

---

# **Introduction to Computation and Programming Using Python**

---

**With Application to Computational  
Modeling and Understanding Data**

Third Edition

**John V. Guttag**

# **Introduction to Computation and Programming Using Python**

**with Application to Computational Modeling  
and Understanding Data**

# **Introduction to Computation and Programming Using Python**

**with Application to Computational Modeling  
and Understanding Data**

third edition

John V. Guttag

The MIT Press  
Cambridge, Massachusetts  
London, England

© 2021 Massachusetts Institute of Technology

All rights reserved. No part of this book may be reproduced in any form by any electronic or mechanical means (including photocopying, recording, or information storage and retrieval) without permission in writing from the publisher.

This book was set in Minion Pro by New Best-set Typesetters Ltd.

Library of Congress Cataloging-in-Publication Data

Names: Guttag, John, author.

Title: Introduction to computation and programming using Python : with application to computational modeling and understanding data / John V. Guttag.

Description: Third edition. | Cambridge, Massachusetts : The MIT Press, [2021] | Includes index.

Identifiers: LCCN 2020036760 | ISBN 9780262542364 (paperback)

Subjects: LCSH: Python (Computer program language)—Textbooks. | Computer programming—Textbooks.

Classification: LCC QA76.73.P98 G88 2021 | DDC 005.13/3—dc23

LC record available at <https://lccn.loc.gov/2020036760>

10 9 8 7 6 5 4 3 2 1

d\_ro

*To my family:*

Olga  
David  
Andrea  
Michael  
Mark  
Addie  
Pierce

# CONTENTS

[PREFACE](#)

[ACKNOWLEDGMENTS](#)

[1: GETTING STARTED](#)

[2: INTRODUCTION TO PYTHON](#)

[3: SOME SIMPLE NUMERICAL PROGRAMS](#)

[4: FUNCTIONS, SCOPING, AND ABSTRACTION](#)

[5: STRUCTURED TYPES AND MUTABILITY](#)

[6: RECURSION AND GLOBAL VARIABLES](#)

[7: MODULES AND FILES](#)

[8: TESTING AND DEBUGGING](#)

[9: EXCEPTIONS AND ASSERTIONS](#)

[10: CLASSES AND OBJECT-ORIENTED PROGRAMMING](#)

[11: A SIMPLISTIC INTRODUCTION TO ALGORITHMIC COMPLEXITY](#)

[12: SOME SIMPLE ALGORITHMS AND DATA STRUCTURES](#)

[13: PLOTTING AND MORE ABOUT CLASSES](#)

[14: KNAPSACK AND GRAPH OPTIMIZATION PROBLEMS](#)

[15: DYNAMIC PROGRAMMING](#)

[16: RANDOM WALKS AND MORE ABOUT DATA VISUALIZATION](#)

[17: STOCHASTIC PROGRAMS, PROBABILITY, AND DISTRIBUTIONS](#)

[18: MONTE CARLO SIMULATION](#)  
[19: SAMPLING AND CONFIDENCE](#)  
[20: UNDERSTANDING EXPERIMENTAL DATA](#)  
[21: RANDOMIZED TRIALS AND HYPOTHESIS CHECKING](#)  
[22: LIES, DAMNED LIES, AND STATISTICS](#)  
[23: EXPLORING DATA WITH PANDAS](#)  
[24: A QUICK LOOK AT MACHINE LEARNING](#)  
[25: CLUSTERING](#)  
[26: CLASSIFICATION METHODS](#)  
[PYTHON 3.8 QUICK REFERENCE](#)  
[INDEX](#)

## List of figures

Chapter 1

[Figure 1-1 Flowchart of getting dinner](#)

Chapter 2

[Figure 2-1 Anaconda startup window](#)

[Figure 2-2 Spyder window](#)

[Figure 2-3 Operators on types int and float](#)

[Figure 2-4 Binding of variables to objects](#)

[Figure 2-5 Flowchart for conditional statement](#)

[Figure 2-6 Flowchart for iteration](#)

[Figure 2-7 Squaring an integer, the hard way.](#)

[Figure 2-8 Hand simulation of a small program](#)

[Figure 2-9 Using a for statement](#)

Chapter 3

[Figure 3-1 Using exhaustive enumeration to find the cube root](#)

[Figure 3-2 Using exhaustive enumeration to test primality](#)

[Figure 3-3 A more efficient primality test](#)

[Figure 3-4 Approximating the square root using exhaustive enumeration](#)

[Figure 3-5 Using bisection search to approximate square root](#)

[Figure 3-6 Using bisection search to estimate log base 2](#)

[Figure 3-7 Implementation of Newton–Raphson method](#)

## Chapter 4

[Figure 4-1 Using bisection search to approximate square root of x](#)

[Figure 4-2 Summing a square root and a cube root](#)

[Figure 4-3 A function for finding roots](#)

[Figure 4-4 Code to test find\\_root](#)

[Figure 4-5 Nested scopes](#)

[Figure 4-6 Stack frames](#)

[Figure 4-7 A function definition with a specification](#)

[Figure 4-8 Splitting find\\_root into multiple functions](#)

[Figure 4-9 Generalizing bisection\\_solve](#)

[Figure 4-10 Using bisection\\_solve to approximate logs](#)

## Chapter 5

[Figure 5-1 Two lists](#)

[Figure 5-2 Two lists that appear to have the same value, but don't](#)

[Figure 5-3 Demonstration of mutability](#)

[Figure 5-4 Common methods associated with lists](#)

[Figure 5-5 Applying a function to elements of a list](#)

[Figure 5-6 Common operations on sequence types](#)

[Figure 5-7 Comparison of sequence types](#)

[Figure 5-8 Some methods on strings](#)

[Figure 5-9 Translating text \(badly\)](#)

[Figure 5-10 Some common operations on dicts](#)

Chapter 6

[Figure 6-1 Iterative and recursive implementations of factorial](#)

[Figure 6-2 Growth in population of female rabbits](#)

[Figure 6-3 Recursive implementation of Fibonacci sequence](#)

[Figure 6-4 Palindrome testing](#)

[Figure 6-5 Code to visualize palindrome testing](#)

[Figure 6-6 Using a global variable](#)

Chapter 7

[Figure 7-1 Some code related to circles and spheres](#)

[Figure 7-2 Common functions for accessing files](#)

Chapter 8

[Figure 8-1 Testing boundary conditions](#)

[Figure 8-2 Not the first bug](#)

[Figure 8-3 Program with bugs](#)

Chapter 9

[Figure 9-1 Using exceptions for control flow](#)

[Figure 9-2 Control flow without a try-except](#)

[Figure 9-3 Get grades](#)

Chapter 10

[Figure 10-1 Class Int\\_set](#)

[Figure 10-2 Using magic methods](#)

[Figure 10-3 Class Person](#)

[Figure 10-4 Class MIT\\_person](#)

[Figure 10-5 Two kinds of students](#)

[Figure 10-6 Class Grades](#)

[Figure 10-7 Generating a grade report](#)

[Figure 10-8 Information hiding in classes](#)

[Figure 10-9 New version of get\\_students](#)

[Figure 10-10 Mortgage base class](#)

[Figure 10-11 Mortgage subclasses](#)

Chapter 11

[Figure 11-1 Using exhaustive enumeration to approximate square root](#)

[Figure 11-2 Using bisection search to approximate square root](#)

[Figure 11-3 Asymptotic complexity](#)

[Figure 11-4 Implementation of subset test](#)

[Figure 11-5 Implementation of list intersection](#)

[Figure 11-6 Generating the power set](#)

[Figure 11-7 Constant, logarithmic, and linear growth](#)

[Figure 11-8 Linear, log-linear, and quadratic growth](#)

[Figure 11-9 Quadratic and exponential growth](#)

Chapter 12

[Figure 12-1 Implementing lists](#)

[Figure 12-2 Linear search of a sorted list](#)

[Figure 12-3 Recursive binary search](#)

[Figure 12-4 Selection sort](#)

[Figure 12-5 Merge sort](#)

[Figure 12-6 Sorting a list of names](#)

[Figure 12-7 Implementing dictionaries using hashing](#)

Chapter 13

- [Figure 13-1 A simple plot](#)
- [Figure 13-2 Contents of Figure-Jane.png \(left\) and Figure-Addie.png \(right\)](#)
- [Figure 13-3 Produce plots showing compound growth](#)
- [Figure 13-4 Plots showing compound growth](#)
- [Figure 13-5 Another plot of compound growth](#)
- [Figure 13-6 Strange-looking plot](#)
- [Figure 13-7 Class Mortgage with plotting methods](#)
- [Figure 13-8 Subclasses of Mortgage](#)
- [Figure 13-9 Compare mortgages](#)
- [Figure 13-10 Generate mortgage plots](#)
- [Figure 13-11 Monthly payments of different kinds of mortgages](#)
- [Figure 13-12 Cost over time of different kinds of mortgages](#)
- [Figure 13-13 Balance remaining and net cost for different kinds of mortgages](#)
- [Figure 13-14 Simulation of spread of an infectious disease](#)
- [Figure 13-15 Function to plot history of infection](#)
- [Figure 13-16 Produce plot with a single set of parameters](#)
- [Figure 13-17 Static plot of number of infections](#)
- [Figure 13-18 Interactive plot with initial slider values](#)
- [Figure 13-19 Interactive plot with changed slider values](#)
- Chapter 14
- [Figure 14-1 Table of items](#)
- [Figure 14-2 Class Item](#)
- [Figure 14-3 Implementation of a greedy algorithm](#)
- [Figure 14-4 Using a greedy algorithm to choose items](#)
- [Figure 14-5 Brute-force optimal solution to the 0/1 knapsack problem](#)

[Figure 14-6 The bridges of Königsberg \(left\) and Euler's simplified map \(right\)](#)

