has lived at least five years in the U.S. prior to the birth of the child, provided that at least two of those years were after the citizen's fourteenth birthday.

The first part is simple; if you are born in the United States, you are a natural-born citizen (such as Barack Obama). If you are not born in the U.S., then one has to decide if your parents are U.S. citizens (either natural born or naturalized). To determine if your parents are U.S. citizens, you might have to look at your grandparents, and so on.

In general, a recursive definition is made up of two parts. There is at least one **base case** that directly specifies the result for a special case (case 1 in the example above), and there is at least one **recursive (inductive) case** (cases 2 and 3 in the example above) that defines the answer in terms of the answer to the question on some other input, typically a simpler version of the same problem.

The world's simplest recursive definition is probably the factorial function (typically written in mathematics using !) on natural numbers.[27] The classic **inductive definition** is

$$1! = 1$$
$$(n + 1)! = (n + 1) * n!$$

The first equation defines the base case. The second equation defines factorial for all natural numbers, except the base case, in terms of the factorial of the previous number.

Figure 4.5 contains both an iterative (`factI`) and a recursive (`factR`) implementation of factorial.

```
def factI(n):                    def factrR(n):
   """Assumes n an int > 0          """Assumes n an int > 0
      Returns n!"""                    Returns n!"""
   result = 1                       if n == 1:
   while n > 1:                        return n
      result = result * n           else:
      n -= 1                           return n*factR(n - 1)
   return result
```

Figure 4.5 Iterative and recursive implementations of factorial

This function is sufficiently simple that neither implementation is hard to follow. Still, the second is a more obvious translation of the original recursive definition.

[27] The exact definition of "natural number" is subject to debate. Some define it as the positive integers and others as the nonnegative integers. That's why we were explicit about the possible values of `n` in the docstrings in Figure 4.5.

It almost seems like cheating to implement `factR` by calling `factR` from within the body of `factR`. It works for the same reason that the iterative implementation works. We know that the iteration in `factI` will terminate because `n` starts out positive and each time around the loop it is reduced by 1. This means that it cannot be greater than `1` forever. Similarly, if `factR` is called with 1, it returns a value without making a recursive call. When it does make a recursive call, it always does so with a value one less than the value with which it was called. Eventually, the recursion terminates with the call `factR(1)`.

4.3.1 Fibonacci Numbers

The Fibonacci sequence is another common mathematical function that is usually defined recursively. "They breed like rabbits," is often used to describe a population that the speaker thinks is growing too quickly. In the year 1202, the Italian mathematician Leonardo of Pisa, also known as Fibonacci, developed a formula to quantify this notion, albeit with some not terribly realistic assumptions.[28]

Suppose a newly born pair of rabbits, one male and one female, are put in a pen (or worse, released in the wild). Suppose further that the rabbits are able to mate at the age of one month (which, astonishingly, some breeds can) and have a one-month gestation period (which, astonishingly, some breeds do). Finally, suppose that these mythical rabbits never die, and that the female always produces one new pair (one male, one female) every month from its second month on. How many female rabbits will there be at the end of six months?

On the last day of the first month (call it month 0), there will be one female (ready to conceive on the first day of the next month). On the last day of the second month, there will still be only one female (since she will not give birth until the first day of the next month). On the last day of the next month, there will be two females (one pregnant and one not). On the last day of the next month, there will be three females (two pregnant and one not). And so on. Let's look at this progression in tabular form, Figure 4.6.

Notice that for month n > 1, females(n) = females(n-1) + females(n-2). This is not an accident. Each female that was alive in month n-1 will still be alive in month n. In addition, each female that was alive in month n-2 will produce one new female in month n. The new females can be added to the females alive in month n-1 to get the number of females in month n.

[28] That we call this a Fibonacci sequence is an example of a Eurocentric interpretation of history. Fibonacci's great contribution to European mathematics was his book *Liber Abaci*, which introduced to European mathematicians many concepts already well known to Indian and Arabic scholars. These concepts included Hindu-Arabic numerals and the decimal system. What we today call the Fibonacci sequence was taken from the work of the Sanskrit mathematician Pingala.

Month	Females
0	1
1	1
2	2
3	3
4	5
5	8
6	13

Figure 4.6 Growth in population of female rabbits

The growth in population is described naturally by the **recurrence**[29]

```
females(0) = 1
females(1) = 1
females(n + 2) = females(n+1) + females(n)
```

This definition is a little different from the recursive definition of factorial:

- It has two base cases, not just one. In general, you can have as many base cases as you need.
- In the recursive case, there are two recursive calls, not just one. Again, there can be as many as you need.

Figure 4.7 contains a straightforward implementation of the Fibonacci recurrence,[30] along with a function that can be used to test it.

```
def fib(n):
   """Assumes n int >= 0
      Returns Fibonacci of n"""
    if n == 0 or n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)

def testFib(n):
    for i in range(n+1):
        print('fib of', i, '=', fib(i))
```

Figure 4.7 Recursive implementation of Fibonacci sequence

[29] This version of the Fibonacci sequence corresponds to the definition used in Fibonacci's *Liber Abaci*. Other definitions of the sequence start with 0 rather than 1.

[30] While obviously correct, this is a terribly inefficient implementation of the Fibonacci function. There is a simple iterative implementation that is much better.

Writing the code is the easy part of solving this problem. Once we went from the vague statement of a problem about bunnies to a set of recursive equations, the code almost wrote itself. Finding some kind of abstract way to express a solution to the problem at hand is very often the hardest step in building a useful program. We will talk much more about this later in the book.

As you might guess, this is not a perfect model for the growth of rabbit populations in the wild. In 1859, Thomas Austin, an Australian farmer, imported twenty-four rabbits from England, to be used as targets in hunts. Ten years later, approximately two million rabbits were shot or trapped each year in Australia, with no noticeable impact on the population. That's a lot of rabbits, but not anywhere close to the 120th Fibonacci number.[31]

Though the Fibonacci sequence does not actually provide a perfect model of the growth of rabbit populations, it does have many interesting mathematical properties. Fibonacci numbers are also quite common in nature.

Finger exercise: When the implementation of `fib` in Figure 4.7 is used to compute `fib(5)`, how many times does it compute the value of `fib(2)` on the way to computing `fib(5)`?

4.3.2 Palindromes

Recursion is also useful for many problems that do not involve numbers. Figure 4.8 contains a function, `isPalindrome`, that checks whether a string reads the same way backwards and forwards.

The function `isPalindrome` contains two internal **helper functions**. This should be of no interest to clients of the function, who should care only that `isPalindrome` meets its specification. But you should care, because there are things to learn by examining the implementation.

The helper function `toChars` converts all letters to lowercase and removes all non-letters. It starts by using a built-in method on strings to generate a string that is identical to `s`, except that all uppercase letters have been converted to lowercase. We will talk a lot more about **method invocation** when we get to classes. For now, think of it as a peculiar syntax for a function call. Instead of putting the first (and in this case only) argument inside parentheses following the function name, we use **dot notation** to place that argument before the function name.

[31] The damage done by the descendants of those twenty-four cute bunnies has been estimated to be $600 million per year, and they are in the process of eating many native plants into extinction.

```
def isPalindrome(s):
   """Assumes s is a str
      Returns True if letters in s form a palindrome; False
        otherwise. Non-letters and capitalization are ignored."""

   def toChars(s):
      s = s.lower()
      letters = ''
      for c in s:
        if c in 'abcdefghijklmnopqrstuvwxyz':
            letters = letters + c
      return letters

   def isPal(s):
      if len(s) <= 1:
        return True
      else:
        return s[0] == s[-1] and isPal(s[1:-1])

   return isPal(toChars(s))
```

Figure 4.8 Palindrome testing

The helper function isPal uses recursion to do the real work. The two base cases are strings of length zero or one. This means that the recursive part of the implementation is reached only on strings of length two or more. The conjunction[32] in the else clause is evaluated from left to right. The code first checks whether the first and last characters are the same, and if they are goes on to check whether the string minus those two characters is a palindrome. That the second conjunct is not evaluated unless the first conjunct evaluates to True is semantically irrelevant in this example. However, later in the book we will see examples where this kind of **short-circuit evaluation** of Boolean expressions is semantically relevant.

This implementation of isPalindrome is an example of an important problem-solving principle known as **divide-and-conquer**. (This principle is related to but slightly different from divide-and-conquer algorithms, which are discussed in Chapter 10.) The problem-solving principle is to conquer a hard problem by breaking it into a set of subproblems with the properties that

- the subproblems are easier to solve than the original problem, and
- solutions of the subproblems can be combined to solve the original problem.

[32] When two Boolean-valued expressions are connected by "and," each expression is called a **conjunct**. If they are connected by "or," they are called **disjuncts**.

Divide-and-conquer is a very old idea. Julius Caesar practiced what the Romans referred to as *divide et impera* (divide and rule). The British practiced it brilliantly to control the Indian subcontinent. Benjamin Franklin was well aware of the British expertise in using this technique, prompting him to say at the signing of the U.S. Declaration of Independence, "We must all hang together, or assuredly we shall all hang separately."

In this case, we solve the problem by breaking the original problem into a simpler version of the same problem (checking whether a shorter string is a palindrome) and a simple thing we know how to do (comparing single characters), and then combine the solutions with `and`. Figure 4.9 contains some code that can be used to visualize how this works.

```
def isPalindrome(s):
   """Assumes s is a str
      Returns True if s is a palindrome; False otherwise.
       Punctuation marks, blanks, and capitalization are ignored."""

   def toChars(s):
      s = s.lower()
      letters = ''
      for c in s:
        if c in 'abcdefghijklmnopqrstuvwxyz':
            letters = letters + c
      return letters

   def isPal(s):
      print('  isPal called with', s)
      if len(s) <= 1:
         print('  About to return True from base case')
         return True
      else:
         answer = s[0] == s[-1] and isPal(s[1:-1])
         print('  About to return', answer, 'for', s)
         return answer

   return isPal(toChars(s))

def testIsPalindrome():
   print('Try dogGod')
   print(isPalindrome('dogGod'))
   print('Try doGood')
   print(isPalindrome('doGood'))
```

Figure 4.9 Code to visualize palindrome testing

When testIsPalindrome is run, it will print

```
Try dogGod
  isPal called with doggod
  isPal called with oggo
  isPal called with gg
  isPal called with
  About to return True from base case
  About to return True for gg
  About to return True for oggo
  About to return True for doggod
True
Try doGood
  isPal called with dogood
  isPal called with ogoo
  isPal called with go
  About to return False for go
  About to return False for ogoo
  About to return False for dogood
False
```

4.4 Global Variables

If you tried calling fib with a large number, you probably noticed that it took a very long time to run. Suppose we want to know how many recursive calls are made? We could do a careful analysis of the code and figure it out, and in Chapter 9 we will talk about how to do that. Another approach is to add some code that counts the number of calls. One way to do that uses **global variables**.

Until now, all of the functions we have written communicate with their environment solely through their parameters and return values. For the most part, this is exactly as it should be. It typically leads to programs that are relatively easy to read, test, and debug. Every once in a while, however, global variables come in handy. Consider the code in Figure 4.10.

In each function, the line of code global numFibCalls tells Python that the name numCalls should be defined at the outermost scope of the module (see Section 4.5) in which the line of code appears rather than within the scope of the function in which the line of code appears. Had we not included the code global numFibCalls, the name numFibCalls would have been local to each of the functions fib and testFib, because numFibCalls occurs on the left-hand side of an assignment statement in both fib and testFib. The functions fib and testFib both have unfettered access to the object referenced by the variable numFibCalls. The function `testFib` binds `numFibCalls` to `0` each time it calls `fib`, and `fib` increments the value of `numFibCalls` each time `fib` is entered.

```
def fib(x):
    """Assumes x an int >= 0
       Returns Fibonacci of x"""
    global numFibCalls
    numFibCalls += 1
    if x == 0 or x == 1:
        return 1
    else:
        return fib(x-1) + fib(x-2)

def testFib(n):
    for i in range(n+1):
        global numFibCalls
        numFibCalls = 0
        print('fib of', i, '=', fib(i))
        print('fib called', numFibCalls, 'times.')
```

Figure 4.10 Using a global variable

The call `testFib(6)` produces the output

```
fib of 0 = 1
fib called 1 times.
fib of 1 = 1
fib called 1 times.
fib of 2 = 2
fib called 3 times.
fib of 3 = 3
fib called 5 times.
fib of 4 = 5
fib called 9 times.
fib of 5 = 8
fib called 15 times.
fib of 6 = 13
fib called 25 times.
```

It is with some trepidation that we introduce the topic of global variables. Since the 1970s card-carrying computer scientists have inveighed against them. The indiscriminate use of global variables can lead to lots of problems. The key to making programs readable is locality. One reads a program a piece at a time, and the less context needed to understand each piece, the better. Since global variables can be modified or read in a wide variety of places, the sloppy use of them can destroy locality. Nevertheless, there are times when they are just what is needed.

4.5 Modules

So far, we have operated under the assumption that our entire program is stored in one file. This is perfectly reasonable as long as programs are small. As programs get larger, however, it is typically more convenient to store different parts of them in different files. Imagine, for example, that multiple people are working on the same program. It would be a nightmare if they were all trying to update the same file. Python modules allow us to easily construct a program from code in multiple files.

A **module** is a `.py` file containing Python definitions and statements. We could create, for example, a file `circle.py` containing the code in Figure 4.11.

```
pi = 3.14159

def area(radius):
    return pi*(radius**2)

def circumference(radius):
    return 2*pi*radius

def sphereSurface(radius):
    return 4.0*area(radius)

def sphereVolume(radius):
    return (4.0/3.0)*pi*(radius**3)
```

Figure 4.11 Some code related to circles and spheres

A program gets access to a module through an `import` statement. So, for example, the code

```
import circle
pi = 3
print(pi)
print(circle.pi)
print(circle.area(3))
print(circle.circumference(3))
print(circle.sphereSurface(3))
```

will print

```
3
3.14159
28.27431
18.849539999999998
113.09724
```

Modules are typically stored in individual files. Each module has its own private symbol table. Consequently, within `circle.py` we access objects (e.g., `pi` and `area`) in the usual way. Executing `import M` creates a binding for module `M` in the scope in which the `import` appears. Therefore, in the importing context we use dot notation to indicate that we are referring to a name defined in the imported module.[33] For example, outside of `circle.py`, the references `pi` and `circle.pi` can (and in this case do) refer to different objects.

At first glance, the use of dot notation may seem cumbersome. On the other hand, when one imports a module one often has no idea what local names might have been used in the implementation of that module. The use of dot notation to fully qualify names avoids the possibility of getting burned by an accidental name clash. For example, executing the assignment `pi = 3` outside of the `circle` module does not change the value of `pi` used within the `circle` module.

There is a variant of the `import` statement that allows the importing program to omit the module name when accessing names defined inside the imported module. Executing the statement `from M import *` creates bindings in the current scope to all objects defined within `M`, but not to `M` itself. For example, the code

```
from circle import *
print(pi)
print(circle.pi)
```

will first print `3.14159`, and then produce the error message

```
NameError: name 'circle' is not defined
```

Some Python programmers frown upon using this form of `import` because they believe that it makes code more difficult to read.

As we have seen, a module can contain executable statements as well as function definitions. Typically, these statements are used to initialize the module. For this reason, the statements in a module are executed only the first time a module is imported into a program. Moreover, a module is imported only once per interpreter session. If you start up a shell, import a module, and then change the contents of that module, the interpreter will still be using the original version of the

[33] Superficially, this may seem unrelated to the use of dot notation in method invocation. However, as we will see in Chapter 8, there is a deep connection.

module. This can lead to puzzling behavior when debugging. When in doubt, start a new shell.

There are lots of useful modules that come as part of the standard Python library. For example, it is rarely necessary to write your own implementations of common mathematical or string functions. A description of this library can be found at `http://docs.python.org/2/library/`.

4.6 Files

Every computer system uses **files** to save things from one computation to the next. Python provides many facilities for creating and accessing files. Here we illustrate some of the basic ones.

Each operating system (e.g., Windows and MAC OS) comes with its own file system for creating and accessing files. Python achieves operating-system independence by accessing files through something called a **file handle**. The code

```
nameHandle = open('kids', 'w')
```

instructs the operating system to create a file with the name `kids`, and return a file handle for that file. The argument `'w'` to `open` indicates that the file is to be opened for **writing**. The following code opens a file, uses the **`write`** method to write two lines, and then closes the file. It is important to remember to close the file when the program is finished using it. Otherwise there is a risk that some or all of the writes may not be saved.

```
nameHandle = open('kids', 'w')
for i in range(2):
    name = input('Enter name: ')
    nameHandle.write(name + '\n')
nameHandle.close()
```

In a Python string, the escape character “\” is used to indicate that the next character should be treated in a special way. In this example, the string `'\n'` indicates a newline character.

We can now open the file for **reading** (using the argument `'r'`), and print its contents. Since Python treats a file as a sequence of lines, we can use a `for` statement to iterate over the file’s contents.

```
nameHandle = open('kids', 'r')
for line in nameHandle:
    print(line)
nameHandle.close()
```

If we had typed in the names David and Andrea, this will print

```
David

Andrea
```

The extra line between David and Andrea is there because print starts a new line each time it encounters the `'\n'` at the end of each line in the file. We could have avoided printing that by writing `print line[:-1]`. The code

```
nameHandle = open('kids', 'w')
nameHandle.write('Michael\n')
nameHandle.write('Mark\n')
nameHandle.close()
nameHandle = open('kids', 'r')
for line in nameHandle:
    print(line[:-1])
nameHandle.close()
```

will print

```
Michael
Mark
```

Notice that we have overwritten the previous contents of the file `kids`. If we don't want to do that we can open the file for **appending** (instead of writing) by using the argument `'a'`. For example, if we now run the code

```
nameHandle = open('kids', 'a')
nameHandle.write('David\n')
nameHandle.write('Andrea\n')
nameHandle.close()
nameHandle = open('kids', 'r')
for line in nameHandle:
    print(line[:-1])
nameHandle.close()
```

it will print

```
Michael
Mark
David
Andrea
```

Some of the common operations on files are summarized in Figure 4.12.

`open(fn, 'w')` `fn` is a string representing a file name. Creates a file for writing and returns a file handle.

`open(fn, 'r')` `fn` is a string representing a file name. Opens an existing file for reading and returns a file handle.

`open(fn, 'a')` `fn` is a string representing a file name. Opens an existing file for appending and returns a file handle.

`fh.read()` returns a string containing the contents of the file associated with the file handle `fh`.

`fh.readline()` returns the next line in the file associated with the file handle `fh`.

`fh.readlines()` returns a list each element of which is one line of the file associated with the file handle `fh`.

`fh.write(s)` writes the string `s` to the end of the file associated with the file handle `fh`.

`fh.writeLines(S)` `S` is a sequence of strings. Writes each element of `S` as a separate line to the file associated with the file handle `fh`.

`fh.close()` closes the file associated with the file handle `fh`.

Figure 4.12 Common functions for accessing files

5 STRUCTURED TYPES, MUTABILITY, AND HIGHER-ORDER FUNCTIONS

The programs we have looked at thus far have dealt with three types of objects: `int`, `float`, and `str`. The numeric types `int` and `float` are scalar types. That is to say, objects of these types have no accessible internal structure. In contrast, `str` can be thought of as a structured, or non-scalar, type. One can use indexing to extract individual characters from a string and slicing to extract substrings.

In this chapter, we introduce four additional structured types. One, `tuple`, is a rather simple generalization of `str`. The other three—`list`, `range`,[34] and `dict`—are more interesting. We also return to the topic of functions with some examples that illustrate the utility of being able to treat functions in the same way as other types of objects.

5.1 Tuples

Like strings, **tuples** are immutable ordered sequences of elements. The difference is that the elements of a tuple need not be characters. The individual elements can be of any type, and need not be of the same type as each other.