[Figure 14-7 Nodes and edges](#)

[Figure 14-8 Classes Graph and Digraph](#)

[Figure 14-9 Depth-first-search shortest-path algorithm](#)

[Figure 14-10 Test depth-first-search code](#)

[Figure 14-11 Breadth-first-search shortest path algorithm](#)

Chapter 15

[Figure 15-1 Tree of calls for recursive Fibonacci](#)

[Figure 15-2 Implementing Fibonacci using a memo](#)

[Figure 15-3 Table of items with values and weights](#)

[Figure 15-4 Decision tree for knapsack problem](#)

[Figure 15-5 Using a decision tree to solve a knapsack problem](#)

[Figure 15-6 Testing the decision tree-based implementation](#)

[Figure 15-7 Dynamic programming solution to knapsack problem](#)

[Figure 15-8 Performance of dynamic programming solution](#)

Chapter 16

[Figure 16-1 An unusual farmer](#)

[Figure 16-2 Location and Field classes](#)

[Figure 16-3 Classes defining Drunks](#)

[Figure 16-4 The drunkard's walk \(with a bug\).](#)

[Figure 16-5 Distance from starting point versus steps taken](#)

[Figure 16-6 Subclasses of Drunk base class](#)

[Figure 16-7 Iterating over styles](#)

[Figure 16-8 Plotting the walks of different drunks](#)

[Figure 16-9 Mean distance for different kinds of drunks](#)

[Figure 16-10 Plotting final locations](#)

[Figure 16-11 Where the drunk stops](#)

[Figure 16-12 Tracing walks](#)

[Figure 16-13 Trajectory of walks](#)

[Figure 16-14 Fields with strange properties](#)

[Figure 16-15 A strange walk](#)

Chapter 17

[Figure 17-1 Roll die](#)

[Figure 17-2 Flipping a coin](#)

[Figure 17-3 Regression to the mean](#)

[Figure 17-4 Illustration of regression to mean](#)

[Figure 17-5 Plotting the results of coin flips](#)

[Figure 17-6 The law of large numbers at work](#)

[Figure 17-7 The law of large numbers at work](#)

[Figure 17-8 Variance and standard deviation](#)

[Figure 17-9 Helper function for coin-flipping simulation](#)

[Figure 17-10 Coin-flipping simulation](#)

[Figure 17-11 Convergence of heads/tails ratios](#)

[Figure 17-12 Absolute differences](#)

[Figure 17-13 Mean and standard deviation of heads - tails](#)

[Figure 17-14 Coefficient of variation](#)

[Figure 17-15 Final version of flip plot](#)

[Figure 17-16 Coefficient of variation of heads/tails and  
abs\(heads – tails\)](#)

[Figure 17-17 A large number of trials](#)

[Figure 17-18 Income distribution in Australia](#)

[Figure 17-19 Code and the histogram it generates](#)

- [Figure 17-20 Plot histograms of coin flips](#)  
[Figure 17-21 Histograms of coin flips](#)  
[Figure 17-22 PDF for random.random](#)  
[Figure 17-23 PDF for Gaussian distribution](#)  
[Figure 17-24 A normal distribution](#)  
[Figure 17-25 Plot of absolute value of x](#)  
[Figure 17-26 Checking the empirical rule](#)  
[Figure 17-27 Produce plot with error bars](#)  
[Figure 17-28 Estimates with error bars](#)  
[Figure 17-29 Exponential clearance of molecules](#)  
[Figure 17-30 Exponential decay](#)  
[Figure 17-31 Plotting exponential decay with a logarithmic axis](#)  
[Figure 17-33 A geometric distribution](#)  
[Figure 17-32 Producing a geometric distribution](#)  
[Figure 17-34 Simulating a hash table](#)  
[Figure 17-35 World Series simulation](#)  
[Figure 17-36 Probability of winning a 7-game series](#)

## Chapter 18

- [Figure 18-1 Checking Pascal's analysis](#)  
[Figure 18-2 Craps game class](#)  
[Figure 18-3 Simulating a craps game](#)  
[Figure 18-4 Using table lookup to improve performance](#)  
[Figure 18-5 Unit circle inscribed in a square](#)  
[Figure 18-6 Estimating  \$\pi\$](#)

## Chapter 19

- [Figure 19-1 The first few lines in bm\\_results2012.csv](#)  
[Figure 19-2 Read data and produce plot of Boston Marathon](#)

- [Figure 19-3 Boston Marathon finishing times](#)  
[Figure 19-4 Sampling finishing times](#)  
[Figure 19-5 Analyzing a small sample](#)  
[Figure 19-6 Effect of variance on estimate of mean](#)  
[Figure 19-7 Compute and plot sample means](#)  
[Figure 19-8 Sample means](#)  
[Figure 19-9 Estimating the mean of a continuous die](#)  
[Figure 19-10 An illustration of the CLT](#)  
[Figure 19-11 Produce plot with error bars](#)  
[Figure 19-12 Estimates of finishing times with error bars](#)  
[Figure 19-13 Standard error of the mean](#)  
[Figure 19-14 Sample standard deviation vs. population standard deviation](#)  
[Figure 19-15 Sample standard deviations](#)  
[Figure 19-16 Estimating the population mean 10,000 times](#)

## Chapter 20

- [Figure 20-1 A classic experiment](#)  
[Figure 20-2 Extracting the data from a file](#)  
[Figure 20-3 Plotting the data](#)  
[Figure 20-4 Displacement of spring](#)  
[Figure 20-5 Fitting a curve to data](#)  
[Figure 20-6 Measured points and linear model](#)  
[Figure 20-7 Linear and cubic fits](#)  
[Figure 20-8 Using the model to make a prediction](#)  
[Figure 20-9 A model up to the elastic limit](#)  
[Figure 20-10 Data from projectile experiment](#)  
[Figure 20-11 Plotting the trajectory of a projectile](#)

- [Figure 20-12 Plot of trajectory](#)  
[Figure 20-13 Computing R<sub>2</sub>](#)  
[Figure 20-14 Computing the horizontal speed of a projectile](#)  
[Figure 20-15 Fitting a polynomial curve to an exponential distribution](#)  
[Figure 20-16 Fitting an exponential](#)  
[Figure 20-17 An exponential on a semilog plot](#)  
[Figure 20-18 Using polyfit to fit an exponential](#)  
[Figure 20-19 A fit for an exponential function](#)

## Chapter 21

- [Figure 21-1 Finishing times for cyclists](#)  
[Figure 21-2 January 2020 temperature difference from the 1981-2010 average](#)  
[Figure 21-3 Plotting a t-distribution](#)  
[Figure 21-4 Visualizing the t-statistic](#)  
[Figure 21-5 Compute and print t-statistic and p-value](#)  
[Figure 21-6 Code for generating racing examples](#)  
[Figure 21-7 Probability of p-values](#)  
[Figure 21-8 Lyndsay's simulation of games](#)  
[Figure 21-9 Correct simulation of games](#)  
[Figure 21-10 Impact of sample size on p-value](#)  
[Figure 21-11 Comparing mean finishing times for selected countries](#)  
[Figure 21-12 Checking multiple hypotheses](#)  
[Figure 21-13 Has the sun exploded?](#)

## Chapter 22

- [Figure 22-1 Housing prices in the U.S. Midwest](#)  
[Figure 22-2 Plotting housing prices](#)

- [Figure 22-3 A different view of housing prices](#)  
[Figure 22-4 Housing prices relative to \\$200,000](#)  
[Figure 22-5 Comparing number of Instagram followers](#)  
[Figure 22-6 Do Mexican lemons save lives?](#)  
[Figure 22-7 Statistics for Anscombe's quartet](#)  
[Figure 22-8 Data for Anscombe's quartet](#)  
[Figure 22-9 Welfare vs. full-time jobs](#)  
[Figure 22-10 Sea ice in the Arctic](#)  
[Figure 22-11 Growth of Internet usage in U.S.](#)  
[Figure 22-12 Professor puzzles over students' chalk-throwing accuracy](#)  
[Figure 22-13 Probability of 48 anorexics being born in June](#)  
[Figure 22-14 Probability of 48 anorexics being born in some month](#)

## Chapter 23

- [Figure 23-1 A sample Pandas DataFrame bound to the variable wwc](#)  
[Figure 23-2 An example CSV file](#)  
[Figure 23-3 Building a dictionary mapping years to temperature data](#)  
[Figure 23-4 Building a DataFrame organized around years](#)  
[Figure 23-5 Produce plots relating year to temperature measurements](#)  
[Figure 23-6 Mean and minimum annual temperatures](#)  
[Figure 23-7 Rolling average minimum temperatures](#)  
[Figure 23-8 Average temperatures for select cities](#)  
[Figure 23-9 Variation in temperature extremes](#)  
[Figure 23-10 Global consumption of fossil fuels](#)

## Chapter 24

[Figure 24-1 Two sets of names](#)

[Figure 24-2 Associating a feature vector with each name](#)

[Figure 24-3 Feature vector/label pairs for presidents](#)

[Figure 24-4 Name, features, and labels for assorted animals](#)

[Figure 24-5 Visualizing distance metrics](#)

[Figure 24-6 Minkowski distance](#)

[Figure 24-7 Class Animal](#)

[Figure 24-8 Build table of distances between pairs of animals](#)

[Figure 24-9 Distances between three animals](#)

[Figure 24-10 Distances between four animals](#)

[Figure 24-11 Distances using a different feature representation](#)

## Chapter 25

[Figure 25-1 Height, weight, and shirt color](#)

[Figure 25-2 Class Example](#)

[Figure 25-3 Class Cluster](#)

[Figure 25-4 K-means clustering](#)

[Figure 25-5 Finding the best k-means clustering](#)

[Figure 25-6 A test of k-means](#)

[Figure 25-7 Examples from two distributions](#)

[Figure 25-8 Lines printed by a call to contrived\\_test\(1, 2, True\)](#)

[Figure 25-9 Generating points from three distributions](#)

[Figure 25-10 Points from three overlapping Gaussians](#)

[Figure 25-11 Mammal dentition in dentalFormulas.csv](#)

[Figure 25-12 Read and process CSV file](#)

[Figure 25-13 Scaling attributes](#)

[Figure 25-14 Start of CSV file classifying mammals by diet](#)

[Figure 25-15 Relating clustering to labels](#)

Chapter 26

[Figure 26-1 Plots of voter preferences](#)

[Figure 26-2 Confusion matrices](#)

[Figure 26-3 A more complex model](#)

[Figure 26-4 Functions for evaluating classifiers](#)

[Figure 26-5 First few lines of bm\\_results2012.csv](#)

[Figure 26-6 Build examples and divide data into training and test sets](#)

[Figure 26-7 Finding the k-nearest neighbors](#)

[Figure 26-8 Prevalence-based classifier](#)

[Figure 26-9 Searching for a good k](#)

[Figure 26-10 Choosing a value for k](#)

[Figure 26-11 Linear regression models for men and women](#)

[Figure 26-12 Produce and plot linear regression models](#)

[Figure 26-13 Using linear regression to build a classifier](#)

[Figure 26-14 Using sklearn to do multi-class logistic regression](#)

[Figure 26-15 Example of two-class logistic regression](#)

[Figure 26-16 Use logistic regression to predict gender](#)

[Figure 26-17 Construct ROC curve and find AUROC](#)

[Figure 26-18 ROC curve and AUROC](#)

[Figure 26-19 Class Passenger](#)

[Figure 26-20 Read Titanic data and build list of examples207](#)

[Figure 26-21 Test models for Titanic survival](#)

[Figure 26-22 Print statistics about classifiers](#)

# 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 either a two-semester or a three-quarter introductory computer science sequence.

The book is aimed at 1) readers with little or no prior programming experience who have a desire to understand computational approaches to problem solving, and 2) more experienced programmers who want to learn how to use computation to model things or explore data.

We emphasize breadth rather than depth. The goal is to provide readers 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. Readers 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 readers become skillful at making productive use of computational techniques. They should learn to use computational modes of thoughts to frame problems, to build computational models, 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.

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

more challenging, and are suitable for exam questions. And others are challenging enough to be useful as homework assignments.

Chapters 1-13 contain the kind of material typically included in an introductory computer science course, but the presentation is not conventional. We braid together four strands of material:

- The basics of programming,
- The Python 3 programming language,
- Computational problem solving techniques, and
- Computational complexity.

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 readers learn Python and how to be a good programmer in the context of using computation to solve interesting problems. These chapters have been revised to proceed more gently and include more exercises than the corresponding chapters in the second edition of this book.

Chapter 13 contains an introduction to plotting in Python. This topic is often not covered in introductory courses, but we believe that learning to produce visualizations of information is an important skill that should be included in an introductory computer science course. This chapter includes material not covered in the second edition.

Chapters 14-26 are about using computation to help understand the real world. They cover material that we think should become the usual second course in a computer science curriculum. 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 and analysis, stochastic

programs, 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. Except for Chapter 23, the material in this section of the book focuses on conceptual issues rather than programming. Chapter 23 is an introduction to Pandas, a topic not covered in earlier editions.

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,”<sup>1</sup> 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 complete the finger excercises that involve coding. If you are up for trying more ambitious tasks, try some of the problem sets available through

<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0001-introduction-to-computer-science->

[and-programming-in-python-fall-2016/](#)

and

<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0002-introduction-to-computational-thinking-and-data-science-fall-2016/>.

---

- 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 of the first edition.

Like all successful professors, I owe a great deal to my graduate students. In addition to doing great research (and letting me take some of the credit for it), Guha Balakrishnan, Davis Blalock, Joel Brooks, Ganeshapillai Gartheeban, Jen Gong, Katie Lewis, Yun Liu, Jose Javier Gonzalez Ortiz, Anima Singh, Divya Shanmugam, 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., who edited the first two editions of this book, and Lisa Ruffolo who edited the current edition. Both Julie and Lisa were collaborators who read the book with the eyes of a student, and told 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. They buried me in “suggestions” that were too good to ignore.

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 backpack performs a 100 billion or so calculations a second. It's hard to imagine 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 more than 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 100,000 metric tons. For comparison, that's roughly the combined weight of 16,000 African elephants.<sup>2</sup>

For most of human history, computation was limited by how fast the human brain could calculate and how well the human hand could record computational results. This meant that only the smallest problems could be attacked computationally. Even with the speed of modern computers, some problems are still beyond modern computational models (e.g., fully 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  $y \cdot y =$

$x$ ,” and “it is possible to travel by train from Paris to Rome.” These are statements of fact. Unfortunately, they don’t tell us anything about how to find a square root or how to take trains from Paris to Rome.

**Imperative knowledge** is “how to” knowledge, or recipes for deducing information. Heron of Alexandria was the first<sup>3</sup> to document a way to compute the square root of a number. His method for finding the square root of a number, call it  $x$ , can be summarized as:

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 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  $(3 + 25/3)/2 = 5.67$ .<sup>4</sup>
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 to execute each step. Such a description is called an **algorithm**.<sup>5</sup> The algorithm we used to approximate square root 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 good enough.

More formally, an algorithm is a finite list of instructions describing a set of **computations** that when executed on a set of inputs will proceed through a sequence of well-defined states and eventually produce an output.

An algorithm is 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.

The recipe 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.

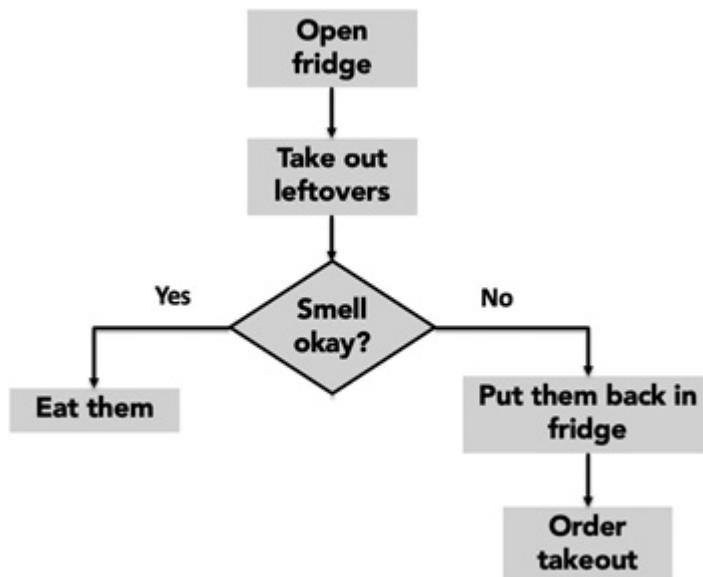
So how does one capture the idea of a recipe in a mechanical process? One way is to design a machine specifically intended to compute square roots. Odd as this may sound, the earliest computing machines were, in fact, **fixed-program computers**, meaning they were designed to solve a specific mathematical 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 to break German Enigma codes. Some simple computers still use this approach. For example, a four-function calculator<sup>6</sup> 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.<sup>7</sup> It was distinguished from its predecessors by being a **stored-program computer**. Such a computer stores (and manipulates) a

sequence of instructions, and has components that execute any instruction in that sequence. The heart of such a computer is an **interpreter** that can execute any legal set of instructions, and thus can be used to compute anything that can be described using those instructions. The result of the computation can even be a new sequence of instructions, which can then be executed by the computer that generated them. In other words, it is possible for a computer to program itself.<sup>8</sup>

Both the program and the data it manipulates reside in memory. Typically, a **program counter** 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 another 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.

People sometimes use **flowcharts** to depict flow of control. By convention, we use rectangular boxes to depict a processing step, a diamond to depict a test, and arrows to indicate the order in which things are done. [Figure 1-1](#) contains a flowchart depicting an approach to getting dinner.



[Figure 1-1](#) Flowchart of getting dinner

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.

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 unlimited memory in the form of a “tape” on which one could write zeroes and ones, and a handful of 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 takes an arbitrary program as input, and prints `true` if and only if the input program 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 you have, and whatever methods you have for assembling them, the best thing and the worst thing about programming are the same: the computer will do exactly what you

tell it to do—nothing more, nothing less. This is a good thing because it means that you can make the computer 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. 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 websites. And Python is an excellent 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 English syntax 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. Consider, for example, the strings “He run quickly” and “I runs quickly.” Each has the form `<pronoun> <regular verb> <adverb>`, which is a syntactically acceptable sequence. Nevertheless, neither is valid English, because of the rather peculiar rule that for a regular verb when the subject of a sentence is first or second person, the verb does not end with an “s,” but when the subject is third person singular, it does. These are examples of static semantic errors.

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 detects all syntactic errors, and does 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 the programmer is able to fix it without too much thought.

Identifying and resolving static semantic errors is 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.

If a program has no syntactic errors and no static semantic errors, it has a meaning, i.e., it has semantics. Of course, it might not have the semantics that its creator intended. 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 an obvious indication that it has done so. In a properly designed computing system, when a program crashes, it does not damage the overall system. Alas, 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 reboot the whole system.
- It might keep running, and running, and running, and never stop. If you have no idea of approximately how long the program is supposed to take to do its job, this situation can be hard to recognize.
- It might run to completion and produce an answer that might, or might not, be correct.

Each of these outcomes is bad, but the last one 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 so 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. (For an amusing illustration of this, take a look at the video <https://www.youtube.com/watch?v=FN2RM-CHkuI&t=24s>.)