Literals of type `tuple` are written by enclosing a comma-separated list of elements within parentheses. For example, we can write

```
t1 = ()
t2 = (1, 'two', 3)
print(t1)
print(t2)
```

Unsurprisingly, the `print` statements produce the output

```
()
(1, 'two', 3)
```

Looking at this example, you might naturally be led to believe that the tuple containing the single value `1` would be written `(1)`. But, to quote Richard Nixon, “that would be wrong.” Since parentheses are used to group expressions, `(1)` is

[34] Type `range` does not exist in Python 2.

merely a verbose way to write the integer 1. To denote the singleton tuple containing this value, we write (1,). Almost everybody who uses Python has at one time or another accidentally omitted that annoying comma.

Repetition can be used on tuples. For example, the expression 3*('a', 2) evaluates to ('a', 2, 'a', 2, 'a', 2).

Like strings, tuples can be concatenated, indexed, and sliced. Consider

```
t1 = (1, 'two', 3)
t2 = (t1, 3.25)
print(t2)
print((t1 + t2))
print((t1 + t2)[3])
print((t1 + t2)[2:5])
```

The second assignment statement binds the name t2 to a tuple that contains the tuple to which t1 is bound and the floating point number 3.25. This is possible because a tuple, like everything else in Python, is an object, so tuples can contain tuples. Therefore, the first print statement produces the output,

```
((1, 'two', 3), 3.25)
```

The second print statement prints the value generated by concatenating the values bound to t1 and t2, which is a tuple with five elements. It produces the output

```
(1, 'two', 3, (1, 'two', 3), 3.25)
```

The next statement selects and prints the fourth element of the concatenated tuple (as always in Python, indexing starts at 0), and the statement after that creates and prints a slice of that tuple, producing the output

```
(1, 'two', 3)
(3, (1, 'two', 3), 3.25)
```

A for statement can be used to iterate over the elements of a tuple:

```
def intersect(t1, t2):
    """Assumes t1 and t2 are tuples
       Returns a tuple containing elements that are in
          both t1 and t2"""
    result = ()
    for e in t1:
        if e in t2:
            result += (e,)
    return result
```

5.1.1 Sequences and Multiple Assignment

If you know the length of a sequence (e.g., a tuple or a string), it can be convenient to use Python's multiple assignment statement to extract the individual elements. For example, after executing the statement `x, y = (3, 4)`, `x` will be bound to `3` and `y` to `4`. Similarly, the statement `a, b, c = 'xyz'` will bind `a` to `'x'`, `b` to `'y'`, and `c` to `'z'`.

This mechanism is particularly convenient when used in conjunction with functions that return fixed-size sequences. Consider, for example the function definition

```
def findExtremeDivisors(n1, n2):
    """Assumes that n1 and n2 are positive ints
       Returns a tuple containing the smallest common divisor > 1 and
         the largest common divisor of n1 and n2. If no common divisor,
         returns (None, None)"""
    minVal, maxVal = None, None
    for i in range(2, min(n1, n2) + 1):
        if n1%i == 0 and n2%i == 0:
            if minVal == None:
                minVal = i
            maxVal = i
    return (minVal, maxVal)
```

The multiple assignment statement

```
minDivisor, maxDivisor = findExtremeDivisors(100, 200)
```

will bind `minDivisor` to `2` and `maxDivisor` to `100`.

5.2 Ranges

Like strings and tuples, ranges are immutable. The `range` function returns an object of type `range`. As stated in Section 3.2, the `range` function takes three integer arguments: `start`, `stop`, and `step`, and returns the progression of integers `start`, `start + step`, `start + 2*step`, etc. If `step` is positive, the last element is the largest integer `start + i*step` less than `stop`. If `step` is negative, the last element is the smallest integer `start + i*step` greater than `stop`. If only two arguments are supplied, a step of 1 is used. If only one argument is supplied, that argument is the stop, start defaults to 0, and step defaults to 1.

All of the operations on tuples are also available for ranges, except for concatenation and repetition. For example, `range(10)[2:6][2]` evaluates to `4`. When the `==` operator is used to compare objects of type `range`, it returns `True` if the two ranges represent the same sequence of integers. For example, `range(0, 7, 2) == range(0, 8, 2)` evaluates to `True`. However, `range(0, 7, 2) == range(6, -1, -2)` evaluates to `False` because though the two ranges contain the same integers, they occur in a different order.

Unlike objects of type `tuple`, the amount of space occupied by an object of type `range` is not proportional to its length. Because a range is fully defined by its start, stop, and step values; it can be stored in a small amount of space.

The most common use of `range` is in `for` loops, but objects of type `range` can be used anywhere a sequence of integers can be used.

5.3 Lists and Mutability

Like a tuple, a **list** is an ordered sequence of values, where each value is identified by an index. The syntax for expressing literals of type `list` is similar to that used for tuples; the difference is that we use square brackets rather than parentheses. The empty list is written as `[]`, and singleton lists are written without that (oh so easy to forget) comma before the closing bracket. So, for example, the code,

```
L = ['I did it all', 4, 'love']
for i in range(len(L)):
    print(L[i])
```

produces the output,

```
I did it all
4
love
```

Occasionally, the fact that square brackets are used for literals of type `list`, indexing into lists, and slicing lists can lead to some visual confusion. For example, the expression `[1,2,3,4][1:3][1]`, which evaluates to `3`, uses the square brackets in three different ways. This is rarely a problem in practice, because most of the time lists are built incrementally rather than written as literals.

Lists differ from tuples in one hugely important way: lists are **mutable**. In contrast, tuples and strings are **immutable**. There are many operators that can be used to create objects of these immutable types, and variables can be bound to objects of these types. But objects of immutable types cannot be modified. On the other hand, objects of type `list` can be modified after they are created.

The distinction between mutating an object and assigning an object to a variable may, at first, appear subtle. However, if you keep repeating the mantra, “In Python a variable is merely a name, i.e., a label that can be attached to an object,” it will bring you clarity.

When the statements

```
Techs = ['MIT', 'Caltech']
Ivys = ['Harvard', 'Yale', 'Brown']
```

are executed, the interpreter creates two new lists and binds the appropriate variables to them, as pictured in Figure 5.1.

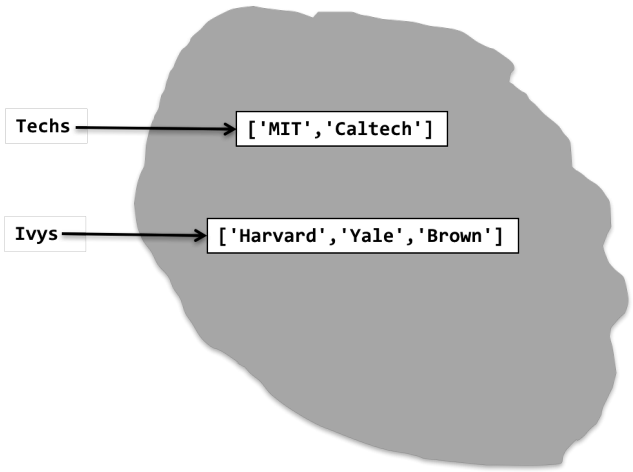


Figure 5.1 Two lists

The assignment statements

```
Univs = [Techs, Ivys]
Univs1 = [['MIT', 'Caltech'], ['Harvard', 'Yale', 'Brown']]
```

also create new lists and bind variables to them. The elements of these lists are themselves lists. The three print statements

```
print('Univs =', Univs)
print('Univs1 =', Univs1)
print(Univs == Univs1)
```

produce the output

```
Univs = [['MIT', 'Caltech'], ['Harvard', 'Yale', 'Brown']]
Univs1 = [['MIT', 'Caltech'], ['Harvard', 'Yale', 'Brown']]
True
```

It appears as if `Univs` and `Univs1` are bound to the same value. But appearances can be deceiving. As Figure 5.2 illustrates, `Univs` and `Univs1` are bound to quite different values.

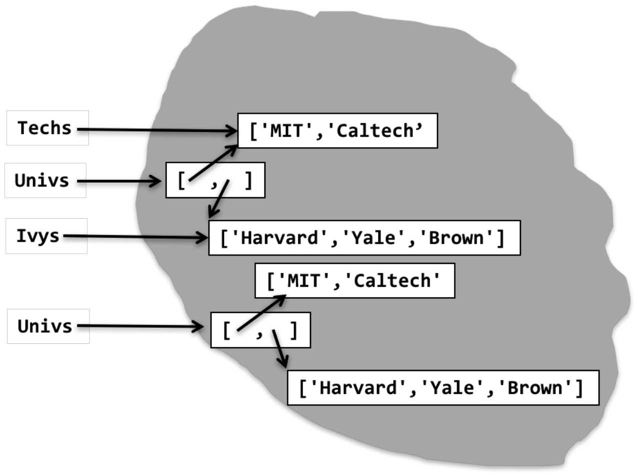


Figure 5.2 Two lists that appear to have the same value, but don't

That Univs and Univs1 are bound to different objects can be verified using the built-in Python function **id**, which returns a unique integer identifier for an object. This function allows us to test for **object equality**. When we run the code

```
print(Univs == Univs1) #test value equality
print(id(Univs) == id(Univs1)) #test object equality
print('Id of Univs =', id(Univs))
print('Id of Univs1 =', id(Univs1))
```

it prints

```
True
False
Id of Univs = 4447805768
Id of Univs1 = 4456134408
```

(Don't expect to see the same unique identifiers if you run this code. The semantics of Python says nothing about what identifier is associated with each object; it merely requires that no two objects have the same identifier.)

Notice that in Figure 5.2 the elements of Univs are not copies of the lists to which Techs and Ivys are bound, but are rather the lists themselves. The elements of Univs1 are lists that contain the same elements as the lists in Univs, but they are not the same lists. We can see this by running the code

```
print('Ids of Univs[0] and Univs[1]', id(Univs[0]), id(Univs[1]))
print('Ids of Univs1[0] and Univs1[1]', id(Univs1[0]), id(Univs1[1]))
```

which prints

```
Ids of Univs[0] and Univs[1] 4447807688 4456134664
Ids of Univs1[0] and Univs1[1] 4447805768 4447806728
```

Why does this matter? It matters because lists are mutable. Consider the code

```
Techs.append('RPI')
```

The `append` method has a **side effect**. Rather than create a new list, it mutates the existing list `Techs` by adding a new element, the string `'RPI'`, to the end of it. Figure 5.3 depicts the state of the computation after `append` is executed.

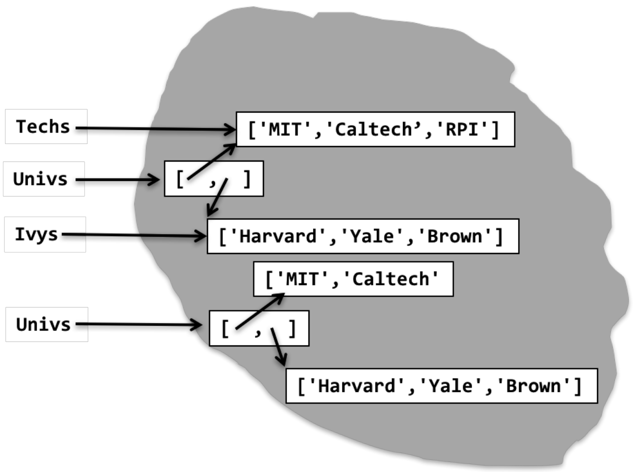


Figure 5.3 Demonstration of mutability

The object to which `Univs` is bound still contains the same two lists, but the contents of one of those lists has been changed. Consequently, the `print` statements

```
print('Univs =', Univs)
print('Univs1 =', Univs1)
```

now produce the output

```
Univs = [['MIT', 'Caltech', 'RPI'], ['Harvard', 'Yale', 'Brown']]
Univs1 = [['MIT', 'Caltech'], ['Harvard', 'Yale', 'Brown']]
```

What we have here is something called **aliasing**. There are two distinct paths to the same list object. One path is through the variable `Techs` and the other through the first element of the `list` object to which `Univs` is bound. One can mutate the object via either path, and the effect of the mutation will be visible through both paths. This can be convenient, but it can also be treacherous. Unintentional aliasing leads to programming errors that are often enormously hard to track down.

As with tuples, a for statement can be used to iterate over the elements of a list. For example,

```
for e in Univs:
    print('Univs contains', e)
    print('  which contains')
    for u in e:
        print('    ', u)
```

will print

```
Univs contains ['MIT', 'Caltech', 'RPI']
  which contains
     MIT
     Caltech
     RPI
Univs contains ['Harvard', 'Yale', 'Brown']
  which contains
     Harvard
     Yale
     Brown
```

When we append one list to another, e.g., Techs.append(Ivys), the original structure is maintained. I.e., the result is a list that contains a list. Suppose we do not want to maintain this structure, but want to add the elements of one list into another list. We can do that by using list concatenation or the extend method, e.g.,

```
L1 = [1,2,3]
L2 = [4,5,6]
L3 = L1 + L2
print('L3 =', L3)
L1.extend(L2)
print('L1 =', L1)
L1.append(L2)
print('L1 =', L1)
```

will print

```
L3 = [1, 2, 3, 4, 5, 6]
L1 = [1, 2, 3, 4, 5, 6]
L1 = [1, 2, 3, 4, 5, 6, [4, 5, 6]]
```

Notice that the operator + does not have a side effect. It creates a new list and returns it. In contrast, extend and append each mutated L1.

Figure 5.4 contains short descriptions of some of the methods associated with lists. Note that all of these except count and index mutate the list.

`L.append(e)` adds the object `e` to the end of `L`.

`L.count(e)` returns the number of times that e occurs in `L`.

`L.insert(i, e)` inserts the object `e` into `L` at index `i`.

`L.extend(L1)` adds the items in list `L1` to the end of `L`.

`L.remove(e)` deletes the first occurrence of `e` from `L`.

`L.index(e)` returns the index of the first occurrence of `e` in `L`, raises an exception (see Chapter 7) if e is not in `L`.

`L.pop(i)` removes and returns the item at index `i in` `L`, raises `an exception` if L is empty. If `i` is omitted, it defaults to `-1`, to remove and return the last element of `L`.

`L.sort()` sorts the elements of `L` in ascending order.

`L.reverse()` reverses the order of the elements in `L`.

Figure 5.4 Methods associated with lists

5.3.1 Cloning

It is usually prudent to avoid mutating a list over which one is iterating. Consider, for example, the code

```
def removeDups(L1, L2):
    """Assumes that L1 and L2 are lists.
       Removes any element from L1 that also occurs in L2"""
    for e1 in L1:
        if e1 in L2:
            L1.remove(e1)
L1 = [1,2,3,4]
L2 = [1,2,5,6]
removeDups(L1, L2)
print('L1 =', L1)
```

You might be surprised to discover that this prints

```
L1 = [2, 3, 4]
```

During a `for` loop, the implementation of Python keeps track of where it is in the list using an internal counter that is incremented at the end of each iteration. When the value of the counter reaches the current length of the list, the loop terminates. This works as one might expect if the list is not mutated within the loop, but can have surprising consequences if the list is mutated. In this case, the hidden counter starts out at `0`, discovers that `L1[0]` is in `L2`, and removes it—reducing the length of `L1` to 3. The counter is then incremented to `1`, and the code proceeds to check if the value of `L1[1]` is in `L2`. Notice that this is not the original value of `L1[1]` (i.e., `2`), but rather the current value of `L1[1]` (i.e., `3`). As you can see, it is possible to figure out what happens when the list is modified within the loop. However, it is not easy. And what happens is likely to be unintentional, as in this example.

One way to avoid this kind of problem is to use slicing to **clone**[35] (i.e., make a copy of) the list and write `for e1 in L1[:]`. Notice that writing

```
newL1 = L1
for e1 in newL1:
```

would not solve the problem. It would not create a copy of `L1`, but would merely introduce a new name for the existing list.

Slicing is not the only way to clone lists in Python. The expression `list(L)` returns a copy of the list `L`. If the list to be copied contains mutable objects that you want to copy as well, import the standard library module `copy` and use the function `copy.deepcopy`.

5.3.2 List Comprehension

List comprehension provides a concise way to apply an operation to the values in a sequence. It creates a new list in which each element is the result of applying a given operation to a value from a sequence (e.g., the elements in another list). For example,

```
L = [x**2 for x in range(1,7)]
print(L)
```

will print the list

```
[1, 4, 9, 16, 25, 36]
```

The `for` clause in a list comprehension can be followed by one or more `if` and `for` statements that are applied to the values produced by the `for` clause. These additional clauses modify the sequence of values generated by the first `for` clause and produce a new sequence of values, to which the operation associated with the comprehension is applied. For example, the code

```
mixed = [1, 2, 'a', 3, 4.0]
print([x**2 for x in mixed if type(x) == int])
```

squares the integers in `mixed`, and then prints `[1, 4, 9]`.

[35] The cloning of animals, including humans, raises a host of technical, ethical, and spiritual conumdrums. Fortunately, the cloning of Python objects does not.

Some Python programmers use list comprehensions in marvelous and subtle ways. That is not always a great idea. Remember that somebody else may need to read your code, and "subtle" is not usually a desirable property.

5.4 Functions as Objects

In Python, functions are **first-class objects**. That means that they can be treated like objects of any other type, e.g., `int` or `list`. They have types, e.g., the expression `type(abs)` has the value `<type 'built-in_function_or_method'>`; they can appear in expressions, e.g., as the right-hand side of an assignment statement or as an argument to a function; they can be elements of lists; etc.

Using functions as arguments allows a style of coding called **higher-order programming**. It can be particularly convenient in conjunction with lists, as shown in Figure 5.5.

```
def applyToEach(L, f):
   """Assumes L is a list, f a function
      Mutates L by replacing each element, e, of L by f(e)"""
   for i in range(len(L)):
      L[i] = f(L[i])

L = [1, -2, 3.33]
print('L =', L)
print('Apply abs to each element of L.')
applyToEach(L, abs)
print('L =', L)
print('Apply int to each element of', L)
applyToEach(L, int)
print('L =', L)
print('Apply factorial to each element of', L)
applyToEach(L, factR)
print('L =', L)
print('Apply Fibonnaci to each element of', L)
applyToEach(L, fib)
print('L =', L)
```

Figure 5.5 Applying a function to elements of a list

The function `applyToEach` is called **higher-order** because it has an argument that is itself a function. The first time it is called, it mutates `L` by applying the unary built-in function `abs` to each element. The second time it is called, it applies a type conversion to each element. The third time it is called, it replaces each element by the result of applying the function `factR` (defined in Figure 4.5) to each element. And the fourth time it is called, it replaces each element by the result of applying the function `fib` (defined in Figure 4.7) to each element. It prints

```
L = [1, -2, 3.33]
Apply abs to each element of L.
L = [1, 2, 3.33]
Apply int to each element of [1, 2, 3.33]
L = [1, 2, 3]
Apply factorial to each element of [1, 2, 3]
L = [1, 2, 6]
Apply Fibonnaci to each element of [1, 2, 6]
L = [1, 2, 13]
```

Python has a built-in higher-order function, `map`, that is similar to, but more general than, the `applyToEach` function defined in Figure 5.5. it is designed to be used in conjunction with a `for` loop. In its simplest form the first argument to `map` is a unary function (i.e., a function that has only one parameter) and the second argument is any ordered collection of values suitable as arguments to the first argument.