---

## 1.1 Terms Introduced in Chapter

declarative knowledge  
imperative knowledge  
algorithm  
computation  
fixed-program computer  
stored-program computer  
interpreter  
program counter  
flow of control  
flowchart  
programming language  
Universal Turing machine  
Church-Turing thesis

halting problem  
Turing completeness  
literals  
infix operators  
syntax  
static semantics  
semantics

---

- 2** Not everything is more expensive than it used to be. In 1960, a bit of computer memory cost about \$0.64. Today it costs about \$0.00000004.
- 3** 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.
- 4** For simplicity, we are rounding results.
- 5** The word “algorithm” is derived from the name of the Persian mathematician Muhammad ibn Musa al-Khwarizmi.
- 6** It might be hard for some of you to believe, but once upon a time people did not carry around phones that doubled as computational facilities. People actually carried small devices that could be used only for arithmetic calculation.
- 7** 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.
- 8** This possibility has been the inspiration for a plethora of dystopian books and movies.

# 2

## INTRODUCTION TO PYTHON

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

- **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 for 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 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 relate to 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 you learn Python, and that's a good thing. What is much more important, however, is that you will learn something about how to write programs that solve problems. You can transfer this skill to any programming language.

Python is a general-purpose programming language you can use 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).

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. A large and growing number of freely available libraries interface to Python and provide useful extended functionality. We use several of these libraries in this book.

We are ready to introduce some of the basic elements of Python. These are common to almost all programming languages in concept, though not in detail.

This book is not just an introduction to Python. It uses 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 every detail because excellent online resources describe every aspect of the language. We suggest that you use these free online resources as needed.

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 important improvements to the language itself, it marked a shift in the evolutionary path of the language. Many groups

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 Python 2. However, Python 3 is not backward compatible. This means that most programs and libraries written for earlier versions of Python cannot be run using implementations of Python 3.

By now, all of the important public domain Python libraries have been ported to Python 3. Today, there is no reason to use Python 2.

---

## 2.1 Installing Python and Python IDEs

Once upon time, programmers used general-purpose text editors to enter their programs. Today, most programmers prefer to use a text editor that is part of an **integrated development environment (IDE)**.

The first Python IDE, IDLE,<sup>9</sup> came as part of the standard Python installation package. As Python has grown in popularity, other IDEs have sprung up. These newer IDEs 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 IDEs. The code appearing in this book was created and tested using Anaconda.

IDEs 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. All of the Python IDEs 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.

This would be a good time to install Anaconda (or some other IDE) on your computer, so that you can run the examples in the

book, and, more importantly, attempt the programming finger exercises. To install Anaconda, go to

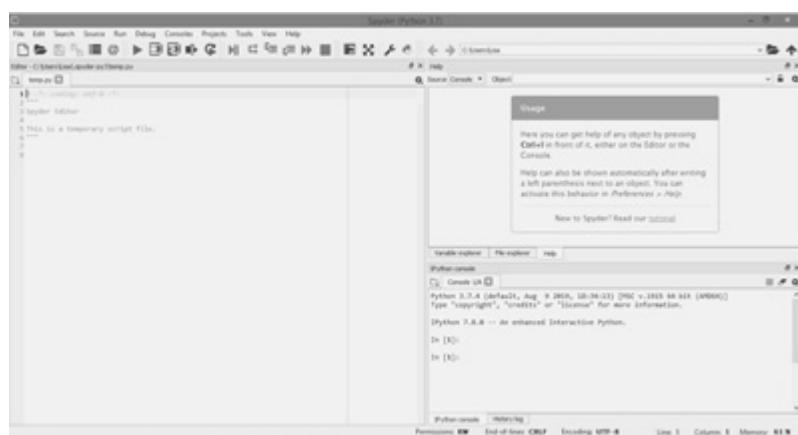
<https://www.anaconda.com/distribution/>

and follow the instructions.

Once the installation is complete, start the application `Anaconda-Navigator`. A window containing a collection of Python tools will appear. The window will look something like [Figure 2-1](#).<sup>10</sup> For now, the only tool we will use is `Spyder`. When you launch Spyder (by clicking on its `Launch` button, of all things), a window similar to [Figure 2-2](#) will open.



[Figure 2-1](#) Anaconda startup window



[Figure 2-2](#) Spyder window

The pane on the lower right of [Figure 2-2](#) is an **IPython console** running an interactive Python **shell**. You can type and execute Python commands in this window. The pane on the upper right is a help window. It is often convenient to close that window (by clicking on the  $\times$ ), so that more real estate is available for the IPython console. The pane on the left is an edit window into which you can type programs that can be saved and run. The toolbar at the top of the window makes it easy to perform various tasks such as opening files and printing programs.<sup>11</sup> Documentation for `Spyder` can be found at <https://www.spyder-ide.org/>.

---

## 2.2 The Basic Elements of Python

A Python **program**, sometimes called a **script**, is a sequence of definitions and commands. The Python interpreter in the shell evaluates the definitions and executes the commands.

We recommend that you start a Python shell (e.g., by starting `Spyder`) now, and use it to try the examples contained in the rest of this chapter. And, for that matter, in the rest of the book.

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<sup>12</sup> `print`, which outputs 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.

### 2.2.1 Objects, Expressions, and Numerical Types

**Objects** are the core things that Python programs manipulate. Each object has a **type** that defines what 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.<sup>13</sup> **Non-scalar** objects, for example strings, have internal structure.

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'` is 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  $1.6 \times 10^3$ , 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 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 3.3.
- `bool` is used to represent the Boolean values `True` and `False`.
- `None` is a type with a single value. We will say more about `None` in Section 4.

Objects and **operators** can be combined to form **expressions**, each of which evaluates to an object of some type. This is called 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 2.2.2. Be forewarned—you will make the mistake of typing “=” when you meant to type “==”. Keep an eye out for this error.

In a Spyder console, something that looks like `In [1]:` is a **shell prompt** indicating that the interpreter is expecting the user to type some Python code into the shell. The line below 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
Out[1]: 3

3+2
Out[2]: 5

3.0+2.0
Out[3]: 5.0

3!=2
Out[4]: True
```

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

```
type(3)
Out[5]: int

type(3.0)
Out[6]: float
```

Operators on objects of type `int` and `float` are listed in [Figure 2-3](#). 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`.

**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 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 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 is a `float`, the result is a `float`.

**i//j** is floor 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 floor division returns the quotient and ignores the remainder. If *j* == 0, an error occurs.

**i/j** is *i* divided by *j*. The `/` operator performs floating-point division. For example, the value of `6/4` is 1.5. If *j* == 0, an error occurs.

**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 is a `float`, the result is a `float`.

[Figure 2-3](#) Operators on types `int` and `float`

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

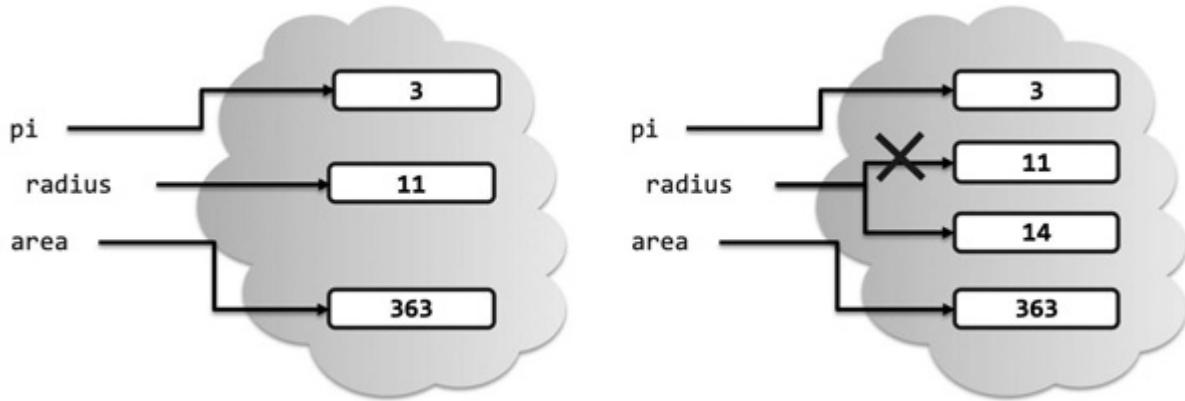
- `a and b` is `True` if both `a` and `b` are `True`, and `False` otherwise.
- `a or b` is `True` if at least one of `a` or `b` is `True`, and `False` otherwise.
- `not a` is `True` if `a` is `False`, and `False` if `a` is `True`.

## 2.2.2 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
```

The code first **binds** the names `pi` and `radius` to different objects of type `int`.<sup>14</sup> It then binds the name `area` to a third object of type `int`. This is depicted on the left side of [Figure 2-4](#).



[Figure 2-4](#). 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 on the right side of [Figure 2-4](#). 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 `=` symbol. Remember this too: an object can have one, more than one, or no name associated with it.

Perhaps we shouldn't have said, “*a variable is just a name*.” Despite what Juliet said,<sup>15</sup> names matter. Programming languages let us describe computations so that computers can 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 so 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, the code fragments 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 suspect 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 (though they cannot start with a digit), and the special character `_` (underscore). Python variable names are case-sensitive e.g., `Romeo` and `romeo` are different names. Finally, a few **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 3.8 are

and	break	elif	for	in	not	True
as	class	else	from	is	or	try
assert	continue	except	global	lambda	pass	while
async	def	False	if	nonlocal	raise	with
await	del	finally	import	None	return	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, we might write

```
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
area_circle = pi*radius**2
area_square = side*side
difference = area_square - area_circle
```

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
```

---

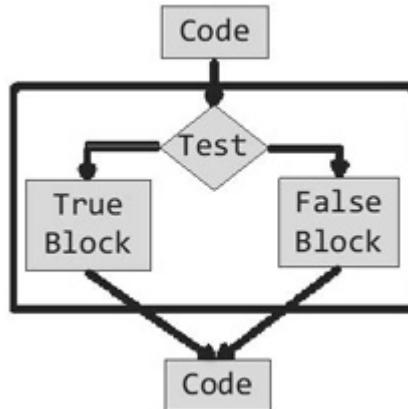
## 2.3 Branching Programs

The kinds of computations we have been looking at so 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 box in [Figure 2-5, 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`
- 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 2-5](#) Flowchart for conditional statement

In Python, a conditional statement has the form

<pre>if Boolean expression:     block of code else:     block of code</pre>	<p style="text-align: center;">or</p> <pre>if Boolean expression:     block of code</pre>
---	---

In describing the form of Python statements, we use italics to identify 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 bracketing symbols to delineate blocks of code, e.g., C encloses blocks in set braces, `{ }`. An advantage of the Python approach is that it ensures that the visual structure of a program is an accurate representation of its semantic structure. Because indentation is semantically important, the notion of a line is also important. A line of code that is too long to read easily can be broken into multiple lines on the screen by ending each line on the screen, other than the last one, with a backslash (`\`). For example,

```
x = 111111111111111111111111111111111111111111111111111111111111111 +  
22222222222333222222222 +\  
33333333333333333333333333333333333333333333333333333333333333333
```

Long lines can also be wrapped using Python's implied line continuation. This is done with bracketing, i.e., parentheses, square brackets, and braces. For example,

```
x = 111111111111111111111111111111111111111111111111111111111111111 +  
22222222222333222222222 +  
333333333333333333333333333333333333333333333333333333333333333
```

is interpreted as two lines (and therefore produces an “unexpected indent” syntax error whereas

```
x = (111111111111111111111111111111111111111111111111111111111111111 +  
22222222222333222222222 +  
33333333333333333333333333333333333333333333333333333333333333)
```

is interpreted as a single line because of the parentheses. Many Python programmers prefer using implied line continuations to using a backslash. Most commonly, programmers break long lines at commas or operators.

Returning to conditionals, when either the true block or the false block of a conditional contains another conditional, the conditional statements are said to be **nested**. The following code contains 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')

```

**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 the smallest value of the three.

You can attack this exercise in a number of ways. There are eight separate cases to consider: they are all odd (one case), exactly two of them are odd (three cases), exactly one of them is odd (three cases), or none of them is odd (one case). So, a simple solution would involve a sequence of eight `if` statements, each with a single `print` statement:

```

if x%2 != 0 and y%2 != 0 and z%2 != 0:
    print(max(x, y, z))
if x%2 != 0 and y%2 != 0 and z%2 == 0:
    print(max(x, y))
if x%2 != 0 and y%2 == 0 and z%2 != 0:
    print(max(x, z))
if x%2 == 0 and y%2 != 0 and z%2 != 0:
    print(max(y, z))
if x%2 != 0 and y%2 == 0 and z%2 == 0:
    print(x)
if x%2 == 0 and y%2 != 0 and z%2 == 0:
    print(y)
if x%2 == 0 and y%2 == 0 and z%2 != 0:
    print(z)
if x%2 == 0 and y%2 == 0 and z%2 == 0:
    print(min(x, y, z))

```

That gets the job done, but is rather cumbersome. Not only is it 16 lines of code, but the variables are repeatedly tested for oddness. The following code is both more elegant and more efficient:

```
answer = min(x, y, z)
if x%2 != 0:
    answer = x
if y%2 != 0 and y > answer:
    answer = y
if z%2 != 0 and z > answer:
    answer = z
print(answer)
```

The code is based on a common programming paradigm. It starts by assigning a provisional value to a variable (`answer`), updating it when appropriate, and then printing the final value of the variable. Notice that it tests whether each variable is odd exactly once, and contains only a single print statement. This code is pretty much as well as we can do, since any correct program must check each variable for oddness and compare the values of the odd variables to find the largest of them.

Python supports **conditional expressions** as well as conditional statements. Conditional expressions are of the form

```
expr1 if condition else expr2
```

If the condition evaluates to `True`, the value of the entire expression is `expr1`; otherwise it is `expr2`. For example, the statement

```
x = y if y > z else z
```

sets `x` to the maximum of `y` and `z`. A conditional expression can appear any place an ordinary expression can appear, including within conditional expressions. So, for example,

```
print((x if x > z else z) if x > y else (y if y > z else z))
```

prints the maximum of `x`, `y`, and `z`.

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 the program 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 limited in what they can do. Consider 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, 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 allow us write programs of arbitrary complexity. We get to that in Section 2.5.

---

## 2.4 Strings and Input

Objects of type `str` are used to represent characters.<sup>16</sup> 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 `123`.

Try typing the following expressions in to the Python interpreter.

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

The operator `+` is said to be **overloaded** because it has different meanings depending upon the types of the objects to which it is applied. For example, the operator `+` 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 is a **repetition operator**—the expression `n*s`, where `n` is an `int` and `s` is a `str`, evaluates to a `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

```
new_id  
'a'*'a'
```

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

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

Because `new_id` 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 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]` evaluates to `'bc'`. Why does it end at index `end-1` rather than `end`? So that expressions such as `'abc'[0:len('abc')]` have the value you 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')]`. It is also possible to supply a third argument to select a non-contiguous slice of a string. For example, the value of the expression `'123456789'[0:8:2]` is the string `'1357'`.

It is often convenient to convert objects of other types to strings using the `str` function. Consider, for example, the code

```
num = 30000000
fraction = 1/2
print(num*fraction, 'is', fraction*100, '%', 'of', num)
print(num*fraction, 'is', str(fraction*100) + '%', 'of',
      num)
```

which prints

```
15000000.0 is 50.0 % of 30000000
15000000.0 is 50.0% of 30000000
```

The first print statement inserts a space between 50 and % because Python automatically inserts a space between the arguments to print. The second print statement produces a more appropriate output by combining the 50 and the % into a single argument of type str.

**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.

Returning to the output of our print statements, you might be wondering about that .0 at the end of the first number printed. That appears because 1/2 is a floating-point number, and the product of an int and a float is a float. It can be avoided by converting num\*fraction to an int. The code

```
print(int(num*fraction), 'is', str(fraction*100) + '%',
      'of', num)
```

prints 15000000 is 50.0% of 30000000.

Python 3.6 introduced an alternative, more compact, way to build string expressions. An **f-string** consists of the character f (or F) following by a special kind of string literal called a **formatted string literal**. Formatted string literals contain both sequences of characters (like other string literals) and expressions bracketed by curly braces. These expressions are evaluated at runtime and automatically converted to strings. The code

```
print(f'{int(num*fraction)} is {fraction*100}% of {num}')
```

produces the same output as the previous, more verbose, print statement. If you want to include a curly brace in the string denoted by an f-string, use two braces. E.g., print(f'{{{{3\*5}}}}') prints {15}.

The expression inside an f-string can contain modifiers that control the appearance of the output string.<sup>17</sup> These modifiers are separated from the expression denoting the value to be modified by a colon. For example, the f-string f'{3.14159:.2f}' evaluates to the string '3.14' because the modifier .2f instructs Python to truncate the string representation of a floating-point number to two digits after the decimal point. And the statement

```
print(f'{num*fraction:,.0f} is {fraction*100}% of {num:,}')  
prints 15,000,000 is 50.0% of 30,000,000 because the , modifier  
instructs Python to use commas as thousands separators. We will  
introduce other modifiers as convenient later in the book.
```

### 2.4.1 Input

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

Executing the code `name = input('Enter your name: ')` will display the line

```
Enter your name:
```

in the console window. If you then type `George Washington` and press `enter`, the string '`George Washington`' will be assigned to the variable `name`. If you then execute `print('Are you really', name, '?')`, the line

```
Are you really George Washington ?
```

would be displayed. Notice that the `print` statement introduces a space before the "?". It does this because when `print` is given multiple arguments, it places a space between the values associated with the arguments. The space could be avoided by executing `print('Are you really ' + name + '?')` or `print(f'Are you really {name}?')`, each of which produces a single string and passes that string as the only argument to `print`.

Now consider the code

```
n = input('Enter an int: ')  
print(type(n))
```

This code will always print

```
<class 'str'>
```

because `input` always returns an object of type `str`, even if the user has entered something that looks like an integer. For example, if the user had entered `3`, `n` would be bound to the `str` '`3`' not the `int` `3`. So, the value of the expression `n*4` would be '`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.

**Finger exercise:** Write code that asks the user to enter their birthday in the form `mm/dd/yyyy`, and then prints a string of the form '`You were born in the year yyyy.`'

### 2.4.2 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.

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` 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 webpages.<sup>18</sup> 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 of the web uses UTF-8, I was able to cut the string from a webpage and paste it directly into my program. There are ways to directly enter Unicode characters from a keyboard, but unless you have a special keyboard, they are all rather cumbersome.

---

## 2.5 While Loops

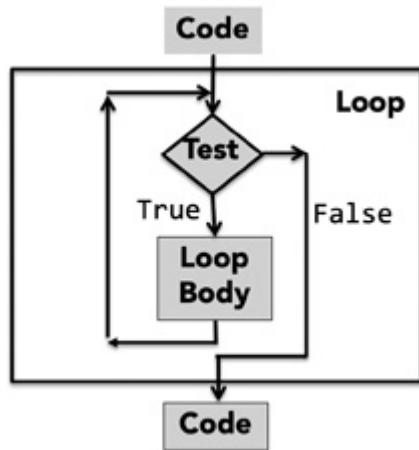
Near the end of Section 2.3, we mentioned that most computational tasks cannot be accomplished using branching programs. Consider, for example, writing a program that asks for the number of X's. You might think about writing something like

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

But it would quickly become apparent that you would need as many conditionals as there are positive integers—and there are an infinite number of those. What you want to write is a program that looks like (the following is **pseudocode**, not Python)

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

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 box in [Figure 2-6](#). 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.