When used in a `for` loop, `map` behaves like the `range` function in that it returns one value for each iteration of the loop.[36] These values are generated by applying the first argument to each element of the second argument. For example, the code

```
for i in map(fib, [2, 6, 4]):
    print(i)
```

prints

```
2
13
5
```

More generally, the first argument to `map` can be a function of n arguments, in which case it must be followed by n subsequent ordered collections (each of the same length). For example, the code

```
L1 = [1, 28, 36]
L2 = [2, 57, 9]
for i in map(min, L1, L2):
    print(i)
```

[36] In Python 2, `map` does not return values one at a time. Instead, it returns a list of values. That is to say, it behaves like the Python 2 `range` function rather than like the Python 2 `xrange` function.

prints

```
1
28
9
```

Python supports the creation of anonymous functions (i.e., functions that are not bound to a name), using the reserved word `lambda`. The general form of a **lambda expression** is

```
lambda <sequence of variable names>: <expression>
```

For example, the lambda expression `lambda x, y: x*y` returns a function that returns the product of its two arguments. Lambda expressions are frequently used as arguments to higher-order functions. For example, the code

```
L = []
for i in map(lambda x, y: x**y, [1 ,2 ,3, 4], [3, 2, 1, 0]):
    L.append(i)
print(L)
```

prints `[1, 4, 3, 1]`.

5.5 Strings, Tuples, Ranges, and Lists

We have looked at four different sequence types: `str`, `tuple`, `range`, and `list`. They are similar in that objects of of these types can be operated upon as described in Figure 5.6. Some of their other similarities and differences are summarized in Figure 5.7.

`seq[i]` returns the ith element in the sequence.

`len(seq)` returns the length of the sequence.

`seq1 + seq2` returns the concatenation of the two sequences (not available for ranges).

`n*seq` returns a sequence that repeats seq n times (not available for ranges).

`seq[start:end]` returns a slice of the sequence.

`e in seq` is `True` if `e` is contained in the sequence and `False` otherwise.

`e not in seq` is `True` if `e` is not in the sequence and `False` otherwise.

`for e in seq` iterates over the elements of the sequence.

Figure 5.6 Common operations on sequence types

Type	Type of elements	Examples of literals	Mutable
`str`	characters	`'', 'a', 'abc'`	No
`tuple`	any type	`(), (3,), ('abc', 4)`	No
`range`	integers	`range(10), range(1, 10, 2)`	No
`list`	any type	`[], [3], ['abc', 4]`	Yes

Figure 5.7 Comparison of sequence types

Python programmers tend to use lists far more often than tuples. Since lists are mutable, they can be constructed incrementally during a computation. For example, the following code incrementally builds a list containing all of the even numbers in another list.

```
evenElems = []
for e in L:
    if e%2 == 0:
        evenElems.append(e)
```

One advantage of tuples is that because they are immutable, aliasing is never a worry. Another advantage of their being immutable is that tuples, unlike lists, can be used as keys in dictionaries, as we will see in the next section.

Since strings can contain only characters, they are considerably less versatile than tuples or lists. On the other hand, when you are working with a string of characters there are many built-in methods that make life easy. Figure 5.8 contains short descriptions of a few of them. Keep in mind that since strings are immutable these all return values and have no side effect.

One of the more useful built-in methods is `split`, which takes two strings as arguments. The second argument specifies a separator that is used to split the first argument into a sequence of substrings. For example,

```
print('My favorite professor--John G.--rocks'.split(' '))
print('My favorite professor--John G.--rocks'.split('-'))
print('My favorite professor--John G.--rocks'.split('--'))
```

prints

```
['My', 'favorite', 'professor--John', 'G.--rocks']
['My favorite professor', '', 'John G.', '', 'rocks']
['My favorite professor', 'John G.', 'rocks']
```

The second argument is optional. If that argument is omitted the first string is split using arbitrary strings of **whitespace characters** (space, tab, newline, return, and formfeed).

`s.count(s1)` counts how many times the string `s1` occurs in `s`.

`s.find(s1)` returns the index of the first occurrence of the substring `s1` in `s`, and `-1` if `s1` is not in `s`.

`s.rfind(s1)` same as `find`, but starts from the end of `s` (the "r" in `rfind` stands for reverse).

`s.index(s1)` same as `find`, but raises an exception (Chapter 7) if `s1` is not in `s`.

`s.rindex(s1)` same as `index`, but starts from the end of `s`.

`s.lower()` converts all uppercase letters in `s` to lowercase.

`s.replace(old, new)` replaces all occurrences of the string `old` in `s` with the string `new`.

`s.rstrip()` removes trailing white space from `s`.

`s.split(d)` Splits `s` using `d` as a delimiter. Returns a list of substrings of `s`. For example, the value of `'David Guttag plays basketball'.split(' ')` is `['David', 'Guttag', 'plays', 'basketball']`. If `d` is omitted, the substrings are separated by arbitrary strings of whitespace characters.

Figure 5.8 Some methods on strings

5.6 Dictionaries

Objects of type **`dict`** (short for dictionary) are like lists except that we index them using **keys**. Think of a dictionary as a set of key/value pairs. Literals of type dict are enclosed in curly braces, and each element is written as a key followed by a colon followed by a value. For example, the code,

```
monthNumbers = {'Jan':1, 'Feb':2, 'Mar':3, 'Apr':4, 'May':5,
                1:'Jan', 2:'Feb', 3:'Mar', 4:'Apr', 5:'May'}
print('The third month is ' + monthNumbers[3])
dist = monthNumbers['Apr'] - monthNumbers['Jan']
print('Apr and Jan are', dist, 'months apart')
```

will print

```
The third month is Mar
Apr and Jan are 3 months apart
```

The entries in a `dict` are unordered and cannot be accessed with an index. That's why `monthNumbers[1]` unambiguously refers to the entry with the key `1` rather than the second entry.

Like lists, dictionaries are mutable. We can add an entry by writing

```
monthNumbers['June'] = 6
```

or change an entry by writing

```
monthNumbers['May'] = 'V'
```

Dictionaries are one of the great things about Python. They greatly reduce the difficulty of writing a variety of programs. For example, in Figure 5.9 we use dictionaries to write a (pretty horrible) program to translate between languages. Since one of the lines of code was too long to fit on the page, we used a backslash, \, to indicate that the next line of text is a continuation of the previous line.

The code in the figure prints,

```
Je bois "good" rouge vin, et mange pain.
I drink of wine red.
```

Remember that dictionaries are mutable. So one must be careful about side effects. For example,

```
FtoE['bois'] = 'wood'
Print(translate('Je bois du vin rouge.', dicts, 'French to English'))
```

will print

```
I wood of wine red
```

Most programming languages do not contain a built-in type that provides a mapping from keys to values. Instead, programmers use other types to provide similar functionality. It is, for example, relatively easy to implement a dictionary by using a list in which each element is a key/value pair. One can then write a simple function that does the associative retrieval, e.g.,

```
def keySearch(L, k):
    for elem in L:
        if elem[0] == k:
            return elem[1]
    return None
```

The problem with such an implementation is that it is computationally inefficient. In the worst case, a program might have to examine each element in the list to perform a single retrieval. In contrast, the built-in implementation is quite fast. It uses a technique called hashing, described in Chapter 10, to do the lookup in time that is nearly independent of the size of the dictionary.

```
EtoF = {'bread':'pain', 'wine':'vin', 'with':'avec', 'I':'Je',
        'eat':'mange', 'drink':'bois', 'John':'Jean',
        'friends':'amis', 'and': 'et', 'of':'du','red':'rouge'}
FtoE = {'pain':'bread', 'vin':'wine', 'avec':'with', 'Je':'I',
        'mange':'eat', 'bois':'drink', 'Jean':'John',
        'amis':'friends', 'et':'and', 'du':'of', 'rouge':'red'}
dicts = {'English to French':EtoF, 'French to English':FtoE}

def translateWord(word, dictionary):
    if word in dictionary.keys():
        return dictionary[word]
    elif word != '':
        return '"' + word + '"'
    return word

def translate(phrase, dicts, direction):
    UCLetters = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
    LCLetters = 'abcdefghijklmnopqrstuvwxyz'
    letters = UCLetters + LCLetters
    dictionary = dicts[direction]
    translation = ''
    word = ''
    for c in phrase:
        if c in letters:
            word = word + c
        else:
            translation = translation\
                          + translateWord(word, dictionary) + c
            word = ''
    return translation + ' ' + translateWord(word, dictionary)

print(translate('I drink good red wine, and eat bread.',
                dicts,'English to French'))
print(translate('Je bois du vin rouge.',
                dicts, 'French to English'))
```

Figure 5.9 Translating text (badly)

A `for` statement can be used to iterate over the entries in a dictionary. However, the value assigned to the iteration variable is a key, not a key/value pair. The order in which the keys are seen in the iteration is not defined. For example, the code

```
monthNumbers = {'Jan':1, 'Feb':2, 'Mar':3, 'Apr':4, 'May':5,
                1:'Jan', 2:'Feb', 3:'Mar', 4:'Apr', 5:'May'}
keys = []
for e in monthNumbers:
    keys.append(str(e))
print(keys)
keys.sort()
print(keys)
```

might print

```
['Jan', 'Mar', '2', '3', '4', '5', '1', 'Feb', 'May', 'Apr']
['1', '2', '3', '4', '5', 'Apr', 'Feb', 'Jan', 'Mar', 'May']
```

The method `keys` returns an object of type `dict_keys`.[37] This is an example of a **view object**. The order in which the keys appear in the view is not defined. A view object is dynamic in that if the object with which it is associated changes, the change is visible through the view object. For example,

```
birthStones = {'Jan':'Garnet', 'Feb':'Amethyst', 'Mar':'Acquamarine',
               'Apr':'Diamond', 'May':'Emerald'}
months = birthStones.keys()
print(months)
birthStones['June'] = 'Pearl'
print(months)
```

might print

```
dict_keys(['Jan', 'Feb', 'May', 'Apr', 'Mar'])
dict_keys(['Jan', 'Mar', 'June', 'Feb', 'May', 'Apr'])
```

Objects of type `dict_keys` can be iterated over using `for`, and membership can be tested using `in`. An object of type `dict_keys` can easily be converted into a list, e.g., `list(months)`.

Not all types of of objects can be used as keys: A key must be an object of a **hashable type**. A type is hashable if it has

- A `__hash__` method that maps an object of the type to an `int`, and for every object the value returned by `__hash__` does not change during the lifetime of the object, and
- An `__eq__` method that is used to compare objects for equality.

All of Python's built-in immutable types are hashable, and none of Python's built-in mutable types are hashable. It is often convenient to use tuples as keys. Imagine, for example, using a tuple of the form `(flightNumber, day)` to represent

[37] In Python 2, `keys` returns a list containing the keys of the dictionary.

airline flights. It would then be easy to use such tuples as keys in a dictionary implementing a mapping from flights to arrival times.

As with lists, there are many useful methods associated with dictionaries, including some for removing elements. We do not enumerate all of them here, but will use them as convenient in examples later in the book. Figure 5.10 contains some of the more useful operations on dictionaries.[38]

`len(d)` returns the number of items in `d`.

`d.keys()` returns a view of the keys in `d`.

`d.values()` returns a view of the values in `d`.

`k in d` returns `True` if key `k` is in `d`.

`d[k]` returns the item in `d` with key `k`.

`d.get(k, v)` returns `d[k]` if `k` is in `d`, and `v` otherwise.

`d[k] = v` associates the value `v` with the key `k in d`. If there is already a value associated with `k`, that value is replaced.

`del d[k]` removes the key `k` from `d`.

`for k in d` iterates over the keys in `d`.

Figure 5.10 Some common operations on dicts

[38] All of the methods that return a view in Python 3 return a list in Python 2.

6 TESTING AND DEBUGGING

We hate to bring this up, but Dr. Pangloss was wrong. We do not live in "the best of all possible worlds." There are some places where it rains too little, and others where it rains too much. Some places are too cold, some too hot, and some too hot in the summer and too cold in the winter. Sometimes the stock market goes down—a lot. And, annoyingly, our programs don't always function properly the first time we run them.

Books have been written about how to deal with this last problem, and there is a lot to be learned from reading these books. However, in the interest of providing you with some hints that might help you get that next problem set in on time, this chapter provides a highly condensed discussion of the topic. While all of the programming examples are in Python, the general principles are applicable to getting any complex system to work.

Testing is the process of running a program to try and ascertain whether or not it works as intended. **Debugging** is the process of trying to fix a program that you already know does not work as intended.

Testing and debugging are not processes that you should begin to think about after a program has been built. Good programmers design their programs in ways that make them easier to test and debug. The key to doing this is breaking the program up into separate components that can be implemented, tested, and debugged independently of other components. At this point in the book, we have discussed only one mechanism for modularizing programs, the function. So, for now, all of our examples will be based around functions. When we get to other mechanisms, in particular classes, we will return to some of the topics covered in this chapter.

The first step in getting a program to work is getting the language system to agree to run it—that is, eliminating syntax errors and static semantic errors that can be detected without running the program. If you haven't gotten past that point in your programming, you're not ready for this chapter. Spend a bit more time working on small programs, and then come back.

6.1 Testing

The most important thing to say about testing is that its purpose is to show that bugs exist, not to show that a program is bug-free. To quote Edsger Dijkstra, "Program testing can be used to show the presence of bugs, but never to show their absence!"[39] Or, as Albert Einstein reputedly said, "No amount of experimentation can ever prove me right; a single experiment can prove me wrong."

Why is this so? Even the simplest of programs has billions of possible inputs. Consider, for example, a program that purports to meet the specification

```
def isBigger(x, y):
    """Assumes x and y are ints
       Returns True if x is less than y and False otherwise."""
```

Running it on all pairs of integers would be, to say the least, tedious. The best we can do is to run it on pairs of integers that have a reasonable probability of producing the wrong answer if there is a bug in the program.

The key to testing is finding a collection of inputs, called a **test suite**, that has a high likelihood of revealing bugs, yet does not take too long to run. The key to doing this is partitioning the space of all possible inputs into subsets that provide equivalent information about the correctness of the program, and then constructing a test suite that contains at least one input from each partition. (Usually, constructing such a test suite is not actually possible. Think of this as an unachievable ideal.)

A **partition** of a set divides that set into a collection of subsets such that each element of the original set belongs to exactly one of the subsets. Consider, for example, `isBigger(x, y)`. The set of possible inputs is all pairwise combinations of integers. One way to partition this set is into these seven subsets:

x positive, y positive	x negative, y negative	
x positive, y negative	x negative, y positive	
x = 0, y = 0	x = 0, y ≠ 0	x ≠ 0, y = 0

If one tested the implementation on at least one value from each of these subsets, there would be reasonable probability (but no guarantee) of exposing a bug if one exists.

For most programs, finding a good partitioning of the inputs is far easier said than done. Typically, people rely on heuristics based on exploring different paths through some combination of the code and the specifications. Heuristics based

[39] "Notes On Structured Programming," Technical University Eindhoven, T.H. Report 70-WSK-03, April 1970.

on exploring paths through the code fall into a class called **glass-box testing**. Heuristics based on exploring paths through the specification fall into a class called **black-box testing**.

6.1.1 Black-Box Testing

In principle, black-box tests are constructed without looking at the code to be tested. Black-box testing allows testers and implementers to be drawn from separate populations. When those of us who teach programming courses generate test cases for the problem sets we assign students, we are developing black-box test suites. Developers of commercial software often have quality assurance groups that are largely independent of development groups.

This independence reduces the likelihood of generating test suites that exhibit mistakes that are correlated with mistakes in the code. Suppose, for example, that the author of a program made the implicit, but invalid, assumption that a function would never be called with a negative number. If the same person constructed the test suite for the program, he would likely repeat the mistake, and not test the function with a negative argument.

Another positive feature of black-box testing is that it is robust with respect to implementation changes. Since the test data is generated without knowledge of the implementation, the tests need not be changed when the implementation is changed.

As we said earlier, a good way to generate black-box test data is to explore paths through a specification. Consider, the specification

```
def sqrt(x, epsilon):
    """Assumes x, epsilon floats
               x >= 0
               epsilon > 0
       Returns result such that
               x-epsilon <= result*result <= x+epsilon"""
```

There seem to be only two distinct paths through this specification: one corresponding to `x = 0` and one corresponding to `x > 0`. However, common sense tells us that while it is necessary to test these two cases, it is hardly sufficient.

Boundary conditions should also be tested. When looking at lists, this often means looking at the empty list, a list with exactly one element, and a list containing lists. When dealing with numbers, it typically means looking at very small and very large values as well as "typical" values. For `sqrt`, for example, it might make sense to try values of `x` and `epsilon` similar to those in Figure 6.1.

The first four rows are intended to represent typical cases. Notice that the values for `x` include a perfect square, a number less than one, and a number with an irrational square root. If any of these tests fail, there is a bug in the program that needs to be fixed.

X	Epsilon
0.0	0.0001
25.0	0.0001
0.5	0.0001
2.0	0.0001
2.0	1.0/2.0**64.0
1.0/2.0**64	1.0/2.0**64.0
2.0**64.0	1.0/2.0**64.0
1.0/2.0**64.0	2.0**64.0
2.0**64.0	2.0**64.0

Figure 6.1 Testing boundary conditions

The remaining rows test extremely large and small values of `x` and `epsilon`. If any of these tests fail, something needs to be fixed. Perhaps there is a bug in the code that needs to be fixed, or perhaps the specification needs to be changed so that it is easier to meet. It might, for example, be unreasonable to expect to find an approximation of a square root when `epsilon` is ridiculously small.

Another important boundary condition to think about is aliasing. Consider, for example, the code

```
def copy(L1, L2):
    """Assumes L1, L2 are lists
       Mutates L2 to be a copy of L1"""
    while len(L2) > 0: #remove all elements from L2
        L2.pop() #remove last element of L2
    for e in L1: #append L1's elements to initially empty L2
        L2.append(e)
```

It will work most of the time, but not when `L1` and `L2` refer to the same list. Any test suite that did not include a call of the form `copy(L, L)`, would not reveal the bug.

6.1.2 Glass-box Testing

Black-box testing should never be skipped, but it is rarely sufficient. Without looking at the internal structure of the code, it is impossible to know which test cases are likely to provide new information. Consider the trivial example:

```
def isPrime(x):
    """Assumes x is a nonnegative int
       Returns True if x is prime; False otherwise"""
    if x <= 2:
        return False
    for i in range(2, x):
        if x%i == 0:
            return False
    return True
```

Looking at the code, we can see that because of the test `if x <= 2`, the values `0`, `1`, and `2` are treated as special cases, and therefore need to be tested. Without looking at the code, one might not test `isPrime(2),` and would therefore not discover that the function call `isPrime(2)` returns `False`, erroneously indicating that 2 is not a prime.

Glass-box test suites are usually much easier to construct than black-box test suites. Specifications are usually incomplete and often pretty sloppy, making it a challenge to estimate how thoroughly a black-box test suite explores the space of interesting inputs. In contrast, the notion of a path through code is well defined, and it is relatively easy to evaluate how thoroughly one is exploring the space. There are, in fact, commercial tools that can be used to objectively measure the completeness of glass-box tests.

A glass-box test suite is **path-complete** if it exercises every potential path through the program. This is typically impossible to achieve, because it depends upon the number of times each loop is executed and the depth of each recursion. For example, a recursive implementation of factorial follows a different path for each possible input (because the number of levels of recursion will differ).

Furthermore, even a path-complete test suite does not guarantee that all bugs will be exposed. Consider:

```
def abs(x):
    """Assumes x is an int
       Returns x if x>=0 and –x otherwise"""
    if x < -1:
        return -x
    else:
        return x
```

The specification suggests that there are two possible cases: `x` either is negative or it isn't. This suggests that the set of inputs `{2, -2}` is sufficient to explore all paths in the specification. This test suite has the additional nice property of forcing the program through all of its paths, so it looks like a complete glass-box suite as well. The only problem is that this test suite will not expose the fact that `abs(-1)` will return `-1`.