[Figure 2-6](#) Flowchart for iteration

We can write the kind of loop depicted in [Figure 2-6](#) using a **while** statement. Consider the code in [Figure 2-7](#).

```
x = 3
ans = 0
num_iterations = 0
while (num_iterations < x):
    ans = ans + x
    num_iterations = num_iterations + 1
print(f'{x}*{x} = {ans}')
```

[Figure 2-7](#) Squaring an integer, the hard way

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 2-8](#) shows the value associated with each variable each time the test at the start of the loop is reached. We constructed the table 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 quaint, but it is an excellent way to understand how a program behaves.<sup>[19](#)</sup>

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

[Figure 2-8](#) 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 `num_iterations` will also be `0`, and the loop body will never be executed.

Suppose `x > 0`. The initial value of `num_iterations` will be less than `x`, and the loop body will be executed at least once. Each time the loop body is executed, the value of `num_iterations` is increased by exactly `1`. This means that since `num_iterations` started out less than `x`, after some finite number of iterations of the loop, `num_iterations` will equal `x`. 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 `num_iterations` farther from `x` 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? Changing the test to `num_iterations < abs(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.

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

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.

**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.

---

## 2.6 For Loops and Range

The `while` loops we have used so far are highly stylized, often iterating 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. For example, the code

```
total = 0  
for num in (77, 11, 3):  
    total = total + num  
print(total)
```

will print 91. The expression `(77, 11, 3)` is a **tuple**. We discuss tuples in detail in Section 5. For now, just think of a tuple as a sequence of values.

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 such that `(start + i*step)` is strictly less than `stop`. If `step` is negative, the last element is the smallest integer such that `(start + i*step)` is greater than `stop`. For example, the expression `range(5, 40, 10)` yields the sequence 5, 15, 25, 35, and the expression `range(40, 5, -10)` yields the sequence 40, 30, 20, 10.

If the first argument to `range` 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. We will discuss `range` in more depth in Section 5.2.

Consider the code

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

It prints

```
0
1
2
3
```

The code in [Figure 2-9](#) reimplements the algorithm in [Figure 2-7](#) for squaring an integer (corrected so that it works for negative numbers). Notice that unlike the `while` loop implementation, the number of iterations is not controlled by an explicit test, and the index variable `num_iterations` is not explicitly incremented.

```
x = 3
xans = 0
for num_iterations in range(abs(x)):
    ans = ans + abs(x)
print(f'{x}*{x} = {ans}')
```

[Figure 2-9](#) Using a `for` statement

Notice that the code in [Figure 2-9](#) does not change the value of `num_iterations` within the body of the `for` loop. This is typical, but not necessary, which raises the question of what happens if the index variable is modified within the `for` loop. Consider

```
for i in range(2):
    print(i)
    i = 0
    print(i)
```

Do you think that it will print 0, 0, 1, 0, and then halt? Or do you think it will print 0 over and over again?

The answer is 0, 0, 1, 0. Before the first iteration of the `for` loop, the `range` function is evaluated and the first value in the sequence it yields is assigned to the index variable, `i`. At the start of

each subsequent iteration of the loop, `i` is assigned the next value in the sequence. When the sequence is exhausted, the loop terminates. The above `for` loop is equivalent to the code

```
index = 0
last_index = 1
while index <= last_index:
    i = index
    print(i)
    i = 0
    print(i)
    index = index + 1
```

Notice, by the way, that code with the `while` loop is considerably more cumbersome than the `for` loop. The `for` loop is a convenient linguistic mechanism.

Now, what do you think

```
x = 1
for i in range(x):
    print(i)
x = 4
```

prints? Just 0, because 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.

Now, let's see how often things are evaluated when we nest loops. Consider

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

How many times is each of the two loops executed? We already saw that the `range(x)` controlling the outer loop is evaluated the first time it is reached and not reevaluated for each iteration, so there are four iterations of the outer loop. That implies that the inner `for` loop is reached four times. The first time it is reached, the variable `x = 4`, so there will be four iterations. However, the next three times it is reached, `x = 2`, so there will be two iterations each time. Consequently, if you run

```
x = 3
for j in range(x):
    print('Iteration of outer loop')
    for i in range(x):
        print('    Iteration of inner loop')
    x = 2
```

it prints

```
Iteration of outer loop
    Iteration of inner loop
    Iteration of inner loop
    Iteration of inner loop
Iteration of outer loop
    Iteration of inner loop
    Iteration of inner loop
Iteration of outer loop
    Iteration of inner loop
    Iteration of inner loop
```

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:** Write a program that prints the sum of the prime numbers greater than 2 and less than 1000. Hint: you probably want to use a `for` loop that is a primality test nested inside a `for` loop that iterates over the odd integers between 3 and 999.

---

## 2.7 Style Matters

Much of this book is devoted to helping you learn a programming language. But knowing a language and knowing how to use a language well are two different things. Consider the following two sentences:

*“Everybody knows that if a man is unmarried and has a lot of money, he needs to get married.”*

*“It is a truth universally acknowledged, that a single man in possession of a good fortune, must be in want of a wife.”<sup>20</sup>*

Each is a proper English sentence, and each means approximately the same thing. But they are not equally compelling, and perhaps not equally easy to understand. Just as style matters when writing in English, style matters when writing in Python. However, while having a distinctive voice might be an asset for a novelist, it is not an asset for a programmer. The less time readers of a program have to spend thinking about things irrelevant to the meaning of the code, the better. That's why good programmers follow coding conventions designed to make programs easy to understand, rather than fun to read.

Most Python programmers follow the conventions laid out in the **PEP 8 style guide**.<sup>21</sup> Like all sets of conventions, some of its prescriptions are arbitrary. For example, it prescribes using four spaces for indents. Why four spaces and not three or five? No particularly good reason. But if everybody uses the same number of spaces, it is easier to read (and perhaps combine) code written by different people. More generally, if everyone uses the same set of conventions when writing Python, readers can concentrate on understanding the semantics of the code rather than wasting mental cycles assimilating stylistic decisions.

The most important conventions have to do with naming. We have already discussed the importance of using variable names that convey the meaning of the variable. Noun phrases work well for this. For example, we used the name `num_iterations` for a variable denoting the number of iterations. When a name includes multiple words, the convention in Python is to use an underscore (`_`) to separate the words. Again, this convention is arbitrary. Some programmers prefer to use what is often called camelCase, e.g., `numIterations`—arguing that it is faster to type and uses less space.

There are also some conventions for single-character variable names. The most important is to avoid using a lowercase L or uppercase I (which are easily confused with the number one) or an uppercase O (which is easily confused with the number zero).

That's enough about conventions for now. We will return to the topic as we introduce various aspects of Python.

We have now covered pretty much everything about Python that you 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 you have already learned to solve some simple problems.

---

## 2.8 Terms Introduced in Chapter

low-level language

high-level language

interpreted language

compiled language

source code

machine code

Python

integrated development environment (IDE)

Anaconda

Spyder

IPython console

shell

program (script)

command (statement)

object

type

scalar object

non-scalar object

literal

floating point  
bool  
None  
operator  
expression  
value  
shell prompt  
variable  
binding  
assignment  
reserved word  
comment (in code)  
straight-line program  
branching program  
conditional  
indentation (in Python)  
nested statement  
compound expression  
constant time  
computational complexity  
conditional expression  
strings  
overloaded operator  
repetition operator  
type checking  
indexing  
slicing

type conversion (casting)  
formatted string expression  
input  
Unicode  
iteration (looping)  
pseudocode  
while loop  
hand simulation  
break  
for loop  
tuple  
range  
in operator  
PEP 8 style guide

---

- 9** 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.
- 10** Since Anaconda is frequently updated, by the time you read this, the appearance of the `Spyder` window may well have changed.
- 11** If you don't like the appearance of the `Spyder` window, click on the wrench in the toolbar to open the Preferences window, and change settings as you see fit.
- 12** Functions are discussed in Section 4.
- 13** Yes, atoms are not truly indivisible. However, splitting them is not easy, and doing so can have consequences that are not always desirable.

- 14 If you believe that the actual value of  $\pi$  is not 3, you're right. We even demonstrate that fact in Section 18.4.
- 15 “What's in a name? That which we call a rose by any other name would smell as sweet.”
- 16 Unlike many programming languages, Python has no type corresponding to a character. Instead, it uses strings of length 1.
- 17 These modifiers are the same modifiers used in the `.format` method associated with strings.
- 18 In 2016, over 85% of the pages on the web were encoded using UTF-8.
- 19 It is also possible to hand-simulate a program using pen and paper, or even a text editor.
- 20 *Pride and Prejudice*, Jane Austen.
- 21 PEP is an acronym standing for “Python Enhancement Proposal.” PEP 8 was written in 2001 by Guido van Rossum, Barry Warsaw, and Nick Coghlan.

# 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 simple programs. Along the way, we'll sneak in 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. The operator `!=` means not equal.

```
#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

The code first attempts to set the variable `ans` to the cube root of the absolute value of `x`. If it succeeds, it then sets `ans` to `-ans` if `x` is negative. The heavy lifting (such as it is) in this code is done in the

`while` loop. Whenever a program contains a loop, it is important to understand what causes the program to eventually exit this loop. For what values of `x` will this `while` loop 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.

This argument is based upon the notion of a **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  $\leq 0$ , 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 `ctrl` 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. Remove or comment out the print statement that you inserted for debugging, and reinsert the statement `ans = ans + 1`. Now 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, run time is not usually a problem. Modern computers are amazingly fast. It takes less than one nanosecond—one billionth of a second—to execute an instruction. It's 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 100 feet, a modern computer can execute millions of instructions.

Just for fun, try executing the code

```
max_val = int(input('Enter a positive integer: '))
i = 0
while i < max_val:
    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.