Despite the limitations of glass-box testing, there are a few rules of thumb that are usually worth following:

- Exercise both branches of all `if` statements.
- Make sure that each `except` clause (see Chapter 7) is executed.
- For each `for` loop, have test cases in which
    - The loop is not entered (e.g., if the loop is iterating over the elements of a list, make sure that it is tested on the empty list),
    - The body of the loop is executed exactly once, and
    - The body of the loop is executed more than once.
- For each `while` loop,
    - Look at the same kinds of cases as when dealing with `for` loops.
    - Include test cases corresponding to all possible ways of exiting the loop. For example, for a loop starting with

      ```
      while len(L) > 0 and not L[i] == e
      ```

      find cases where the loop exits because `len(L)` is greater than zero and cases where it exits because `L[i] == e`.
- For recursive functions, include test cases that cause the function to return with no recursive calls, exactly one recursive call, and more than one recursive call.

6.1.3 Conducting Tests

Testing is often thought of as occurring in two phases. One should always start with **unit testing**. During this phase testers construct and run tests designed to ascertain whether individual units of code (e.g., functions) work properly. This is followed by **integration testing**, which is designed to ascertain whether the program as a whole behaves as intended. In practice, testers cycle through these two phases, since failures during integration testing lead to making changes to individual units.

Integration testing is almost always more challenging than unit testing. One reason for this is that the intended behavior of an entire program is often considerably harder to characterize than the intended behavior of each of its parts. For example, characterizing the intended behavior of a word processor is considerably more challenging than characterizing the behavior of a function that counts the number of characters in a document. Problems of scale can also make integration testing difficult. It is not unusual for integration tests to take hours or even days to run.

Many industrial software development organizations have a **software quality assurance (SQA)** group that is separate from the group charged with implementing the software. The mission of this group is to ensure that before the software is released it is suitable for its intended purpose. In some organizations the development group is responsible for unit testing and the QA group for integration testing.

In industry, the testing process is often highly automated. Testers[40] do not sit at terminals typing inputs and checking outputs. Instead, they use **test drivers** that autonomously

- Set up the environment needed to invoke the program (or unit) to be tested,
- Invoke the program (or unit) to be tested with a predefined or automatically generated sequence of inputs,
- Save the results of these invocations,
- Check the acceptability of the results of the tests, and
- Prepare an appropriate report.

During unit testing, we often need to build **stubs** as well as drivers. Drivers simulate parts of the program that use the unit being tested, whereas stubs simulate parts of the program used by the unit being tested. Stubs are useful because they allow people to test units that depend upon software or sometimes even hardware that does not yet exist. This allows teams of programmers to simultaneously develop and test multiple parts of a system.

Ideally, a stub should

- Check the reasonableness of the environment and arguments supplied by the caller (calling a function with inappropriate arguments is a common error),
- Modify arguments and global variables in a manner consistent with the specification, and
- Return values consistent with the specification.

Building adequate stubs is often a challenge. If the unit the stub is replacing is intended to perform some complex task, building a stub that performs actions consistent with the specification may be tantamount to writing the program that the stub is designed to replace. One way to surmount this problem is to limit the

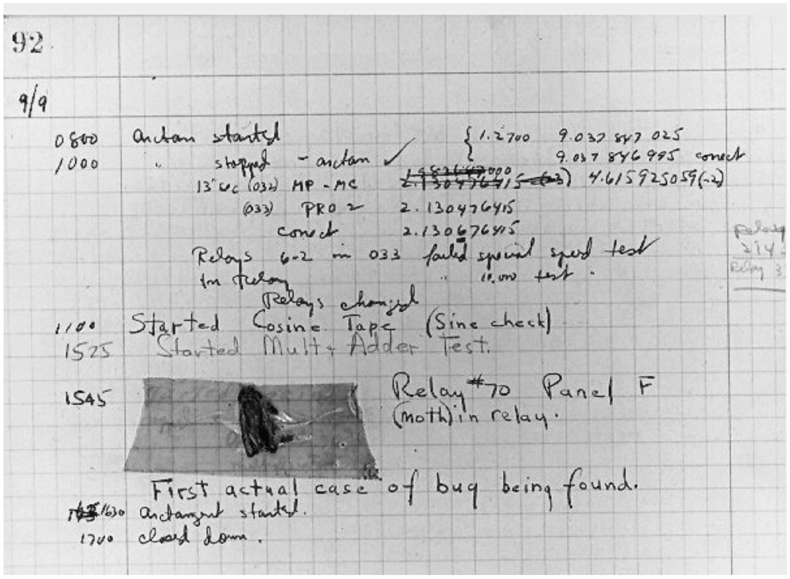
[40] Or, for that matter, those who grade problem sets in very large programming courses.

set of arguments accepted by the stub, and create a table that contains the values to be returned for each combination of arguments to be used in the test suite.

One attraction of automating the testing process is that it facilitates **regression testing**. As programmers attempt to debug a program, it is all too common to install a "fix" that breaks something that used to work. Whenever any change is made, no matter how small, you should check that the program still passes all of the tests that it used to pass.

6.2 Debugging

There is a charming urban legend about how the process of fixing flaws in software came to be known as debugging. The photo in Figure 6.2 is of a September 9, 1947, page in a laboratory book from the group working on the Mark II Aiken Relay Calculator at Harvard University.



92

9/9

0800 Anctan started {1.2700 9.037 847 025
1000 " stopped - anctan ✓ 9.037 846 995 conect
13" uc (032) MP - MC ~~1.982147000~~ ~~2.130476415~~ (-3)) 4.615925059(-2)
(033) PRO 2 2.130476415
conect 2.130676415
Relays 6-2 in 033 failed special speed test
in relay " 10,000 test.
Relays changed
1100 Started Cosine Tape (Sine check)
1525 Started Mult + Adder Test.
1545 Relay #70 Panel F
(moth) in relay.
First actual case of bug being found.
~~1[illegible]~~ 1630 Anctangent started.
1700 closed down.

Figure 6.2 Not the first bug

Some have claimed that the discovery of that unfortunate moth trapped in the Mark II led to the use of the phrase debugging. However the wording, "First actual case of a bug being found," suggests that a less literal interpretation of the phrase was already common. Grace Murray Hopper, a leader of the Mark II project, made it clear that the term "bug" was already in wide use to describe problems with electronic systems during World War II. And well prior to that, *Hawkins' New Catechism of Electricity*, an 1896 electrical handbook, included the entry, "The term 'bug' is used to a limited extent to designate any fault or trouble in the connections or working of electric apparatus." In English usage the word "bugbear" means "anything causing seemingly needless or excessive fear or anxiety."[41] Shakespeare seems to have shortened this to "bug," when he had Hamlet kvetch about "bugs and goblins in my life."

The use of the word "bug" sometimes leads people to ignore the fundamental fact that if you wrote a program and it has a "bug," you messed up. Bugs do not crawl unbidden into flawless programs. If your program has a bug, it is because you put it there. Bugs do not breed in programs. If your program has multiple bugs, it is because you made multiple mistakes. Runtime bugs can be categorized along two dimensions:

- **Overt → covert**: An **overt bug** has an obvious manifestation, e.g., the program crashes or takes far longer (maybe forever) to run than it should. A **covert bug** has no obvious manifestation. The program may run to conclusion with no problem—other than providing an incorrect answer. Many bugs fall between the two extremes, and whether or not the bug is overt can depend upon how carefully one examines the behavior of the program.
- **Persistent → intermittent**: A **persistent bug** occurs every time the program is run with the same inputs. An **intermittent bug** occurs only some of the time, even when the program is run on the same inputs and seemingly under the same conditions. When we get to Chapter 14, we will start writing programs that model situations in which randomness plays a role. In programs of that of the kind, intermittent bugs are common.

The best kinds of bugs to have are overt and persistent. Developers can be under no illusion about the advisability of deploying the program. And if someone else is foolish enough to attempt to use it, they will quickly discover their folly. Perhaps the program will do something horrible before crashing, e.g., delete files, but at least the user will have reason to be worried (if not panicked). Good programmers try to write their programs in such a way that programming mistakes lead to bugs that are both overt and persistent. This is often called **defensive programming**.

The next step into the pit of undesirability is bugs that are overt but intermittent. An air traffic control system that computes the correct location for planes almost all of the time would be far more dangerous than one that makes obvious

[41] Webster's New World College Dictionary.

mistakes all the time. One can live in a fool's paradise for a period of time, and maybe get so far as deploying a system incorporating the flawed program, but sooner or later the bug will become manifest. If the conditions prompting the bug to become manifest are easily reproducible, it is often relatively easy to track down and repair the problem. If the conditions provoking the bug are not clear, life is much harder.

Programs that fail in covert ways are often highly dangerous. Since they are not apparently problematical, people use them and trust them to do the right thing. Increasingly, society relies on software to perform critical computations that are beyond the ability of humans to carry out or even check for correctness. Therefore, a program can provide undetected fallacious answer for long periods of time. Such programs can, and have, caused a lot of damage.[42] A program that evaluates the risk of a mortgage bond portfolio and confidently spits out the wrong answer can get a bank (and perhaps all of society) into a lot of trouble. A radiation therapy machine that delivers a little more or a little less radiation than intended can be the difference between life and death for a person with cancer. A program that makes a covert error only occasionally may or may not wreak less havoc than one that always commits such an error. Bugs that are both covert and intermittent are almost always the hardest to find and fix.

6.2.1 Learning to Debug

Debugging is a learned skill. Nobody does it well instinctively. The good news is that it's not hard to learn, and it is a transferable skill. The same skills used to debug software can be used to find out what is wrong with other complex systems, e.g., laboratory experiments or sick humans.

For at least four decades people have been building tools called debuggers, and there are debugging tools built into all of the popular Python IDE's. These are supposed to help people find bugs in their programs. They can help, but only a little. What's much more important is how you approach the problem. Many experienced programmers don't even bother with debugging tools. Most programmers say that the most important debugging tool is the `print` statement.

Debugging starts when testing has demonstrated that the program behaves in undesirable ways. Debugging is the process of searching for an explanation of

[42] On August 1, 2012, Knight Capital Group, Inc. deployed a new piece of stock-trading software. Within forty-five minutes a bug in that software lost the company $440,000,000. The next day, the CEO of Knight commented that the bug caused the software to enter "a ton of orders, all erroneous."

that behavior. The key to being consistently good at debugging is being systematic in conducting that search.

Start by studying the available data. This includes the test results and the program text. Study all of the test results. Examine not only the tests that revealed the presence of a problem, but also those tests that seemed to work perfectly. Trying to understand why one test worked and another did not is often illuminating. When looking at the program text, keep in mind that you don't completely understand it. If you did, there probably wouldn't be a bug.

Next, form a hypothesis that you believe to be consistent with all the data. The hypothesis could be as narrow as "if I change line 403 from `x < y` to `x <= y`, the problem will go away" or as broad as "my program is not terminating because I have the wrong exit condition in some `while` loop."

Next, design and run a repeatable experiment with the potential to refute the hypothesis. For example, you might put a print statement before and after each `while` loop. If these are always paired, then the hypothesis that a `while` loop is causing nontermination has been refuted. Decide before running the experiment how you would interpret various possible results. If you wait until after you run the experiment, you are more likely to fall prey to wishful thinking.

Finally, it's important to keep a record of what experiments you have tried. When you've spent many hours changing your code trying to track down an elusive bug, it's easy to forget what you have already tried. If you aren't careful, it is easy to waste way too many hours trying the same experiment (or more likely an experiment that looks different but will give you the same information) over and over again. Remember, as many have said, "insanity is doing the same thing, over and over again, but expecting different results."[43]

6.2.2 Designing the Experiment

Think of debugging as a search process, and each experiment as an attempt to reduce the size of the search space. One way to reduce the size of the search space is to design an experiment that can be used to decide whether a specific region of code is responsible for a problem uncovered during integration testing. Another way to reduce the search space is to reduce the amount of test data needed to provoke a manifestation of a bug.

Let's look at a contrived example to see how one might go about debugging it. Imagine that you wrote the palindrome-checking code in Figure 6.3.

[43] This line appears in Rita Mae Brown's, *Sudden Death*. However, it has been variously attributed to many other sources—including Albert Einstein.

```
def isPal(x):
    """Assumes x is a list
       Returns True if the list is a palindrome; False otherwise"""
    temp = x
    temp.reverse
    if temp == x:
        return True
    else:
        return False

def silly(n):
    """Assumes n is an int > 0
       Gets n inputs from user
       Prints 'Yes' if the sequence of inputs forms a palindrome;
           'No' otherwise"""
    for i in range(n):
        result = []
        elem = input('Enter element: ')
        result.append(elem)
    if isPal(result):
        print('Yes')
    else:
        print('No')
```

Figure 6.3 Program with bugs

Now, imagine that you are so confident of your programming skills that you put this code up on the Web—without testing it. Suppose further that you receive an email saying, "I tested your !!**! program on the following 1000-string input, and it printed Yes. Yet any fool can see that it is not a palindrome. Fix it!"

You could try and test it on the supplied 1000-string input. But it might be more sensible to begin by trying it on something smaller. In fact, it would make sense to test it on a minimal non-palindrome, e.g.,

```
>>> silly(2)
Enter element: a
Enter element: b
```

The good news is that it fails even this simple test, so you don't have to type in a thousand strings. The bad news is that you have no idea why it failed.

In this case, the code is small enough that you can probably stare at it and find the bug (or bugs). However, let's pretend that it is too large to do this, and start to systematically reduce the search space.

Often the best way to do this is to conduct a bisection search. Find some point about halfway through the code, and devise an experiment that will allow you to decide if there is a problem before that point that might be related to the symptom. (Of course, there may be problems after that point as well, but it is usually best to hunt down one problem at a time.) In choosing such a point, look for a place where there are some easily examined intermediate values that provide useful information. If an intermediate value is not what you expected, there is probably a problem that occurred prior to that point in the code. If the intermediate values all look fine, the bug probably lies somewhere later in the code. This process can be repeated until you have narrowed the region in which a problem is located to a few lines of code.

Looking at `silly`, the halfway point is around the line `if isPal(result)`. The obvious thing to check is whether `result` has the expected value, `['a', 'b']`. We check this by inserting the statement `print(result)` before the `if` statement in `silly`. When the experiment is run, the program prints `['b']`, suggesting that something has already gone wrong. The next step is to print `result` roughly halfway through the loop. This quickly reveals that `result` is never more than one element long, suggesting that the initialization of `result` needs to be moved outside the `for` loop.

The "corrected" code for `silly` is

```
def silly(n):
    """Assumes n is an int > 0
       Gets n inputs from user
       Prints 'Yes' if the sequence of inputs forms a palindrome;
           'No' otherwise"""
    result = []
    for i in range(n):
        elem = input('Enter element: ')
        result.append(elem)
    print(result)
    if isPal(result):
        print('Yes')
    else:
        print('No')
```

Let's try that, and see if `result` has the correct value after the `for` loop. It does, but unfortunately the program still prints `Yes`. Now, we have reason to believe that a second bug lies below the `print` statement. So, let's look at `isPal`. The line of code `if temp == x:` is about halfway through that function. So, we insert the line

```
print(temp, x)
```

before that line. When we run the code, we see that `temp` has the expected value, but `x` does not. Moving up the code, we insert a print statement after the line of code `temp = x`, and discover that both `temp` and `x` have the value `['a', 'b']`. A quick inspection of the code reveals that in `isPal` we wrote `temp.reverse` rather than `temp.reverse()`—the evaluation of `temp.reverse` returns the built-in `reverse` method for lists, but does not invoke it.[44]

We run the test again, and now it seems that both `temp` and `x` have the value `['b','a']`. We have now narrowed the bug to one line. It seems that `temp.reverse()` unexpectedly changed the value of `x`. An aliasing bug has bitten us: `temp` and `x` are names for the same list, both before and after the list gets reversed. One way to fix it is to replace the first assignment statement in `isPal` by `temp = x[:]`, which causes a copy of `x` to be made.

The corrected version of `isPal` is

```
def isPal(x):
    """Assumes x is a list
       Returns True if the list is a palindrome; False otherwise"""
    temp = x[:]
    temp.reverse()
    if temp == x:
        return True
    else:
        return False
```

6.2.3 When the Going Gets Tough

Joseph P. Kennedy, father of U.S. President John F. Kennedy, reputedly instructed his children, "When the going gets tough, the tough get going."[45] But he never debugged a piece of software. This subsection contains a few pragmatic hints about what do when the debugging gets tough.

- *Look for the usual suspects*. E.g., have you
    - Passed arguments to a function in the wrong order,
    - Misspelled a name, e.g., typed a lowercase letter when you should have typed an uppercase one,
    - Failed to reinitialize a variable,
    - Tested that two floating point values are equal (==) instead of nearly equal (remember that floating point arithmetic is not the same as the arithmetic you learned in school),
    - Tested for value equality (e.g., compared two lists by writing the expression `L1 == L2`) when you meant object equality (e.g., `id(L1) == id(L2)`),

[44] One might well wonder why there isn't a static checker that detected the fact that the line of code `temp.reverse` doesn't do any useful computatation, and is therefore likely to be an error.

[45] He also reputedly told JFK, "Don't buy a single vote more than necessary. I'll be damned if I'm going to pay for a landslide."

- Forgotten that some built-in function has a side effect,
- Forgotten the `()` that turns a reference to an object of type `function` into a function invocation,
- Created an unintentional alias, or
- Made any other mistake that is typical for you.

- *Stop asking yourself why the program isn't doing what you want it to. Instead, ask yourself why it is doing what it is.* That should be an easier question to answer, and will probably be a good first step in figuring out how to fix the program.
- *Keep in mind that the bug is probably not where you think it is.* If it were, you would probably have found it long ago. One practical way to go about deciding where to look is asking where the bug cannot be. As Sherlock Holmes said, "Eliminate all other factors, and the one which remains must be the truth."[46]
- *Try to explain the problem to somebody else.* We all develop blind spots. It is often the case that merely attempting to explain the problem to someone will lead you to see things you have missed. A good thing to try to explain is why the bug cannot be in certain places.
- *Don't believe everything you read.* In particular, don't believe the documentation. The code may not be doing what the comments suggest.
- *Stop debugging and start writing documentation.* This will help you approach the problem from a different perspective.
- *Walk away, and try again tomorrow.* This may mean that bug is fixed later in time than if you had stuck with it, but you will probably spend a lot less of your time looking for it. That is, it is possible to trade latency for efficiency. (Students, this is an excellent reason to start work on programming problem sets earlier rather than later!)

6.2.4 When You Have Found "The" Bug

When you think you have found a bug in your code, the temptation to start coding and testing a fix is almost irresistible. It is often better, however, to slow down a little. Remember that the goal is not to fix one bug, but to move rapidly and efficiently towards a bug-free program.

Ask yourself if this bug explains all the observed symptoms, or whether it is just the tip of the iceberg. If the latter, it may be better to think about taking care of this bug in concert with other changes. Suppose, for example, that you have discovered that the bug is the result of having accidentally mutated a list. You could circumvent the problem locally, perhaps by making a copy of the list. Al-

[46] Arthur Conan Doyle, "The Sign of the Four."

ternatively, you could consider using a tuple instead of a list (since tuples are immutable), perhaps eliminating similar bugs elsewhere in the code.

Before making any change, try and understand the ramification of the proposed "fix." Will it break something else? Does it introduce excessive complexity? Does it offer the opportunity to tidy up other parts of the code?

Always make sure that you can get back to where you are. There is nothing more frustrating than realizing that a long series of changes have left you farther from the goal than when you started, and having no way to get back to where you started. Disk space is usually plentiful. Use it to store old versions of your program.

Finally, if there are many unexplained errors, you might consider whether finding and fixing bugs one at a time is even the right approach. Maybe you would be better off thinking about whether there is some better way to organize your program or some simpler algorithm that will be easier to implement correctly.

7 EXCEPTIONS AND ASSERTIONS

An "exception" is usually defined as "something that does not conform to the norm," and is therefore somewhat rare. There is nothing rare about **exceptions** in Python. They are everywhere. Virtually every module in the standard Python library uses them, and Python itself will raise them in many different circumstances. You've already seen some of them.

Open a Python shell and enter

```
test = [1,2,3]
test[3]
```

and the interpreter will respond with something like

```
IndexError: list index out of range
```

`IndexError` is the type of exception that Python **raises** when a program tries to access an element that is outside the bounds of an indexable type. The string following `IndexError` provides additional information about what caused the exception to occur.

Most of the built-in exceptions of Python deal with situations in which a program has attempted to execute a statement with no appropriate semantics. (We will deal with the exceptional exceptions—those that do not deal with errors—later in this chapter.) Those readers (all of you, we hope) who have attempted to write and run Python programs will already have encountered many of these. Among the most commonly occurring types of exceptions are `TypeError`, `IndexError`, `NameError`, and `ValueError`.

7.1 Handling Exceptions

Up to now, we have treated exceptions as fatal events. When an exception is raised, the program terminates (crashes might be a more appropriate word in this case), and we go back to our code and attempt to figure out what went wrong. When an exception is raised that causes the program to terminate, we say that an **unhandled exception** has been raised.

An exception does not need to lead to program termination. Exceptions, when raised, can and should be **handled** by the program. Sometimes an exception is raised because there is a bug in the program (like accessing a variable that doesn't exist), but many times, an exception is something the programmer can and should anticipate. A program might try to open a file that does not exist. If an interactive program asks a user for input, the user might enter something inappropriate.

If you know that a line of code might raise an exception when executed, you should handle the exception. In a well-written program, unhandled exceptions should be the exception.

Consider the code

```
successFailureRatio = numSuccesses/numFailures
print('The success/failure ratio is', successFailureRatio)
print('Now here')
```

Most of the time, this code will work just fine, but it will fail if `numFailures` happens to be zero. The attempt to divide by zero will cause the Python runtime system to raise a `ZeroDivisionError` exception, and the `print` statements will never be reached.

It would have been better to have written something along the lines of

```
try:
    successFailureRatio = numSuccesses/numFailures
    print('The success/failure ratio is', successFailureRatio)
except ZeroDivisionError:
    print('No failures, so the success/failure ratio is undefined.')
print('Now here')
```

Upon entering the `try` block, the interpreter attempts to evaluate the expression `numSuccesses/numFailures`. If expression evaluation is successful, the program assigns the value of the expression to the variable `successFailureRatio`, executes the `print` statement at the end of the `try` block, and proceeds to the `print` statement following the `try-except`. If, however, a `ZeroDivisionError` exception is raised during the expression evaluation, control immediately jumps to the `except` block (skipping the assignment and the `print` statement in the `try` block), the `print` statement in the `except` block is executed, and then execution continues at the `print` statement following the `try-except` block.

Finger exercise: Implement a function that meets the specification below. Use a `try-except` block.

```
def sumDigits(s):
    """Assumes s is a string
       Returns the sum of the decimal digits in s
          For example, if s is 'a2b3c' it returns 5"""
```

Let's look at another example. Consider the code

```
val = int(input('Enter an integer: '))
print('The square of the number you entered is', val**2)
```

If the user obligingly types a string that can be converted to an integer, everything will be fine. But suppose the user types abc? Executing the line of code will cause the Python runtime system to raise a ValueError exception, and the print statement will never be reached.

What the programmer should have written would look something like

```
while True:
    val = input('Enter an integer: ')
    try:
        val = int(val)
        print('The square of the number you entered is', val**2)
        break #to exit the while loop
    except ValueError:
        print(val, 'is not an integer')
```

After entering the loop, the program will ask the user to enter an integer. Once the user has entered something, the program executes the try–except block. If neither of the first two statements in the try block causes a ValueError exception to be raised, the break statement is executed and the while loop is exited. However, if executing the code in the try block raises a ValueError exception, control is immediately transferred to the code in the except block. Therefore, if the user enters a string that does not represent an integer, the program will ask the user to try again. No matter what text the user enters, it will not cause an unhandled exception.

The downside of this change is that the program text has grown from two lines to eight. If there are many places where the user is asked to enter an integer, this can be problematical. Of course, this problem can be solved by introducing a function:

```
def readInt():
    while True:
        val = input('Enter an integer: ')
        try:
            return(int(val)) #convert str to int before returning
        except ValueError:
            print(val, 'is not an integer')
```

Better yet, this function can be generalized to ask for any type of input:

```
def readVal(valType, requestMsg, errorMsg):
  while True:
      val = input(requestMsg + ' ')
      try:
          return(valType(val)) #convert str to valType before returning
      except ValueError:
          print(val, errorMsg)

readVal(int, 'Enter an integer:', 'is not an integer')
```

The function `readVal` is **polymorphic**, i.e., it works for arguments of many different types. Such functions are easy to write in Python, since types are first-class objects. We can now ask for an integer using the code

```
val = readVal(int, 'Enter an integer:', 'is not an integer')
```

Exceptions may seem unfriendly (after all, if not handled, an exception will cause the program to crash), but consider the alternative. What should the type conversion `int` do, for example, when asked to convert the string `'abc'` to an object of type `int`? It could return an integer corresponding to the bits used to encode the string, but this is unlikely to have any relation to the intent of the programmer. Alternatively, it could return the special value `None`. If it did that, the programmer would need to insert code to check whether the type conversion had returned `None`. A programmer who forgot that check would run the risk of getting some strange error during program execution.

With exceptions, the programmer still needs to include code dealing with the exception. However, if the programmer forgets to include such code and the exception is raised, the program will halt immediately. This is a good thing. It alerts the user of the program to the fact that something troublesome has happened. (And, as we discussed in Chapter 6, overt bugs are much better than covert bugs.) Moreover, it gives someone debugging the program a clear indication of where things went awry.

If it is possible for a block of program code to raise more than one kind of exception, the reserved word `except` can be followed by a tuple of exceptions, e.g.,

```
except (ValueError, TypeError):
```

in which case the `except` block will be entered if any of the listed exceptions is raised within the `try` block.

Alternatively, we can write a separate `except` block for each kind of exception, which allows the program to choose an action based upon which exception was raised. If the programmer writes

```
except:
```

the `except` block will be entered if any kind of exception is raised within the `try` block. These features are shown in Figure 7.1.

7.2 Exceptions as a Control Flow Mechanism

Don't think of exceptions as purely for errors. They are a convenient flow-of-control mechanism that can be used to simplify programs.

In many programming languages, the standard approach to dealing with errors is to have functions return a value (often something analogous to Python's `None`) indicating that something has gone amiss. Each function invocation has to check whether that value has been returned. In Python, it is more usual to have a function raise an exception when it cannot produce a result that is consistent with the function's specification.

The Python **`raise`** statement forces a specified exception to occur. The form of a raise statement is

```
raise exceptionName(arguments)
```

The *exceptionName* is usually one of the built-in exceptions, e.g., `ValueError`. However, programmers can define new exceptions by creating a subclass (see Chapter 8) of the built-in class `Exception`. Different types of exceptions can have different types of arguments, but most of the time the argument is a single string, which is used to describe the reason the exception is being raised.

Finger Exercise: Implement a function that satisfies the specification

```
def findAnEven(L):
    """Assumes L is a list of integers
       Returns the first even number in L
       Raises ValueError if L does not contain an even number"""
```

Consider the function definition in Figure 7.1.

```
def getRatios(vect1, vect2):
    """Assumes: vect1 and vect2 are equal length lists of numbers
       Returns: a list containing the meaningful values of
             vect1[i]/vect2[i]"""
    ratios = []
    for index in range(len(vect1)):
        try:
            ratios.append(vect1[index]/vect2[index])
        except ZeroDivisionError:
            ratios.append(float('nan')) #nan = Not a Number
        except:
            raise ValueError('getRatios called with bad arguments')
    return ratios
```

Figure 7.1 Using exceptions for control flow

There are two except blocks associated with the try block. If an exception is raised within the try block, Python first checks to see if it is a ZeroDivisionError. If so, it appends a special value, nan, of type float to ratios. (The value nan stands for "not a number." There is no literal for it, but it can be denoted by converting the string 'nan' or the string 'NaN' to type float. When nan is used as an operand in an expression of type float, the value of that expression is also nan.) If the exception is anything other than a ZeroDivisionError, the code executes the second except block, which raises a ValueError exception with an associated string.

In principle, the second except block should never be entered, because the code invoking getRatios should respect the assumptions in the specification of getRatios. However, since checking these assumptions imposes only an insignificant computational burden, it is probably worth practicing defensive programming and checking anyway.

The following code illustrates how a program might use getRatios. The name msg in the line except ValueError as msg: is bound to the argument (a string in this case) associated with ValueError when it was raised.[47] When the code

```
try:
    print(getRatios([1.0,2.0,7.0,6.0], [1.0,2.0,0.0,3.0]))
    print(getRatios([], []))
    print(getRatios([1.0, 2.0], [3.0]))
except ValueError as msg:
    print(msg)
```

[47] In Python 2 one writes except ValueError, msg rather than except ValueError as msg.

is executed it prints

```
[1.0, 1.0, nan, 2.0]
[]
getRatios called with bad arguments
```

For comparison, Figure 7.2 contains an implementation of the same specification, but without using a try-except.

```
def getRatios(vect1, vect2):
    """Assumes: vect1 and vect2 are lists of equal length of numbers
       Returns: a list containing the meaningful values of
             vect1[i]/vect2[i]"""
    ratios = []
    if len(vect1) != len(vect2):
        raise ValueError('getRatios called with bad arguments')
    for index in range(len(vect1)):
        vect1Elem = vect1[index]
        vect2Elem = vect2[index]
        if (type(vect1Elem) not in (int, float))\
           or (type(vect2Elem) not in (int, float)):
            raise ValueError('getRatios called with bad arguments')
        if vect2Elem == 0.0:
            ratios.append(float('NaN')) #NaN = Not a Number
        else:
            ratios.append(vect1Elem/vect2Elem)
    return ratios
```

Figure 7.2 Control flow without a try-except

The code in Figure 7.2 is longer and more difficult to read than the code in Figure 7.1. It is also less efficient. (The code in Figure 7.2 could be slightly shortened by eliminating the local variables vect1Elem and vect2Elem, but only at the cost of introducing yet more inefficiency by indexing into the lists repeatedly.)

Let us look at one more example, Figure 7.3. The function getGrades either returns a value or raises an exception with which it has associated a value. It raises a ValueError exception if the call to open raises an IOError. It could have ignored the IOError and let the part of the program calling getGrades deal with it, but that would have provided less information to the calling code about what went wrong. The code that uses getGrades either uses the returned value to compute another value or handles the exception and prints an informative error message.

```
def getGrades(fname):
    try:
        gradesFile = open(fname, 'r') #open file for reading
    except IOError:
        raise ValueError('getGrades could not open ' + fname)
    grades = []
    for line in gradesFile:
        try:
            grades.append(float(line))
        except:
            raise ValueError('Unable to convert line to float')
    return grades

try:
   grades = getGrades('quiz1grades.txt')
   grades.sort()
   median = grades[len(grades)//2]
   print('Median grade is', median)
except ValueError as errorMsg:
   print('Whoops.', errorMsg)
```

Figure 7.3 Get grades

7.3 Assertions

The Python `assert` statement provides programmers with a simple way to confirm that the state of a computation is as expected. An `assert` statement can take one of two forms:

`assert` *`Boolean expression`*

or

`assert` *`Boolean expression, argument`*

When an `assert` statement is encountered, the Boolean expression is evaluated. If it evaluates to `True`, execution proceeds on its merry way. If it evaluates to `False`, an `AssertionError` exception is raised.

Assertions are a useful defensive programming tool. They can be used to confirm that the arguments to a function are of appropriate types. They are also a useful debugging tool. The can be used, for example, to confirm that intermediate values have the expected values or that a function returns an acceptable value.

8 CLASSES AND OBJECT-ORIENTED PROGRAMMING

We now turn our attention to our last major topic related to programming in Python: using classes to organize programs around modules and data abstractions.

Classes can be used in many different ways. In this book we emphasize using them in the context of **object-oriented programming**. The key to object-oriented programming is thinking about objects as collections of both data and the methods that operate on that data.

The ideas underlying object-oriented programming are more than forty years old, and have been widely accepted and practiced over the last twenty-five years or so. In the mid-1970s people began to write articles explaining the benefits of this approach to programming. About the same time, the programming languages SmallTalk (at Xerox PARC) and CLU (at MIT) provided linguistic support for the ideas. But it wasn't until the arrival of C++ and Java that it really took off in practice.

We have been implicitly relying on object-oriented programming throughout most of this book. Back in Section 2.1.1 we said "Objects are the core things that Python programs manipulate. Every object has a **type** that defines the kinds of things that programs can do with that object." Since Chapter 2, we have relied upon built-in types such as `list` and `dict` and the methods associated with those types. But just as the designers of a programming language can build in only a small fraction of the useful functions, they can build in only a small fraction of the useful types. We have already looked at a mechanism that allows programmers to define new functions; we now look at a mechanism that allows programmers to define new types.

8.1 Abstract Data Types and Classes

The notion of an abstract data type is quite simple. An **abstract data type** is a set of objects and the operations on those objects. These are bound together so that one can pass an object from one part of a program to another, and in doing so provide access not only to the data attributes of the object but also to operations that make it easy to manipulate that data.

The specifications of those operations define an **interface** between the abstract data type and the rest of the program. The interface defines the behavior of the operations—what they do, but not how they do it. The interface thus provides an **abstraction barrier** that isolates the rest of the program from the data structures, algorithms, and code involved in providing a realization of the type abstraction.

Programming is about managing complexity in a way that facilitates change. There are two powerful mechanisms available for accomplishing this: decomposition and abstraction. Decomposition creates structure in a program, and abstraction suppresses detail. The key is to suppress the appropriate details. This is where data abstraction hits the mark. One can create domain-specific types that provide a convenient abstraction. Ideally, these types capture concepts that will be relevant over the lifetime of a program. If one starts the programming process by devising types that will be relevant months and even decades later, one has a great leg up in maintaining that software.

We have been using abstract data types (without calling them that) throughout this book. We have written programs using integers, lists, floats, strings, and dictionaries without giving any thought to how these types might be implemented. To paraphrase Molière's *Bourgeois Gentilhomme*, *"Par ma foi, il y a plus de cent pages que nous avons utilisé ADTs, sans que nous le sachions."*[48]

In Python, one implements data abstractions using **classes**. Figure 8.1 contains a **class definition** that provides a straightforward implementation of a set-of-integers abstraction called `IntSet`.

A class definition creates an object of type `type` and associates with that class object a set of objects of type `instancemethod`. For example, the expression `IntSet.insert` refers to the method `insert` defined in the definition of the class `IntSet`. And the code

```
print(type(IntSet), type(IntSet.insert))
```

will print

```
<class 'type'> <class 'function'>
```

Notice that the docstring (the comment enclosed in `"""`) at the top of the class definition describes the abstraction provided by the class, not information about how the class is implemented. In contrast, the comments below the docstring contain information about the implementation. That information is aimed at programmers who might want to modify the implementation or build sub-

[48] "Good heavens, for more than one hundred pages we have been using ADTs without knowing it."

classes (see Section 8.2) of the class, not at programmers who might want to use the abstraction.

```
class IntSet(object):
    """An intSet is a set of integers"""
    #Information about the implementation (not the abstraction)
    #Value of the set is represented by a list of ints, self.vals.
    #Each int in the set occurs in self.vals exactly once.

    def __init__(self):
        """Create an empty set of integers"""
        self.vals = []

    def insert(self, e):
        """Assumes e is an integer and inserts e into self"""
        if e not in self.vals:
            self.vals.append(e)

    def member(self, e):
        """Assumes e is an integer
           Returns True if e is in self, and False otherwise"""
        return e in self.vals

    def remove(self, e):
        """Assumes e is an integer and removes e from self
           Raises ValueError if e is not in self"""
        try:
            self.vals.remove(e)
        except:
            raise ValueError(str(e) + ' not found')

    def getMembers(self):
        """Returns a list containing the elements of self.
           Nothing can be assumed about the order of the elements"""
        return self.vals[:]

    def __str__(self):
        """Returns a string representation of self"""
        self.vals.sort()
        result = ''
        for e in self.vals:
            result = result + str(e) + ','
        return '{' + result[:-1] + '}' #-1 omits trailing comma
```

Figure 8.1 Class IntSet

When a function definition occurs within a class definition, the defined function is called a **method** and is associated with the class. These methods are sometimes referred to as **method attributes** of the class. If this seems confusing at the moment, don't worry about it. We will have lots more to say about this topic later in this chapter.

Classes support two kinds of operations:

- **Instantiation** is used to create instances of the class. For example, the statement `s = IntSet()` creates a new object of type `IntSet`. This object is called an **instance** of `IntSet`.
- **Attribute references** use dot notation to access attributes associated with the class. For example, `s.member` refers to the method `member` associated with the instance `s` of type `IntSet`.

Each class definition begins with the reserved word `class` followed by the name of the class and some information about how it relates to other classes. In this case, the first line indicates that `IntSet` is a subclass of `object`. For now, ignore what it means to be a subclass. We will get to that shortly.

As we will see, Python has a number of special method names that start and end with two underscores. The first of these we will look at is `__init__`. Whenever a class is instantiated, a call is made to the `__init__` method defined in that class. When the line of code

```
s = IntSet()
```

is executed, the interpreter will create a new instance of type `IntSet`, and then call `IntSet.__init__` with the newly created object as the actual parameter that is bound to the formal parameter `self`. When invoked, `IntSet.__init__` creates `vals`, an object of type `list`, which becomes part of the newly created instance of type `IntSet`. (The list is created using the by now familiar notation `[]`, which is simply an abbreviation for `list()`.) This list is called a **data attribute** of the instance of `IntSet`. Notice that each object of type `IntSet` will have a different `vals` list, as one would expect.

As we have seen, methods associated with an instance of a class can be invoked using dot notation. For example, the code,

```
s = IntSet()
s.insert(3)
print(s.member(3))
```

creates a new instance of `IntSet`, inserts the integer 3 into that `IntSet`, and then prints `True`.

At first blush there appears to be something inconsistent here. It looks as if each method is being called with one argument too few. For example, `member` has two formal parameters, but we appear to be calling it with only one actual parameter. This is an artifact of the dot notation. The object associated with the expression preceding the dot is implicitly passed as the first parameter to the method. Throughout this book, we follow the convention of using `self` as the name of the formal parameter to which this actual parameter is bound. Python programmers observe this convention almost universally, and we strongly suggest that you use it as well.

A class should not be confused with instances of that class, just as an object of type `list` should not be confused with the `list` type. Attributes can be associated either with a class itself or with instances of a class:

- Method attributes are defined in a class definition, for example `IntSet.member` is an attribute of the class `IntSet`. When the class is instantiated, e.g., by the statement `s = IntSet()`, instance attributes, e.g., `s.member`, are created. Keep in mind that `IntSet.member` and `s.member` are different objects. While `s.member` is initially bound to the `member` method defined in the class `IntSet`, that binding can be changed during the course of a computation. For example, you could (but shouldn't!) write `s.member = IntSet.insert`.
- When data attributes are associated with a class we call them **class variables**. When they are associated with an instance we call them **instance variables**. For example, `vals` is an instance variable because for each instance of class `IntSet`, `vals` is bound to a different list. So far, we haven't seen a class variable. We will use one in Figure 8.3.

Data abstraction achieves representation-independence. Think of the implementation of an abstract type as having several components:

- Implementations of the methods of the type,
- Data structures that together encode values of the type, and
- Conventions about how the implementations of the methods are to use the data structures. A key convention is captured by the representation invariant.

The **representation invariant** defines which values of the data attributes correspond to valid representations of class instances. The representation invariant for `IntSet` is that `vals` contains no duplicates. The implementation of `__init__` is responsible for establishing the invariant (which holds on the empty list), and the other methods are responsible for maintaining that invariant. That is why `insert` appends `e` only if it is not already in `self.vals`.