Let's look at another example of exhaustive enumeration: testing whether an integer is a prime number and returning the smallest divisor if it is not. A prime number is an integer greater than 1 that is evenly divisible only by itself and 1. For example, 2, 3, 5, and 111,119 are primes, and 4, 6, 8 and 62,710,561 are not primes.

The simplest way to find out if an integer,  $x$ , greater than 3 is prime, is to divide  $x$  by each integer between 2 and,  $x-1$ . If the remainder of any of those divisions is 0,  $x$  is not prime, otherwise  $x$  is prime. The code in [Figure 3-2](#) implements that approach. It first asks the user to enter an integer, converts the returned string to an `int`, and assigns that integer to the variable `x`. It then sets up the initial conditions for an exhaustive enumeration by initializing `guess` to 2 and the variable `smallest_divisor` to `None`—indicating that until proven otherwise, the code assumes that  $x$  is prime.

The exhaustive enumeration is done within a `for` loop. The loop terminates when either all possible integer divisors of  $x$  have been tried or it has discovered an integer that is a divisor of  $x$ .

After exiting the loop, the code checks the value of `smallest_divisor` and prints the appropriate text. The trick of initializing a variable before entering a loop, and then checking whether that value has been changed upon exit, is a common one.

```
# Test if an int > 2 is prime. If not, print smallest divisor
x = int(input('Enter an integer greater than 2: '))
smallest_divisor = None
for guess in range(2, x):
    if x%guess == 0:
        smallest_divisor = guess
        break
if smallest_divisor != None:
    print('Smallest divisor of', x, 'is', smallest_divisor)
else:
    print(x, 'is a prime number')
```

[Figure 3-2](#) Using exhaustive enumeration to test primality

**Finger exercise:** Change the code in [Figure 3-2](#) so that it returns the largest rather than the smallest divisor. Hint: if  $y^*z = x$  and  $y$  is the smallest divisor of  $x$ ,  $z$  is the largest divisor of  $x$ .

The code in [Figure 3-2](#) works, but is unnecessarily inefficient. For example, there is no need to check even numbers beyond 2, since if an integer is divisible by any even number, it is divisible by 2. The code in [Figure 3-3](#) takes advantage of this fact by first testing whether  $x$  is an even number. If not, it uses a loop to test whether  $x$  is divisible by any odd number.

While the code in [Figure 3-3](#) is slightly more complex than the code in [Figure 3-2](#), it is considerably faster, since half as many numbers are checked within the loop. The opportunity to trade code complexity for runtime efficiency is a common phenomenon. But faster does not always mean better. There is a lot to be said for simple code that is obviously correct and still fast enough to be useful.

```
# Test if an int > 2 is prime. If not, print smallest divisor
x = int(input('Enter an integer greater than 2: '))
smallest_divisor = None
if x%2 == 0:
    smallest_divisor = 2
else:
    for guess in range(3, x, 2):
        if x%guess == 0:
            smallest_divisor = guess
            break
if smallest_divisor != None:
    print('Smallest divisor of', x, 'is', smallest_divisor)
else:
    print(x, 'is a prime number')
```

[Figure 3-3](#) A more efficient primality test

**Finger exercise:** Write a program that asks the user to enter an integer and prints two integers, `root` and `pwr`, such that  $1 < \text{pwr} < 6$  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.

**Finger exercise:** Write a program that prints the sum of the prime numbers greater than 2 and less than 1000. Hint: you probably want to have a loop that is a primality test nested inside a loop that iterates over the odd integers between 3 and 999.

---

## 3.2 Approximate Solutions and Bisection Search

Imagine that someone asks you to write a program that prints 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  $\sqrt{2}$ ? The square root of  $\sqrt{2}$  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 thing a program can do is find 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 3-4](#) implements an algorithm that prints an approximation to the square root of  $x$ .

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

[Figure 3-4](#). 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.

If  $x$  is 25, the code will print

```
number of guesses = 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. Alas, it will report

```
number of guesses = 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:
```

When we run our code after this change, it reports that

```
0.48989999999996237 is close to square root of 0.25
```

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 our starting point, 0, and on the size of the steps. Roughly speaking, the program will execute the `while` loop at most  $x/\text{step}$  times.

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

```
number of guesses = 3513631
Failed on square root of 123456
```

What do you think happened? Surely there is 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. Once again, we are exhaustively searching a space that doesn't contain a solution. 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 neighborhood of 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 a hard-copy dictionary<sup>22</sup> of the English language. Exhaustive enumeration would, in principle, work. You could begin 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 looking a word up in a physical (rather than online) dictionary would never proceed in this way.

Fortunately, the folks who publish hard-copy 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 to 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,  $n_1$  and  $n_2$ , either  $n_1 < n_2$  or  $n_1 > n_2$ . 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 3-5](#) contains an implementation and test of this algorithm.

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

[Figure 3-5](#) Using bisection search to approximate square root

When run for  $x = 25$ , 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. For this reason, the algorithm is called **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 30 guesses to find an acceptable answer. How about  $x = 123456789$ ? It takes only 45 guesses.

There is nothing special about 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.

Bisection search is a widely useful technique for many things other than finding roots. For example, the code in [Figure 3-6](#) uses bisection search to find an approximation to the log base 2 of  $x$  (i.e., a number, `ans`, such that  $2^{**ans}$  is close to  $x$ ). It is structured exactly like the code used to find an approximation to a square root. It first finds an interval containing a suitable answer, and then uses bisection search to efficiently explore that interval.

```

# Find lower bound on ans
lower_bound = 0
while 2**lower_bound < x:
    lower_bound += 1
low = lower_bound - 1
high = lower_bound + 1
# Perform bisection search
ans = (high + low)/2
while abs(2**ans - x) >= epsilon:
    if 2**ans < x:
        low = ans
    else:
        high = ans
    ans = (high + low)/2
print(ans, 'is close to the log base 2 of', x)

```

[Figure 3-6](#) Using bisection search to estimate log base 2

Bisection search is an example of a **successive approximation** method. Such methods work by making a sequence of guesses with the property that each guess is closer to a correct answer than the previous guess. We will look at an important successive approximation algorithm, Newton's method, later in this chapter.

**Finger exercise:** What would the code in [Figure 3-5](#) do if  $x = -25$ ?

**Finger exercise:** What would have to be changed to make the code in [Figure 3-5](#) 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.

**Finger exercise:** The Empire State Building is 102 stories high. A man wanted to know the highest floor from which he could drop an egg without the egg breaking. He proposed to drop an egg from the top floor. If it broke, he would go down a floor, and try it again. He would do this until the egg did not break. At worst, this method requires 102 eggs. Implement a method that at worst uses seven eggs.

---

### 3.3 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')
```

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

```
0.999999999999999 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 is the  $10^1$  place, etc. For example, the sequence of decimal digits 302 represents  $3 \cdot 10^2 + 0 \cdot 10^1 + 2 \cdot 10^0$ . How many different numbers can be represented by a sequence of length  $n$ ? A sequence of length 1 can represent any one of 10 numbers (0-9); a sequence of length 2 can represent 100 different numbers (0-99). More generally, a sequence of length  $n$  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 is the  $2^1$  place, etc. For example, the sequence of binary digits 101 represents  $1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 5$ . 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 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 computers use a binary representation and people a decimal representation can lead to occasional cognitive dissonance.

In 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  $1949 \cdot 10^{-3}$ .

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. They represent the significant digits and exponents in binary rather than decimal, and raise  $2$  rather than  $10$  to the exponent. For example, the number represented by the decimal digits  $0.625$  ( $5/8$ ) would be represented as the pair  $(101, -11)$ ; because  $101$  is the binary representation of the number  $5$  and  $-11$  is the binary representation of  $-3$ , the pair  $(101, -11)$  stands for  $5 \cdot 2^{-3} = 5/8 = 0.625$ .

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) uses to represent floating-point numbers, it can 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

```
11001100110011001100110011001100110011001100110011001100110011001
```

This is equivalent to the decimal number

```
0.100000000000000055511151231257827021181583404541015625
```

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`. This explains why the `else` clause was executed. But why did it decide that `x` was less than `1.0` when the floating-point representation of `0.1` is slightly greater than `0.1`? Because during some iteration of the loop, Python ran out of significant digits and did some rounding, which happened to be downwards. What gets printed if we add to the end of the `else` clause the code `print x == 10.0*0.1?` It prints `False`. 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`.

By the way, if you want to explicitly round a floating-point number, use the `round` function. The expression `round(x, num_digits)` returns the floating-point number equivalent to

rounding the value of `x` to `num_digits` 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.

Does the difference between real and floating-point numbers really matter? Most of the time, mercifully, it does not. There are few situations where the difference between `0.999999999999999`, `1.0`, and `1.000000000000001` matter. 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 okay, 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.

---

## 3.4 Newton–Raphson<sup>23</sup>

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.<sup>24</sup> It can be used to find the real roots of many functions, but we will 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 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 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  $2$ ).

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  $x^2 - 24$  is close to  $0$ .

Newton proved a theorem that implies that if a value, call it  $\text{guess}$ , is an approximation to a root of a polynomial, then  $\text{guess} - p(\text{guess})/p'(\text{guess})$ , where  $p'$  is the first derivative of  $p$ , is a better approximation than  $\text{guess}$ .

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  $x$ . For example, the first derivative of a constant is  $0$ , because the value of a constant doesn't change. For any term  $c^*x^p$ , the first derivative of that term is  $c^*p^*x^{p-1}$ . So, the first derivative of a polynomial of the form

$$c_1 * x^p + c_2 * x^{p-1} + \dots + cx + k$$

is

$$c_1 * p * x^{p-1} + c_2 * (p - 1)x^{p-2} + \dots + c$$

To find the square root of a number, say  $k$ , we need to find a value  $x$  such that  $x^2 - k = 0$ . The first derivative of this polynomial is simply  $2x$ . Therefore, we know that we can improve on the current guess by choosing as our next guess  $\text{guess} - (\text{guess}^2 - k)/2 * \text{guess}$ . [Figure 3-7](#) contains code illustrating how to use this method 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
guess = k/2
while abs(guess**2 - k) >= epsilon:
    guess = guess - (((guess**2) - k)/(2*guess))
print('Square root of', k, 'is about', guess)
```

[Figure 3-7](#) 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 far more efficient.)