The implementation of `remove` exploits the assumption that the representation invariant is satisfied when `remove` is entered. It calls `list.remove` only once, since the representation invariant guarantees that there is at most one occurrence of e in self.vals.

The last method defined in the class, __str__, is another one of those special __ methods. When the print command is used, the __str__ function associated with the object to be printed is automatically invoked. For example, the code

```
s = IntSet()
s.insert(3)
s.insert(4)
print(s)
```

will print

```
{3,4}
```

(If no __str__ method were defined, executing print(s) would cause something like <__main__.IntSet object at 0x1663510> to be printed.) We could also print the value of s by writing print s.__str__() or even print IntSet.__str__(s), but using those forms is less convenient. The __str__ method of a class is also invoked when a program converts an instance of that class to a string by calling str.

All instances of user-defined classes are hashable, and therefore can be used as dictionary keys. If no __hash__ method is provided, the hash value of the object is derived from the function id (see Section 5.3). If no __eq__ method is provided, all objects are considered unequal (except to themselves). If a user-defined __hash__ is provided, it should ensure that the hash value of an object is constant throughout the lifetime of that object.

8.1.1 Designing Programs Using Abstract Data Types

Abstract data types are a big deal. They lead to a different way of thinking about organizing large programs. When we think about the world, we rely on abstractions. In the world of finance people talk about stocks and bonds. In the world of biology people talk about proteins and residues. When trying to understand concepts such as these, we mentally gather together some of the relevant data and features of these kinds of objects into one intellectual package. For example, we think of bonds as having an interest rate and a maturity date as data attributes. We also think of bonds as having operations such as "set price" and "calculate yield to maturity." Abstract data types allow us to incorporate this kind of organization into the design of programs.

Data abstraction encourages program designers to focus on the centrality of data objects rather than functions. Thinking about a program more as a collection of types than as a collection of functions leads to a profoundly different organizing principle. Among other things, it encourages one to think about programming as a process of combining relatively large chunks, since data abstractions typically encompass more functionality than do individual functions. This, in turn, leads us to think of the essence of programming as a process not of writing individual lines of code, but of composing abstractions.

The availability of reusable abstractions not only reduces development time, but also usually leads to more reliable programs, because mature software is usually more reliable than new software. For many years, the only program libraries in common use were statistical or scientific. Today, however, there is a great range of available program libraries (especially for Python), often based on a rich set of data abstractions, as we shall see later in this book.

8.1.2 Using Classes to Keep Track of Students and Faculty

As an example use of classes, imagine that you are designing a program to help keep track of all the students and faculty at a university. It is certainly possible to write such a program without using data abstraction. Each student would have a family name, a given name, a home address, a year, some grades, etc. This could all be kept in some combination of lists and dictionaries. Keeping track of faculty and staff would require some similar data structures and some different data structures, e.g., data structures to keep track of things like salary history.

Before rushing in to design a bunch of data structures, let's think about some abstractions that might prove useful. Is there an abstraction that covers the common attributes of students, professors, and staff? Some would argue that they are all human. Figure 8.2 contains a class that incorporates some of the common attributes (name and birthday) of humans. It makes use of the standard Python library module `datetime`, which provides many convenient methods for creating and manipulating dates.

```
import datetime

class Person(object):

    def __init__(self, name):
        """Create a person"""
        self.name = name
        try:
            lastBlank = name.rindex(' ')
            self.lastName = name[lastBlank+1:]
        except:
            self.lastName = name
        self.birthday = None

    def getName(self):
        """Returns self's full name"""
        return self.name

    def getLastName(self):
        """Returns self's last name"""
        return self.lastName

    def setBirthday(self, birthdate):
        """Assumes birthdate is of type datetime.date
           Sets self's birthday to birthdate"""
        self.birthday = birthdate

    def getAge(self):
        """Returns self's current age in days"""
        if self.birthday == None:
            raise ValueError
        return (datetime.date.today() - self.birthday).days

    def __lt__(self, other):
        """Returns True if self precedes other in alphabetical
           order, and False otherwise. Comparison is based on last
           names, but if these are the same full names are
           compared."""
        if self.lastName == other.lastName:
            return self.name < other.name
        return self.lastName < other.lastName

    def __str__(self):
        """Returns self's name"""
        return self.name
```

Figure 8.2 Class Person

The following code makes use of Person.

```
me = Person('Michael Guttag')
him = Person('Barack Hussein Obama')
her = Person('Madonna')
print(him.getLastName())
him.setBirthday(datetime.date(1961, 8, 4))
her.setBirthday(datetime.date(1958, 8, 16))
print(him.getName(), 'is', him.getAge(), 'days old')
```

Notice that whenever `Person` is instantiated an argument is supplied to the `__init__` function. In general, when instantiating a class we need to look at the specification of the `__init__` function for that class to know what arguments to supply and what properties those arguments should have.

After the above code is executed, there will be three instances of class `Person`. One can then access information about these instances using the methods associated with them. For example, `him.getLastName()` will return `'Obama'`. The expression `him.lastName` will also return `'Obama'; however`, for reasons discussed later in this chapter, writing expressions that directly access instance variables is considered poor form, and should be avoided. Similarly, there is no appropriate way for a user of the `Person` abstraction to extract a person's birthday, despite the fact that the implementation contains an attribute with that value. (Of course, it would be easy to add a `getBirthday` method to the class.) There is, however, a way to extract information that depends upon the person's birthday, as illustrated by the last `print` statement in the above code.

Class `Person` defines yet another specially named method, `__lt__`. This method overloads the < operator. The method `Person__lt__` gets called whenever the first argument to the < operator is of type `Person`. The `__lt__` method in class Person is implemented using the binary < operator of type `str`. The expression `self.name < other.name` is shorthand for `self.name.__lt__(other.name)`. Since `self.name` is of type `str`, this `__lt__` method is the one associated with type `str`.

In addition to providing the syntactic convenience of writing infix expressions that use <, this overloading provides automatic access to any polymorphic method defined using `__lt__`. The built-in method `sort` is one such method. So, for example, if `pList` is a list composed of elements of type `Person`, the call `pList.sort()` will sort that list using `the __lt__` method defined in class `Person`.

The code

```
pList = [me, him, her]
for p in pList:
    print(p)
pList.sort()
for p in pList:
    print(p)
```

will first print

```
Michael Guttag
Barack Hussein Obama
Madonna
```

and then print

```
Michael Guttag
Madonna
Barack Hussein Obama
```

8.2 Inheritance

Many types have properties in common with other types. For example, types `list` and `str` each have `len` functions that mean the same thing. **Inheritance** provides a convenient mechanism for building groups of related abstractions. It allows programmers to create a type hierarchy in which each type inherits attributes from the types above it in the hierarchy.

The class `object` is at the top of the hierarchy. This makes sense, since in Python everything that exists at run time is an object. Because `Person` inherits all of the properties of objects, programs can bind a variable to a `Person`, append a `Person` to a list, etc.

The class `MITPerson` in Figure 8.3 inherits attributes from its parent class, `Person`, including all of the attributes that `Person` inherited from its parent class, `object`. In the jargon of object-oriented programming, `MITPerson` is a **subclass** of `Person`, and therefore **inherits** the attributes of its **superclass**. In addition to what it inherits, the subclass can:

- Add new attributes. For example, the subclass `MITPerson` has added the class variable `nextIdNum`, the instance variable `idNum`, and the method `getIdNum`.
- **Override**, i.e., replace, attributes of the superclass. For example, MITPerson has overridden `__init__` and `__lt__`. When a method has been overridden, the version of the method that is executed is based on the object that is used to invoke the method. If the type of the object is the subclass, the version defined in

the subclass will be used. If the type of the object is the superclass, the version in the superclass will be used.

The method `MITPerson.__init__` first invokes `Person.__init__` to initialize the `inherited` instance variable `self.name`. It then initializes `self.idNum`, an instance variable that instances of `MITPerson` have but instances of `Person` do not.

The instance variable `self.idNum` is initialized using a **class variable**, `nextIdNum`, that belongs to the class `MITPerson`, rather than to instances of the class. When an instance of `MITPerson` is created, a new instance of `nextIdNum` is not created. This allows `__init__` to ensure that each instance of `MITPerson` has a unique `idNum`.

```
class MITPerson(Person):

    nextIdNum = 0 #identification number

    def __init__(self, name):
        Person.__init__(self, name)
        self.idNum = MITPerson.nextIdNum
        MITPerson.nextIdNum += 1

    def getIdNum(self):
        return self.idNum

    def __lt__(self, other):
        return self.idNum < other.idNum
```

Figure 8.3 Class `MITPerson`

Consider the code

```
p1 = MITPerson('Barbara Beaver')
print(str(p1) + '\'s id number is ' + str(p1.getIdNum()))
```

The first line creates a new `MITPerson`. The second line is a bit more complicated. When it attempts to evaluate the expression `str(p1)`, the runtime system first checks to see if there is an `__str__` method associated with class `MITPerson`. Since there is not, it next checks to see if there is an `__str__` method associated with the superclass, `Person`, of `MITPerson`. There is, so it uses that. When the runtime system attempts to evaluate the expression `p1.getidNum()`, it first checks to see if there is a `getIdNum` method associated with class `MITPerson`. There is, so it invokes that method and prints

```
Barbara Beaver's id number is 0
```

(Recall that in a string, the character "\" is an escape character used to indicate that the next character should be treated in a special way. In the string

```
'\'s id number is '
```

the "\" indicates that the apostrophe is part of the string, not a delimiter terminating the string.)

Now consider the code

```
p1 = MITPerson('Mark Guttag')
p2 = MITPerson('Billy Bob Beaver')
p3 = MITPerson('Billy Bob Beaver')
p4 = Person('Billy Bob Beaver')
```

We have created four virtual people, three of whom are named Billy Bob Beaver. Two of the Billy Bobs are of type `MITPerson`, and one merely a `Person`. If we execute the lines of code

```
print('p1 < p2 =', p1 < p2)
print('p3 < p2 =', p3 < p2)
print('p4 < p1 =', p4 < p1)
```

the interpreter will print

```
p1 < p2 = True
p3 < p2 = False
p4 < p1 = True
```

Since `p1`, `p2`, and `p3` are all of type `MITPerson`, the interpreter will use the `__lt__` method defined in class `MITPerson` when evaluating the first two comparisons, so the ordering will be based on identification numbers. In the third comparison, the `<` operator is applied to operands of different types. Since the first argument of the expression is used to determine which `__lt__` method to invoke, `p4 < p1` is shorthand for `p4.__lt__(p1)`. Therefore, the interpreter uses the `__lt__ method` associated with the type of `p4`, `Person`, and the "people" will be ordered by name.

What happens if we try

```
Print('p1 < p4 =', p1 < p4)
```

The runtime system will invoke the `__lt__` operator associated with the type of `p1`, i.e., the one defined in class `MITPerson`. This will lead to the exception

```
AttributeError: 'Person' object has no attribute 'idNum'
```

because the object to which `p4` is bound does not have an attribute `idNum`.

8.2.1 Multiple Levels of Inheritance

Figure 8.4 adds another couple of levels of inheritance to the class hierarchy.

```
class Student(MITPerson):
    pass

class UG(Student):
    def __init__(self, name, classYear):
        MITPerson.__init__(self, name)
        self.year = classYear
    def getClass(self):
        return self.year

class Grad(Student):
    pass
```

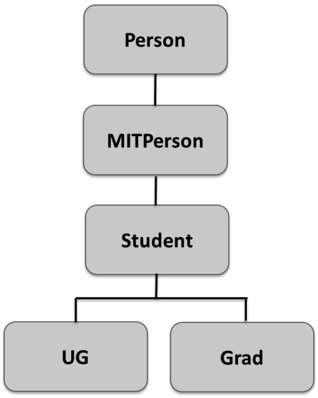


Figure 8.4 Two kinds of students

Adding `UG` seems logical, because we want to associate a year of graduation (or perhaps anticipated graduation) with each undergraduate. But what is going on with the classes `Student` and `Grad`? By using the Python reserved word **pass** as the body, we indicate that the class has no attributes other than those inherited from its superclass. Why would one ever want to create a class with no new attributes?

By introducing the class `Grad`, we gain the ability to create two different kinds of students and use their types to distinguish one kind of object from another. For example, the code

```
p5 = Grad('Buzz Aldrin')
p6 = UG('Billy Beaver', 1984)
print(p5, 'is a graduate student is', type(p5) == Grad)
print(p5, 'is an undergraduate student is', type(p5) == UG)
```

will print

```
Buzz Aldrin is a graduate student is True
Buzz Aldrin is an undergraduate student is False
```

The utility of the intermediate type `Student` is a bit subtler. Consider going back to class `MITPerson` and adding the method

```
def isStudent(self):
    return isinstance(self, Student)
```

The function isinstance is built into Python. The first argument of isinstance can be any object, but the second argument must be an object of type type. The function returns True if and only if the first argument is an instance of the second argument. For example, the value of isinstance([1,2], list) is True.

Returning to our example, the code

```
print(p5, 'is a student is', p5.isStudent())
print(p6, 'is a student is', p6.isStudent())
print(p3, 'is a student is', p3.isStudent())
```

prints

```
Buzz Aldrin is a student is True
Billy Beaver is a student is True
Billy Bob Beaver is a student is False
```

Notice that the meaning of isinstance(p6, Student) is quite different from the meaning of type(p6) == Student. The object to which p6 is bound is of type UG, not student, but since UG is a subclass of Student, the object to which p6 is bound is considered to be an instance of class Student (as well as an instance of MITPerson and Person).

Since there are only two kinds of students, we could have implemented isStudent as,

```
def isStudent(self):
    return type(self) == Grad or type(self) == UG
```

However, if a new type of student were introduced at some later point it would be necessary to go back and edit the code implementing isStudent. By introducing the intermediate class Student and using isinstance we avoid this problem. For example, if we were to add

```
class TransferStudent(Student):

    def __init__(self, name, fromSchool):
        MITPerson.__init__(self, name)
        self.fromSchool = fromSchool

    def getOldSchool(self):
        return self.fromSchool
```

no change needs to be made to isStudent.

It is not unusual during the creation and later maintenance of a program to go back and add new classes or new attributes to old classes. Good programmers design their programs so as to minimize the amount of code that might need to be changed when that is done.

8.2.2 The Substitution Principle

When subclassing is used to define a type hierarchy, the subclasses should be thought of as extending the behavior of their superclasses. We do this by adding new attributes or overriding attributes inherited from a superclass. For example, `TransferStudent` extends `Student` by introducing a former school.

Sometimes, the subclass overrides methods from the superclass, but this must be done with care. In particular, important behaviors of the supertype must be supported by each of its subtypes. If client code works correctly using an instance of the supertype, it should also work correctly when an instance of the subtype is substituted for the instance of the supertype. For example, it should be possible to write client code using the specification of `Student` and have it work correctly on a `TransferStudent`.[49]

Conversely, there is no reason to expect that code written to work for `TransferStudent` should work for arbitrary types of `Student`.

8.3 Encapsulation and Information Hiding

As long as we are dealing with students, it would be a shame not to make them suffer through taking classes and getting grades.

Figure 8.5 contains a class that can be used to keep track of the grades of a collection of students. Instances of class `Grades` are implemented using a list and a dictionary. The list keeps track of the students in the class. The dictionary maps a student's identification number to a list of grades.

Notice that `getGrades` returns a copy of the list of grades associated with a student, and `getStudents` returns a copy of the list of students. The computational cost of copying the lists could have been avoided by simply returning the instance variables themselves. Doing so, however, is likely to lead to problems. Consider the code

```
allStudents = course1.getStudents()
allStudents.extend(course2.getStudents())
```

If `getStudents` returned `self.students`, the second line of code would have the (probably unexpected) side effect of changing the set of students in `course1`.

[49] This **substitution principle** was clearly enunciated by Barbara Liskov and Jeannette Wing in their 1994 paper, "A behavioral notion of subtyping."

The instance variable isSorted is used to keep track of whether or not the list of students has been sorted since the last time a student was added to it. This allows the implementation of getStudents to avoid sorting an already sorted list.

```
class Grades(object):

    def __init__(self):
        """Create empty grade book"""
        self.students = []
        self.grades = {}
        self.isSorted = True

    def addStudent(self, student):
        """Assumes: student is of type Student
           Add student to the grade book"""
        if student in self.students:
            raise ValueError('Duplicate student')
        self.students.append(student)
        self.grades[student.getIdNum()] = []
        self.isSorted = False

    def addGrade(self, student, grade):
        """Assumes: grade is a float
           Add grade to the list of grades for student"""
        try:
            self.grades[student.getIdNum()].append(grade)
        except:
            raise ValueError('Student not in mapping')

    def getGrades(self, student):
        """Return a list of grades for student"""
        try: #return copy of list of student's grades
            return self.grades[student.getIdNum()][:]
        except:
            raise ValueError('Student not in mapping')

    def getStudents(self):
        """Return a sorted list of the students in the grade book"""
        if not self.isSorted:
            self.students.sort()
            self.isSorted = True
        return self.students[:] #return copy of list of students
```

Figure 8.5 Class Grades

Figure 8.6 contains a function that uses class `Grades` to produce a grade report for some students taking a course named `sixHundred`.

```
def gradeReport(course):
    """Assumes course is of type Grades"""
    report = ''
    for s in course.getStudents():
        tot = 0.0
        numGrades = 0
        for g in course.getGrades(s):
            tot += g
            numGrades += 1
        try:
            average = tot/numGrades
            report = report + '\n'\
                     + str(s) + '\'s mean grade is ' + str(average)
        except ZeroDivisionError:
            report = report + '\n'\
                     + str(s) + ' has no grades'
    return report

ug1 = UG('Jane Doe', 2014)
ug2 = UG('John Doe', 2015)
ug3 = UG('David Henry', 2003)
g1 = Grad('Billy Buckner')
g2 = Grad('Bucky F. Dent')
sixHundred = Grades()
sixHundred.addStudent(ug1)
sixHundred.addStudent(ug2)
sixHundred.addStudent(g1)
sixHundred.addStudent(g2)
for s in sixHundred.getStudents():
    sixHundred.addGrade(s, 75)
sixHundred.addGrade(g1, 25)
sixHundred.addGrade(g2, 100)
sixHundred.addStudent(ug3)
print(gradeReport(sixHundred))
```

Figure 8.6 Generating a grade report

When run, the code in the figure prints

```
Jane Doe's mean grade is 75.0
John Doe's mean grade is 75.0
David Henry has no grades
Billy Buckner's mean grade is 50.0
Bucky F. Dent's mean grade is 87.5
```

There are two important concepts at the heart of object-oriented programming. The first is the idea of **encapsulation**. By this we mean the bundling together of data attributes and the methods for operating on them. For example, if we write

```
Rafael = MITPerson('Rafael Reif')
```

we can use dot notation to access attributes such as Rafael's name and identification number.

The second important concept is **information hiding**. This is one of the keys to modularity. If those parts of the program that use a class (i.e., the clients of the class) rely only on the specifications of the methods in the class, a programmer implementing the class is free to change the implementation of the class (e.g., to improve efficiency) without worrying that the change will break code that uses the class.

Some programming languages (Java and C++, for example) provide mechanisms for enforcing information hiding. Programmers can make the attributes of a class **private**, so that clients of the class can access the data only through the object's methods. Python 3 uses a naming convention to make attributes invisible outside the class. When the name of an attribute starts with __ but does not end with __, that attribute is not visible outside the class. Consider the class in Figure 8.7.

```
class infoHiding(object):
    def __init__(self):
        self.visible = 'Look at me'
        self.__alsoVisible__ = 'Look at me too'
        self.__invisible = 'Don\'t look at me directly'

    def printVisible(self):
        print(self.visible)

    def printInvisible(self):
        print(self.__invisible)

    def __printInvisible(self):
        print(self.__invisible)

    def __printInvisible__(self):
        print(self.__invisible)
```

Figure 8.7 Information Hiding in Classes

When we run the code

```
test = infoHiding()
print(test.visible)
print(test.__alsoVisible__)
print(test.__invisible)
```

it prints

```
Look at me
Look at me too
Error: 'infoHiding' object has no attribute '__invisible'
```

The code

```
test = infoHiding()
test.printInvisible()
test.__printInvisible__()
test.__printInvisible()
```

prints

```
Don't look at me directly
Don't look at me directly
Error: 'infoHiding' object has no attribute '__printInvisible'
```

And the code

```
class subClass(infoHiding):
   def __init__(self):
        print('from subclass', self.__invisible)

testSub = subClass()
```

prints

```
Error: 'subClass' object has no attribute '_subClass__invisible'
```

Notice that when a subclass attempts to use a hidden attribute of its superclass an `AttributeError` occurs. This can make using information hiding in Python a bit cumbersome.

Because it can be cumbersome, many Python programmers do not take advantage of the __ mechanism for hiding attributes—as we don't in this book. So, for example, a client of `Person` can write the expression `Rafael.lastName` rather than `Rafael.getLastName()`.

This is unfortunate because it allows the client code to rely upon something that is not part of the specification of `Person`, and is therefore subject to change. If the implementation of `Person` were changed, for example to extract the last name whenever it is requested rather than store it in an instance variable, then the client code would no longer work.

Not only does Python let programs read instance and class variables from outside the class definition, it also lets programs write these variables. So, for example, the code `Rafael.birthday = '8/21/50'` is perfectly legal. This would lead to a runtime type error, were `Rafael.getAge` invoked later in the computation. It is even possible to create instance variables from outside the class definition. For example, Python will not complain if the assignment statement

```
me.age = Rafael.getIdNum()
```

occurs outside the class definition.

While this relatively weak static semantic checking is a flaw in Python, it is not a fatal flaw. A disciplined programmer can simply follow the sensible rule of not directly accessing data attributes from outside the class in which they are defined, as we do in this book.

8.3.1 Generators

A perceived risk of information hiding is that preventing client programs from directly accessing critical data structures leads to an unacceptable loss of efficiency. In the early days of data abstraction, many were concerned about the cost of introducing extraneous function/method calls. Modern compilation technology makes this concern moot. A more serious issue is that client programs will be forced to use inefficient algorithms.

Consider the implementation of `gradeReport` in Figure 8.6. The invocation of `course.getStudents` creates and returns a list of size n, where n is the number of students. This is probably not a problem for a grade book for a single class, but imagine keeping track of the grades of 1.7 million high school students taking the SATs. Creating a new list of that size when the list already exists is a significant inefficiency. One solution is to abandon the abstraction and allow `gradeReport` to directly access the instance variable `course.students`, but that would violate information hiding. Fortunately, there is a better solution.

The code in Figure 8.8, replaces the `getStudents` function in class Grades with a function that uses a kind of statement we have not yet used: a `yield` statement.

Any function definition containing a `yield` statement is treated in a special way. The presence of `yield` tells the Python system that the function is a **generator**. Generators are typically used in conjunction with `for` statements, as in

```
for s in course.getStudents():
```

in Figure 8.6.

```
def getStudents(self):
    """Return the students in the grade book one at a time
       in alphabetical order"""
    if not self.isSorted:
        self.students.sort()
        self.isSorted = True
    for s in self.students:
        yield s
```

Figure 8.8 New version of `getStudents`

At the start of the first iteration of a `for` loop that uses a generator, the generator is invoked and runs until the first time a `yield` statement is executed, at which point it returns the value of the expression in the `yield` statement. On the next iteration, the generator resumes execution immediately following the `yield`, with all local variables bound to the objects to which they were bound when the `yield` statement was executed, and again runs until a `yield` statement is executed. It continues to do this until it runs out of code to execute or executes a `return` statement, at which point the loop is exited.[50]

The version of `getStudents` in Figure 8.8 allows programmers to use a `for` loop to iterate over the students in objects of type `Grades` in the same way they can use a `for` loop to iterate over elements of built-in types such as `list`. For example, the code

```
book = Grades()
book.addStudent(Grad('Julie'))
book.addStudent(Grad('Charlie'))
for s in book.getStudents():
    print(s)
```

prints

```
Julie
Charlie
```

Thus the loop in Figure 8.6 that starts with

```
for s in course.getStudents():
```

does not have to be altered to take advantage of the version of class `Grades` that contains the new implementation of `getStudents`. (Of course, most code that depended upon `getStudents` returning a list would no longer work.) The same `for`

[50] This explanation of generators is a bit simplistic. To fully understand generators, you need to understand how built-in iterators are implemented in Python, which is not covered in this book.

loop can iterate over the values provided by `getStudents` regardless of whether `getStudents` returns a list of values or generates one value at a time. Generating one value at a time will be more efficient, because a new list containing the students will not be created.

8.4 Mortgages, an Extended Example

A collapse in U.S. housing prices helped trigger a severe economic meltdown in the fall of 2008. One of the contributing factors was that many homeowners had taken on mortgages that ended up having unexpected consequences.[51]

In the beginning, mortgages were relatively simple beasts. One borrowed money from a bank and made a fixed-size payment each month for the life of the mortgage, which typically ranged from fifteen to thirty years. At the end of that period, the bank had been paid back the initial loan (the principal) plus interest, and the homeowner owned the house "free and clear."

Towards the end of the twentieth century, mortgages started getting a lot more complicated. People could get lower interest rates by paying "points" to the lender at the time they took on the mortgage. A point is a cash payment of 1% of the value of the loan. People could take mortgages that were "interest-only" for a period of time. That is to say, for some number of months at the start of the loan the borrower paid only the accrued interest and none of the principal. Other loans involved multiple rates. Typically the initial rate (called a "teaser rate") was low, and then it went up over time. Many of these loans were variable-rate—the rate to be paid after the initial period would vary depending upon some index intended to reflect the cost to the lender of borrowing on the wholesale credit market.[52]

In principle, giving consumers a variety of options is a good thing. However, unscrupulous loan purveyors were not always careful to fully explain the possible long-term implications of the various options, and some borrowers made choices that proved to have dire consequences.

Let's build a program that examines the costs of three kinds of loans:

- A fixed-rate mortgage with no points,
- A fixed-rate mortgage with points, and

[51] In this context, it is worth recalling the etymology of the word mortgage. *The American Heritage Dictionary of the English Language* traces the word back to the old French words for dead (*mort*) and pledge (*gage*). (This derivation also explains why the "t" in the middle of mortgage is silent.)

[52] The London Interbank Offered Rate (LIBOR) is probably the most commonly used index.

- A mortgage with an initial teaser rate followed by a higher rate for the duration.

The point of this exercise is to provide some experience in the incremental development of a set of related classes, not to make you an expert on mortgages.

We will structure our code to include a `Mortgage` class, and subclasses corresponding to each of the three kinds of mortgages listed above. Figure 8.9 contains the **abstract class** `Mortgage`. This class contains methods that are shared by each of its subclasses, but it is not intended to be instantiated directly. That is, no objects of type `Mortgage` will be created.

The function `findPayment` at the top of the figure computes the size of the fixed monthly payment needed to pay off the loan, including interest, by the end of its term. It does this using a well-known closed-form expression. This expression is not hard to derive, but it is a lot easier to just look it up and more likely to be correct than one derived on the spot.

Keep in mind that not everything you discover on the Web (or even in textbooks) is correct. When your code incorporates a formula that you have looked up, make sure that:

- You have taken the formula from a reputable source. We looked at multiple reputable sources, all of which contained equivalent formulas.
- You fully understand the meaning of all the variables in the formula.
- You test your implementation against examples taken from reputable sources. After implementing this function, we tested it by comparing our results to the results supplied by a calculator available on the Web.

Looking at `__init__`, we see that all `Mortgage` instances will have instance variables corresponding to the initial loan amount, the monthly interest rate, the duration of the loan in months, a list of payments that have been made at the start of each month (the list starts with `0.0`, since no payments have been made at the start of the first month), a list with the balance of the loan that is outstanding at the start of each month, the amount of money to be paid each month (initialized using the value returned by the function `findPayment`), and a description of the mortgage (which initially has a value of `None`). The `__init__` operation of each subclass of `Mortgage` is expected to start by calling `Mortgage.__init__`, and then to initialize `self.legend` to an appropriate description of that subclass.

```
def findPayment(loan, r, m):
    """Assumes: loan and r are floats, m an int
       Returns the monthly payment for a mortgage of size
       loan at a monthly rate of r for m months"""
    return loan*((r*(1+r)**m)/((1+r)**m - 1))

class Mortgage(object):
    """Abstract class for building different kinds of mortgages"""
    def __init__(self, loan, annRate, months):
        """Assumes: loan and annRate are floats, months an int
        Creates a new mortgage of size loan, duration months, and
        annual rate annRate"""
        self.loan = loan
        self.rate = annRate/12
        self.months = months
        self.paid = [0.0]
        self.outstanding = [loan]
        self.payment = findPayment(loan, self.rate, months)
        self.legend = None #description of mortgage

    def makePayment(self):
        """Make a payment"""
        self.paid.append(self.payment)
        reduction = self.payment - self.outstanding[-1]*self.rate
        self.outstanding.append(self.outstanding[-1] - reduction)

    def getTotalPaid(self):
        """Return the total amount paid so far"""
        return sum(self.paid)

    def __str__(self):
        return self.legend
```

Figure 8.9 `Mortgage` **base class**

The method `makePayment` is used to record mortgage payments. Part of each payment covers the amount of interest due on the outstanding loan balance, and the remainder of the payment is used to reduce the loan balance. That is why `makePayment` updates both `self.paid` and `self.outstanding`.

The method `getTotalPaid` uses the built-in Python function `sum`, which returns the sum of a sequence of numbers. If the sequence contains a non-number, an exception is raised.

```
class Fixed(Mortgage):
    def __init__(self, loan, r, months):
        Mortgage.__init__(self, loan, r, months)
        self.legend = 'Fixed, ' + str(round(r*100, 2)) + '%'

class FixedWithPts(Mortgage):
    def __init__(self, loan, r, months, pts):
        Mortgage.__init__(self, loan, r, months)
        self.pts = pts
        self.paid = [loan*(pts/100)]
        self.legend = 'Fixed, ' + str(round(r*100, 2)) + '%, '\
                      + str(pts) + ' points'

class TwoRate(Mortgage):
    def __init__(self, loan, r, months, teaserRate, teaserMonths):
        Mortgage.__init__(self, loan, teaserRate, months)
        self.teaserMonths = teaserMonths
        self.teaserRate = teaserRate
        self.nextRate = r/12
        self.legend = str(teaserRate*100)\
                     + '% for ' + str(self.teaserMonths)\
                     + ' months, then ' + str(round(r*100, 2)) + '%'
    def makePayment(self):
        if len(self.paid) == self.teaserMonths + 1:
            self.rate = self.nextRate
            self.payment = findPayment(self.outstanding[-1],
                                        self.rate,
                                        self.months - self.teaserMonths)
        Mortgage.makePayment(self)
```

Figure 8.10 Mortgage subclasses

Figure 8.10 contains classes implementing three types of mortgages. The classes `Fixed` and `FixedWithPts` override `__init__` and inherit the other three methods from `Mortgage`. The class `TwoRate` treats a mortgage as the concatenation of two loans, each at a different interest rate. (Since `self.paid` is initialized to a list with one element, it contains one more element than the number of payments that have been made. That's why the method `makePayment` compares `len(self.paid)` to `self.teaserMonths + 1`.)

Figure 8.11 contains a function that computes and prints the total cost of each kind of mortgage for a sample set of parameters. It begins by creating one mortgage of each kind. It then makes a monthly payment on each for a given number of years. Finally, it prints the total amount of the payments made for each loan.

Notice that we used keyword rather than positional arguments in the invocation of `compareMortgages`. We did this because `compareMortgages` has a large number of formal parameters and using keyword arguments makes it easier to ensure that we are supplying the intended actual values to each of the formals.

```
def compareMortgages(amt, years, fixedRate, pts, ptsRate,
                     varRate1, varRate2, varMonths):
    totMonths = years*12
    fixed1 = Fixed(amt, fixedRate, totMonths)
    fixed2 = FixedWithPts(amt, ptsRate, totMonths, pts)
    twoRate = TwoRate(amt, varRate2, totMonths, varRate1, varMonths)
    morts = [fixed1, fixed2, twoRate]
    for m in range(totMonths):
        for mort in morts:
            mort.makePayment()
    for m in morts:
        print(m)
        print(' Total payments = $' + str(int(m.getTotalPaid())))

compareMortgages(amt=200000, years=30, fixedRate=0.07,
                 pts = 3.25, ptsRate=0.05, varRate1=0.045,
                 varRate2=0.095, varMonths=48)
```

Figure 8.11 Evaluate mortgages

When the code in Figure 8.11 is run, it prints

```
Fixed, 7.0%
 Total payments = $479017
Fixed, 5.0%, 3.25 points
 Total payments = $393011
4.5% for 48 months, then 9.5%
 Total payments = $551444
```

At first glance, the results look pretty conclusive. The variable-rate loan is a bad idea (for the borrower, not the lender) and the fixed-rate loan with points costs the least. It's important to note, however, that total cost is not the only metric by which mortgages should be judged. For example, a borrower who expects to have a higher income in the future may be willing to pay more in the later years to lessen the burden of payments in the beginning.

This suggests that rather than looking at a single number, we should look at payments over time. This in turn suggests that our program should be producing plots designed to show how the mortgage behaves over time. We will do that in Section 11.2.

9 A SIMPLISTIC INTRODUCTION TO ALGORITHMIC COMPLEXITY

The most important thing to think about when designing and implementing a program is that it should produce results that can be relied upon. We want our bank balances to be calculated correctly. We want the fuel injectors in our automobiles to inject appropriate amounts of fuel. We would prefer that neither airplanes nor operating systems crash.

Sometimes performance is an important aspect of correctness. This is most obvious for programs that need to run in real time. A program that warns airplanes of potential obstructions needs to issue the warning before the obstructions are encountered. Performance can also affect the utility of many non-real-time programs. The number of transactions completed per minute is an important metric when evaluating the utility of database systems. Users care about the time required to start an application on their phone. Biologists care about how long their phylogenetic inference calculations take.

Writing efficient programs is not easy. The most straightforward solution is often not the most efficient. Computationally efficient algorithms often employ subtle tricks that can make them difficult to understand. Consequently, programmers often increase the **conceptual complexity** of a program in an effort to reduce its **computational complexity**. To do this in a sensible way, we need to understand how to go about estimating the computational complexity of a program. That is the topic of this chapter.

9.1 Thinking About Computational Complexity

How should one go about answering the question “How long will the following function take to run?”

```
def f(i):
   """Assumes i is an int and i >= 0"""
   answer = 1
   while i >= 1:
      answer *= i
      i -= 1
   return answer
```

We could run the program on some input and time it. But that wouldn't be particularly informative because the result would depend upon

- the speed of the computer on which it is run,
- the efficiency of the Python implementation on that machine, and
- the value of the input.

We get around the first two issues by using a more abstract measure of time. Instead of measuring time in milliseconds, we measure time in terms of the number of basic steps executed by the program.

For simplicity, we will use a **random access machine** as our model of computation. In a random access machine, steps are executed sequentially, one at a time.[53] A **step** is an operation that takes a fixed amount of time, such as binding a variable to an object, making a comparison, executing an arithmetic operation, or accessing an object in memory.

Now that we have a more abstract way to think about the meaning of time, we turn to the question of dependence on the value of the input. We deal with that by moving away from expressing time complexity as a single number and instead relating it to the sizes of the inputs. This allows us to compare the efficiency of two algorithms by talking about how the running time of each grows with respect to the sizes of the inputs.

Of course, the actual running time of an algorithm depends not only upon the sizes of the inputs but also upon their values. Consider, for example, the linear search algorithm implemented by

```
def linearSearch(L, x):
   for e in L:
      if e == x:
         return True
   return False
```

Suppose that `L` is a list containing a million elements, and consider the call `linearSearch(L, 3)`. If the first element in `L` is `3`, `linearSearch` will return `True` almost immediately. On the other hand, if `3` is not in `L`, `linearSearch` will have to examine all one million elements before returning `False`.

In general, there are three broad cases to think about:

- The best-case running time is the running time of the algorithm when the inputs are as favorable as possible. I.e., the **best-case** running time is the mini-

[53] A more accurate model for today's computers might be a parallel random access machine. However, that adds considerable complexity to the algorithmic analysis, and often doesn't make an important qualitative difference in the answer.

mum running time over all the possible inputs of a given size. For `linearSearch`, the best-case running time is independent of the size of `L`.

- Similarly, the **worst-case** running time is the maximum running time over all the possible inputs of a given size. For `linearSearch`, the worst-case running time is linear in the size of `L`.
- By analogy with the definitions of the best-case and worst-case running time, the **average-case** (also called **expected-case**) running time is the average running time over all possible inputs of a given size. Alternatively, if one has some *a priori* information about the distribution of input values (e.g., that 90% of the time `x` is in `L`), one can take that into account.

People usually focus on the worst case. All engineers share a common article of faith, Murphy's Law: If something can go wrong, it will go wrong. The worst-case provides an **upper bound** on the running time. This is critical in situations where there is a time constraint on how long a computation can take. It is not good enough to know that "most of the time" the air traffic control system warns of impending collisions before they occur.

Let's look at the worst-case running time of an iterative implementation of the factorial function:

```
def fact(n):
    """Assumes n is a natural number
       Returns n!"""
    answer = 1
    while n > 1:
       answer *=  n
       n -= 1
    return answer
```

The number of steps required to run this program is something like 2 (1 for the initial assignment statement and 1 for the `return`) + 5n (counting 1 step for the test in the `while`, 2 steps for the first assignment statement in the `while` loop, and 2 steps for the second assignment statement in the loop). So, for example, if `n` is 1000, the function will execute roughly 5002 steps.

It should be immediately obvious that as `n` gets large, worrying about the difference between 5n and 5n+2 is kind of silly. For this reason, we typically ignore additive constants when reasoning about running time. Multiplicative constants are more problematical. Should we care whether the computation takes 1000 steps or 5000 steps? Multiplicative factors can be important. Whether a search engine takes a half second or 2.5 seconds to service a query can be the difference between whether people use that search engine or go to a competitor.

On the other hand, when one is comparing two different algorithms, it is often the case that even multiplicative constants are irrelevant. Recall that in Chapter 3 we looked at two algorithms, exhaustive enumeration and bisection search, for finding an approximation to the square root of a floating point number. Functions based on these algorithms are shown in Figure 9.1 and Figure 9.2.

```
def squareRootExhaustive(x, epsilon):
   """Assumes x and epsilon are positive floats & epsilon < 1
      Returns a y such that y*y is within epsilon of x"""
   step = epsilon**2
   ans = 0.0
   while abs(ans**2 - x) >= epsilon and ans*ans <= x:
       ans += step
   if ans*ans > x:
      raise ValueError
   return ans
```

Figure 9.1 Using exhaustive enumeration to approximate square root

```
def squareRootBi(x, epsilon):
   """Assumes x and epsilon are positive floats & epsilon < 1
      Returns a y such that y*y is within epsilon of x"""
   low = 0.0
   high = max(1.0, x)
   ans = (high + low)/2.0
   while abs(ans**2 - x) >= epsilon:
      if ans**2 < x:
         low = ans
      else:
         high = ans
      ans = (high + low)/2.0
   return ans
```

Figure 9.2 Using bisection search to approximate square root

We saw that exhaustive enumeration was so slow as to be impractical for many combinations of `x` and `epsilon`. For example, evaluating `squareRootExhaustive(100, 0.0001)` requires roughly one billion iterations of the `while` loop. In contrast, evaluating `squareRootBi(100, 0.0001)` takes roughly twenty iterations of a slightly more complex `while` loop. When the difference in the number of iterations is this large, it doesn't really matter how many instructions are in the loop. I.e., the multiplicative constants are irrelevant.

9.2 Asymptotic Notation

We use something called **asymptotic notation** to provide a formal way to talk about the relationship between the running time of an algorithm and the size of its inputs. The underlying motivation is that almost any algorithm is sufficiently efficient when run on small inputs. What we typically need to worry about is the efficiency of the algorithm when run on very large inputs. As a proxy for "very large," asymptotic notation describes the complexity of an algorithm as the size of its inputs approaches infinity.

Consider, for example, the code in Figure 9.3.

```
def f(x):
   """Assume x is an int > 0"""
   ans = 0
   #Loop that takes constant time
   for i in range(1000):
      ans += 1
   print('Number of additions so far', ans)
   #Loop that takes time x
   for i in range(x):
      ans += 1
   print('Number of additions so far', ans)
   #Nested loops take time x**2
   for i in range(x):
      for j in range(x):
         ans += 1
         ans += 1
   print('Number of additions so far', ans)
   return ans
```

Figure 9.3 Asymptotic complexity

If one assumes that each line of code takes one unit of time to execute, the running time of this function can be described as $1000 + x + 2x^2$. The constant 1000 corresponds to the number of times the first loop is executed. The term x corresponds to the number of times the second loop is executed. Finally, the term $2x^2$ corresponds to the time spent executing the two statements in the nested `for` loop. Consequently, the call `f(10)` will print

```
Number of additions so far 1000
Number of additions so far 1010
Number of additions so far 1210
```

and the call `f(1000)` will print

```
Number of additions so far 1000
Number of additions so far 2000
Number of additions so far 2002000
```

For small values of `x` the constant term dominates. If `x` is 10, over 80% of the steps are accounted for by the first loop. On the other hand, if `x` is 1000, each of the first two loops accounts for only about 0.05% of the steps. When `x` is 1,000,000, the first loop takes about 0.00000005% of the total time and the second loop about 0.00005%. A full 2,000,000,000,000 of the 2,000,001,001,000 steps are in the body of the inner `for` loop.

Clearly, we can get a meaningful notion of how long this code will take to run on very large inputs by considering only the inner loop, i.e., the quadratic component. Should we care about the fact that this loop takes $2x^2$ steps rather than x^2 steps? If your computer executes roughly 100 million steps per second, evaluating `f` will take about 5.5 hours. If we could reduce the complexity to x^2 steps, it would take about 2.25 hours. In either case, the moral is the same: we should probably look for a more efficient algorithm.

This kind of analysis leads us to use the following rules of thumb in describing the asymptotic complexity of an algorithm:

- If the running time is the sum of multiple terms, keep the one with the largest growth rate, and drop the others.
- If the remaining term is a product, drop any constants.

The most commonly used asymptotic notation is called "**Big O**" notation.[54] Big O notation is used to give an **upper bound** on the asymptotic growth (often called the **order of growth**) of a function. For example, the formula $f(x) \in O(x^2)$ means that the function f grows no faster than the quadratic polynomial x^2, in an asymptotic sense.

We, like many computer scientists, will often abuse Big O notation by making statements like, "the complexity of f(x) is $O(x^2)$." By this we mean that in the worst case f will take $O(x^2)$ steps to run. The difference between a function being "in $O(x^2)$" and "being $O(x^2)$" is subtle but important. Saying that $f(x) \in O(x^2)$ does

[54] The phrase "Big O" was introduced in this context by the computer scientist Donald Knuth in the 1970s. He chose the Greek letter Omicron because number theorists had used that letter since the late 19th century to denote a related concept.

not preclude the worst-case running time of f from being considerably less than $O(x^2)$.

When we say that f(x) is $O(x^2)$, we are implying that x^2 is both an upper and a **lower bound** on the asymptotic worst-case running time. This is called a **tight bound.**[55]

9.3 Some Important Complexity Classes

Some of the most common instances of Big O are listed below. In each case, n is a measure of the size of the inputs to the function.

- O(1) denotes **constant** running time.
- O(log n) denotes **logarithmic** running time.
- O(n) denotes **linear** running time.
- O(n log n) denotes **log-linear** running time.
- $O(n^k)$ denotes **polynomial** running time. Notice that k is a constant.
- $O(c^n)$ denotes **exponential** running time. Here a constant is being raised to a power based on the size of the input.

9.3.1 Constant Complexity

This indicates that the asymptotic complexity is independent of the size of the inputs. There are very few interesting programs in this class, but all programs have pieces (for example finding out the length of a Python list or multiplying two floating point numbers) that fit into this class. Constant running time does not imply that there are no loops or recursive calls in the code, but it does imply that the number of iterations or recursive calls is independent of the size of the inputs.

9.3.2 Logarithmic Complexity

Such functions have a complexity that grows as the log of at least one of the inputs. Binary search, for example, is logarithmic in the length of the list being searched. (We will look at binary search and analyze its complexity in Chapter 10.) By the way, we don't care about the base of the log, since the difference between using one base and another is merely a constant multiplicative factor. For

[55] The more pedantic members of the computer science community use Big Theta, Θ, rather than Big O for this.

example, $O(\log_2(x)) = O(\log_2(10)*\log_{10}(x))$. There are lots of interesting functions with logarithmic complexity. Consider

```
def intToStr(i):
   """Assumes i is a nonnegative int
      Returns a decimal string representation of i"""
   digits = '0123456789'
   if i == 0:
      return '0'
   result = ''
   while i > 0:
      result = digits[i%10] + result
      i = i//10
   return result
```

Since there are no function or method calls in this code, we know that we only have to look at the loops to determine the complexity class. There is only one loop, so the only thing that we need to do is characterize the number of iterations. That boils down to the number of times we can use integer division to divide `i` by 10 before getting a result of 0. So, the complexity of `intToStr` is O(log(i)).

What about the complexity of

```
def addDigits(n):
   """Assumes n is a nonnegative int
      Returns the sum of the digits in n"""
   stringRep = intToStr(n)
   val = 0
   for c in stringRep:
      val += int(c)
   return val
```

The complexity of converting `n` to a string using `intToStr` is O(log(n)), and `intToStr` returns a string of length O(log(n)). The `for` loop will be executed O(len(stringRep)) times, i.e., O(log(n)) times. Putting it all together, and assuming that a character representing a digit can be converted to an integer in constant time, the program will run in time proportional to O(log(n)) + O(log(n)), which makes it O(log(n)).

9.3.3 Linear Complexity

Many algorithms that deal with lists or other kinds of sequences are linear because they touch each element of the sequence a constant (greater than 0) number of times.

Consider, for example,

```
def addDigits(s):
   """Assumes s is a str each character of which is a
         decimal digit.
      Returns an int that is the sum of the digits in s"""
   val = 0
   for c in s:
      val += int(c)
   return val
```

This function is linear in the length of s, i.e., O(len(s))—again assuming that a character representing a digit can be converted to an integer in constant time.

Of course, a program does not need to have a loop to have linear complexity. Consider

```
def factorial(x):
   """Assumes that x is a positive int
      Returns x!"""
   if x == 1:
      return 1
   else:
      return x*factorial(x-1)
```

There are no loops in this code, so in order to analyze the complexity we need to figure out how many recursive calls get made. The series of calls is simply

```
factorial(x), factorial(x-1), factorial(x-2), ... , factorial(1)
```

The length of this series, and thus the complexity of the function, is O(x).

Thus far in this chapter we have looked only at the time complexity of our code. This is fine for algorithms that use a constant amount of space, but this implementation of factorial does not have that property. As we discussed in Chapter 4, each recursive call of `factorial` causes a new stack frame to be allocated, and that frame continues to occupy memory until the call returns. At the maximum depth of recursion, this code will have allocated x stack frames, so the space complexity is also O(x).

The impact of space complexity is harder to appreciate than the impact of time complexity. Whether a program takes one minute or two minutes to complete is quite visible to its user, but whether it uses one megabyte or two megabytes of memory is largely invisible to users. This is why people typically give more attention to time complexity than to space complexity. The exception occurs when a program needs more space than is available in the fast memory of the machine on which it is run.

9.3.4 Log-Linear Complexity

This is slightly more complicated than the complexity classes we have looked at thus far. It involves the product of two terms, each of which depends upon the size of the inputs. It is an important class, because many practical algorithms are log-linear. The most commonly used log-linear algorithm is probably merge sort, which is O(n log(n)), where n is the length of the list being sorted. We will look at that algorithm and analyze its complexity in Chapter 10.

9.3.5 Polynomial Complexity

The most commonly used polynomial algorithms are **quadratic**, i.e., their complexity grows as the square of the size of their input. Consider, for example, the function in Figure 9.4, which implements a subset test.

```
def isSubset(L1, L2):
   """Assumes L1 and L2 are lists.
      Returns True if each element in L1 is also in L2
      and False otherwise."""
   for e1 in L1:
      matched = False
      for e2 in L2:
         if e1 == e2:
            matched = True
            break
      if not matched:
         return False
   return True
```

Figure 9.4 Implementation of subset test

Each time the inner loop is reached it is executed O(len(L2) times. The function `isSubset` will execute the outer loop O(len(L1)) times, so the inner loop will be reached O(len(L1)) times. Therefore, the complexity of the function `isSubset` is O(len(L1)*len(L2)).

Now consider the function `intersect` in Figure 9.5. The running time for the part building the list that might contain duplicates is clearly O(len(L1)*len(L2)). At first glance, it appears that the part of the code that builds the duplicate-free list is linear in the length of `tmp`, but it is not. The test `e not in result` potentially involves looking at each element in `result`, and is therefore O(len(result)); consequently the second part of the implementation is O(len(tmp)*len(result)). However, since the lengths of `result` and `tmp` are bounded by the length of the smaller of L1 and L2, and since we ignore additive terms, the complexity of intersect is O(len(L1)*len(L2)).

```
def intersect(L1, L2):
   """Assumes: L1 and L2 are lists
      Returns a list without duplicates that is the intersection of
      L1 and L2"""
   #Build a list containing common elements
   tmp = []
   for e1 in L1:
      for e2 in L2:
         if e1 == e2:
            tmp.append(e1)
            break
   #Build a list without duplicates
   result = []
   for e in tmp:
      if e not in result:
         result.append(e)
   return result
```

Figure 9.5 Implementation of list intersection

The running time for the part building the list that might contain duplicates is clearly O(len(L1)*len(L2)). At first glance, it appears that the part of the code that builds the duplicate-free list is linear in the length of tmp, but it is not. The test e not in result potentially involves looking at each element in result, and is therefore O(len(result)); consequently the second part of the implementation is O(len(tmp)*len(result)). However, since the lengths of result and tmp are bounded by the length of the smaller of L1 and L2, and since we ignore additive terms, the complexity of intersect is O(len(L1)*len(L2)).

9.3.6 Exponential Complexity

As we will see later in this book, many important problems are inherently exponential, i.e., solving them completely can require time that is exponential in the size of the input. This is unfortunate, since it rarely pays to write a program that has a reasonably high probability of taking exponential time to run. Consider, for example, the code in Figure 9.6.

```
def getBinaryRep(n, numDigits):
   """Assumes n and numDigits are non-negative ints
      Returns a str of length numDigits that is a binary
      representation of n"""
   result = ''
   while n > 0:
      result = str(n%2) + result
      n = n//2
   if len(result) > numDigits:
      raise ValueError('not enough digits')
   for i in range(numDigits - len(result)):
      result = '0' + result
   return result

def genPowerset(L):
   """Assumes L is a list
      Returns a list of lists that contains all possible
      combinations of the elements of L. E.g., if
      L is [1, 2] it will return a list with elements
      [], [1], [2], and [1,2]."""
   powerset = []
   for i in range(0, 2**len(L)):
      binStr = getBinaryRep(i, len(L))
      subset = []
      for j in range(len(L)):
         if binStr[j] == '1':
            subset.append(L[j])
      powerset.append(subset)
   return powerset
```

Figure 9.6 Generating the power set

The function `genPowerset(L)` returns a list of lists that contains all possible combinations of the elements of `L`. For example, if `L` is `['x', 'y']`, the powerset of `L` will be a list containing the lists `[]`, `['x']`, `['y']`, and `['x', 'y']`.

The algorithm is a bit subtle. Consider a list of n elements. We can represent any combination of elements by a string of n 0's and 1's, where a 1 represents the presence of an element and a 0 its absence. The combination containing no items is represented by a string of all 0's, the combination containing all of the items is represented by a string of all 1's, the combination containing only the first and last elements is represented by 100…001, etc.

Generating all sublists of a list `L` of length `n` can be done as follows:

- Generate all `n`-bit binary numbers. These are the numbers from `0` to 2^n.
- For each of these $2^n + 1$ binary numbers, `b`, generate a list by selecting those elements of `L` that have an index corresponding to a `1` in `b`. For example, if `L` is `['x', 'y']` and `b` is `01`, generate the list `['y']`.

Try running `genPowerset` on a list containing the first ten letters of the alphabet. It will finish quite quickly and produce a list with 1024 elements. Next, try running `genPowerset` on the first twenty letters of the alphabet. It will take more than a bit of time to run, and return a list with about a million elements. If you try running `genPowerset` on all twenty-six letters, you will probably get tired of waiting for it to complete, unless your computer runs out of memory trying to build a list with tens of millions of elements. Don't even think about trying to run `genPowerset` on a list containing all uppercase and lowercase letters. Step 1 of the algorithm generates $O(2^{len(L)})$ binary numbers, so the algorithm is exponential in len(L).

Does this mean that we cannot use computation to tackle exponentially hard problems? Absolutely not. It means that we have to find algorithms that provide approximate solutions to these problems or that find perfect solutions on some instances of the problem. But that is a subject for later chapters.

9.3.7 Comparisons of Complexity Classes

The plots in this section are intended to convey an impression of the implications of an algorithm being in one or another of these complexity classes.

The plot on the left in Figure 9.7 compares the growth of a constant-time algorithm to that of a logarithmic algorithm. Note that the size of the input has to reach about a million for the two of them to cross, even for the very small constant of twenty. When the size of the input is five million, the time required by a logarithmic algorithm is still quite small. The moral is that logarithmic algorithms are almost as good as constant-time ones.

The plot on the right of Figure 9.7 illustrates the dramatic difference between logarithmic algorithms and linear algorithms. Notice that the x-axis only goes as high as 1000. While we needed to look at large inputs to appreciate the difference between constant-time and logarithmic-time algorithms, the difference between logarithmic-time and linear-time algorithms is apparent even on small inputs. The dramatic difference in the relative performance of logarithmic and linear algorithms does not mean that linear algorithms are bad. In fact, most of the time a linear algorithm is acceptably efficient.

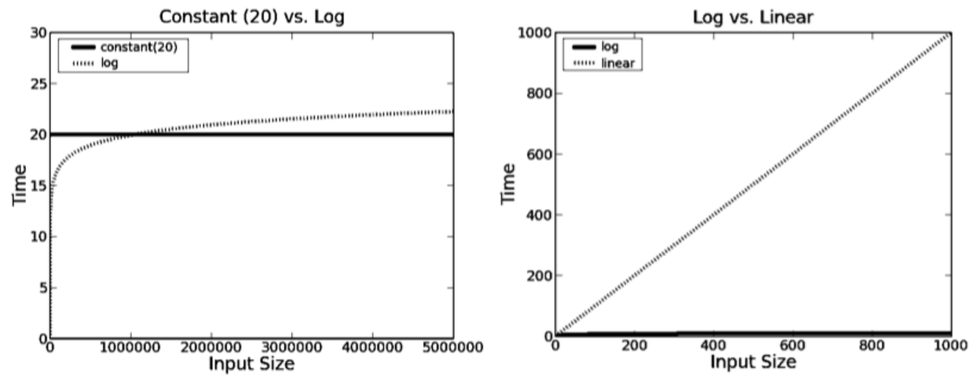


Figure 9.7 Constant, logarithmic, and linear growth

The plot on the left in Figure 9.8 shows that there is a significant difference between O(n) and O(n log(n)). Given how slowly log(n) grows, this may seem a bit surprising, but keep in mind that it is a multiplicative factor. Also keep in mind that in many practical situations, O(n log(n)) is fast enough to be useful. On the other hand, as the plot on the right in Figure 9.8 suggests, there are many situations in which a quadratic rate of growth is prohibitive.

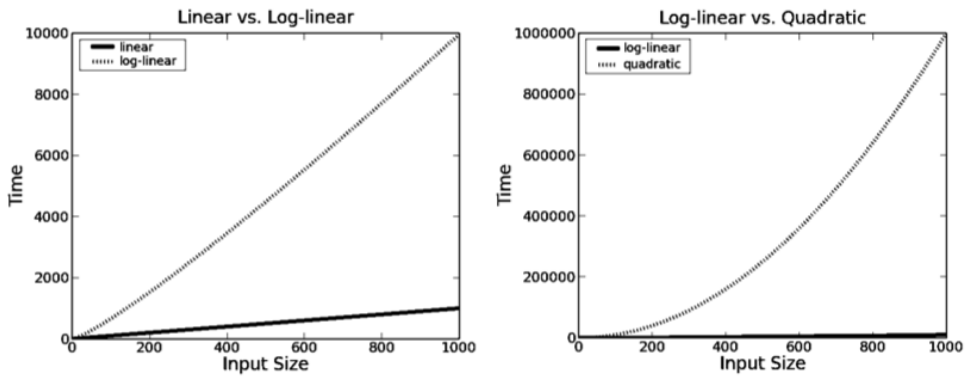


Figure 9.8 Linear, log-linear, and quadratic growth

The plots in Figure 9.9 are about exponential complexity. In the plot on the left of Figure 9.9, the numbers to the left of the y-axis run from 0.0 to 1.2. However, the notation `x1e301` on the top left means that each tick on the y-axis should be multiplied by 10^{301}. So, the plotted y-values range from 0 to roughly $1.1*10^{301}$.

But it looks almost as if there are no curves in the plot on the left in Figure 9.9. That's because an exponential function grows so quickly that relative to the y value of the highest point (which determines the scale of the y-axis), the y values of earlier points on the exponential curve (and all points on the quadratic curve) are almost indistinguishable from 0.

The plot on the right in Figure 9.9 addresses this issue by using a logarithmic scale on the y-axis. One can readily see that exponential algorithms are impractical for all but the smallest of inputs.

Notice that when plotted on a logarithmic scale, an exponential curve appears as a straight line. We will have more to say about this in later chapters.

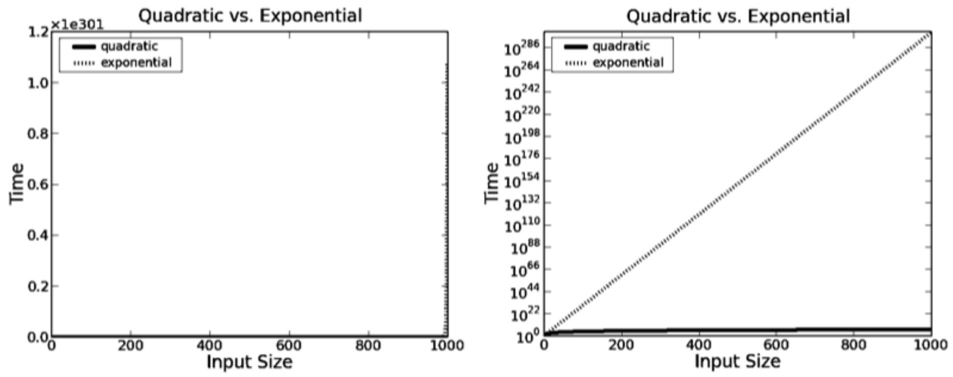


Figure 9.9 Quadratic and exponential growth