---

## 3.5 Terms Introduced in Chapter

decrementing function  
guess-and-check  
exhaustive enumeration  
approximation  
total ordering  
bisection search  
successive approximation  
binary numbers  
bit  
switch  
floating point  
significant digits  
exponent

precision

rounding

Newton–Raphson

polynomial

coefficient

degree

root

---

- 22** If you have never seen such a thing, they can still be found in brick and mortar libraries.
- 23** If you haven't previously encountered polynomials or derivatives, you might find this brief section slow sledding. It is self-contained, so you should be able to work your way through it. However, if you choose to skip it, it will not pose a problem later in the book. Do, at least, give it a try. It's a cool algorithm.
- 24** Newton devised a variant of this algorithm around 1669. Joseph Raphson published a different variant about the same time as Newton. Remarkably Raphson's variant is still widely used today. (Perhaps even more remarkably, Stradivari started making violins the same year, and some of his violins are also in use today.)

# 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 using computation, it can be solved using only those linguistic mechanisms you have already seen.

But just because something can be done, doesn't mean it should be done! While any computation can, in principle, be implemented using only these mechanisms, doing so is wildly impractical. In the last chapter, we looked at an algorithm for finding an approximation to the square root of a positive number, see [Figure 4-1](#).

```
#Find approximation to square root of x
if x < 0:
    print('Does not exist')
else:
    low = 0
    high = max(1, x)
    ans = (high + low)/2
    while abs(ans**2 - x) >= epsilon:
        if ans**2 < x:
            low = ans
        else:
            high = ans
        ans = (high + low)/2
    print(ans, 'is close to square root of', x)
```

[Figure 4-1](#) Using bisection search to approximate square root of x

This is a reasonable piece of code, but it lacks general utility. It works only for the values assigned to 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. 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 computes square roots in two different places), the program would contain multiple chunks of almost identical code.

[Figure 4-2](#) adapts the code in [Figure 4-1](#) to print the sum of the square root of `x1` and the cube root of `x2`. The code works, but it's not pretty.

```

# Find square root of x1
if x1 < 0:
    print('Does not exist')
else:
    low = 0
    high = max(1, x1)
    ans = (high + low)/2
    while abs(ans**2 - x1) >= epsilon:
        if ans**2 < x1:
            low = ans
        else:
            high = ans
        ans = (high + low)/2
x1_root = ans
x2 = -8
# Find cube root of x2
if x2 < 0:
    is_pos = False
    x2 = -x2
else:
    is_pos = True
low = 0
high = max(1, x2)
ans = (high + low)/2
while abs(ans**3 - x2) >= epsilon:
    if ans**3 < x2:
        low = ans
    else:
        high = ans
    ans = (high + low)/2
if is_pos:
    x2_root = ans
else:
    x2_root = -ans
    x2 = -x2
print('Sum of square root of', x1, 'and cube root of', x2,
      'is close to', x1_root + x2_root)

```

[Figure 4-2](#) Summing a square root and a cube root

The more code a program contains, the more chance something can go wrong, and the harder the code is to maintain. Imagine, for example, that there were an error in the initial implementation of bisection search, and that the error came to light when testing the program. It would be all too easy to fix the implementation in one place and not notice similar code elsewhere that needed repair.

Fortunately, 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<sup>25</sup>

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

For example, we could define the function `max_val`<sup>26</sup> by the code

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

`def` is a reserved word that tells Python a function is about to be defined. The function name (`max_val` in this example) is simply a name that is used to refer to the function. The PEP 8 convention is that function names should be in all lowercase with words separated by underscores to improve readability.

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

```
max_val(3, 4)
```

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

The function body is any piece of Python code.<sup>27</sup> 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 returned by the invoked function. For example, the value of the expression `max_val(3, 4) * max_val(3, 2)` is 12, because the first invocation of `max_val` 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 `max_val(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 executed.
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`.)<sup>28</sup>
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 allow programmers to write code that accesses not specific objects, but instead whatever objects the caller of the function chooses to use as actual parameters. This is called **lambda abstraction**.<sup>29</sup>

[Figure 4-3](#) contains a function that has three formal parameters and returns a value, call it `result`, such that `abs(result**power - x) >= epsilon`.

```
def find_root(x, power, epsilon):
    # Find interval containing answer
    if x < 0 and power%2 == 0:
        return None #Negative number has no even-powered roots
    low = min(-1, x)
    high = max(1, x)
    # Use bisection search
    ans = (high + low)/2
    while abs(ans**power - x) >= epsilon:
        if ans**power < x:
            low = ans
        else:
            high = ans
        ans = (high + low)/2
    return ans
```

[Figure 4-3](#) A function for finding roots

[Figure 4-4](#) contains code that can be used to test whether `find_root` works as intended. The testing function `test_find_root` is about the same length as `find_root` itself. To inexperienced programmers, writing **test functions** 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 sitting at a keyboard and typing test cases into the shell over and over during **debugging** (the process of finding out why a program does not work, and then fixing it). Notice that because we are invoking `test_find_root` with three tuples (i.e., sequences of values) of length three, one call checks 27 combinations of parameters. Finally, because `test_find_root` checks whether `find_root` is returning an appropriate answer and reports the result, it saves the programmer from the tedious and error-prone task of visually inspecting each output and checking it for correctness. We return to the subject of testing in Chapter 8.

**Finger exercise:** Use the `find_root` function in [Figure 4-3](#) to print the sum of approximations to the square root of 25, the cube root of -8, and the fourth root of 16. Use 0.001 as epsilon.

**Finger exercise:** Write a function `is_in` 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` operator `in`.

**Finger exercise:** Write a function to test `is_in`.

```
def test_find_root(x_vals, powers, epsilons):
    for x in x_vals:
        for p in powers:
            for e in epsilons:
                result = find_root(x, p, e)
                if result == None:
                    val = 'No root exists'
                else:
                    val = 'Okay'
                    if abs(result**p - x) > e:
                        val = 'Bad'
                print(f'x = {x}, power = {p}, epsilon = {e}: {val}')

x_vals = (0.25, 8, -8)
powers = (1, 2, 3)
epsilons = (0.1, 0.001, 1)
test_find_root(x_vals, powers, epsilons)
```

[Figure 4-4](#). Code to test `find_root`

### 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 one we have used so 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 print_name(first_name, last_name, reverse):
    if reverse:
        print(last_name + ', ' + first_name)
    else:
        print(first_name, last_name)
```

The function `print_name` assumes that `first_name` and `last_name` are strings and that `reverse` is a Boolean. If `reverse == True`, it prints `last_name, first_name`; otherwise it prints `first_name last_name`.

Each of the following is an equivalent invocation of `print_name`:

```
print_name('Olga', 'Puchmajerova', False)
print_name('Olga', 'Puchmajerova', reverse = False)
print_name('Olga', last_name = 'Puchmajerova', reverse =
False)
print_name(last_name = 'Puchmajerova', first_name = '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

```
print_name('Olga', last_name = 'Puchmajerova', False)
```

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

```
def print_name(first_name, last_name, reverse = False):
    if reverse:
        print(last_name + ', ' + first_name)
    else:
        print(first_name, last_name)
```

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

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

will print

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

The last two invocations of `print_name` are semantically equivalent. The last one has the advantage of providing some documentation for the perhaps mysterious argument `True`. More generally, using keyword arguments reduces the risk of unintentionally binding an actual parameter to the wrong formal parameter. The line of code

```
print_name(last_name = 'Puchmajerova', first_name = 'Olga')
```

leaves no ambiguity about the intent of the programmer who wrote it. This is useful because calling a function with the right arguments in the wrong order is a common blunder.

The value associated with a default parameter is computed at function definition time. This can lead to surprising program behavior, as we discuss in Section 5.3.

**Finger exercise:** Write a function `mult` that accepts either one or two ints as arguments. If called with two arguments, the function prints the product of the two arguments. If called with one argument, it prints that argument.

### 4.1.3 Variable Number of Arguments

Python has a number of built-in functions that operate on a variable number of arguments. For example,

```
min(6,4)  
min(3,4,1,6)
```

are both legal (and evaluate to what you think they do). Python makes it easy for programmers to define their own functions that accept a variable number of arguments. The **unpacking operator** `*` allows a function to accept a variable number of positional arguments. For example,

```
def mean(*args):  
    # Assumes at least one argument and all arguments are  
    # numbers  
    # Returns the mean of the arguments  
    tot = 0  
    for a in args:  
        tot += a  
    return tot/len(args)
```

prints `1.5 -1.0`. Note that the name following the `*` in the argument list need not be `args`. It can be any name. For `mean`, it might have been more descriptive to write `def mean(*numbers)`.

#### 4.1.4 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` in the context of the function body of `f`. 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 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 the 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.

- 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.
- When the function completes, its stack frame goes away.

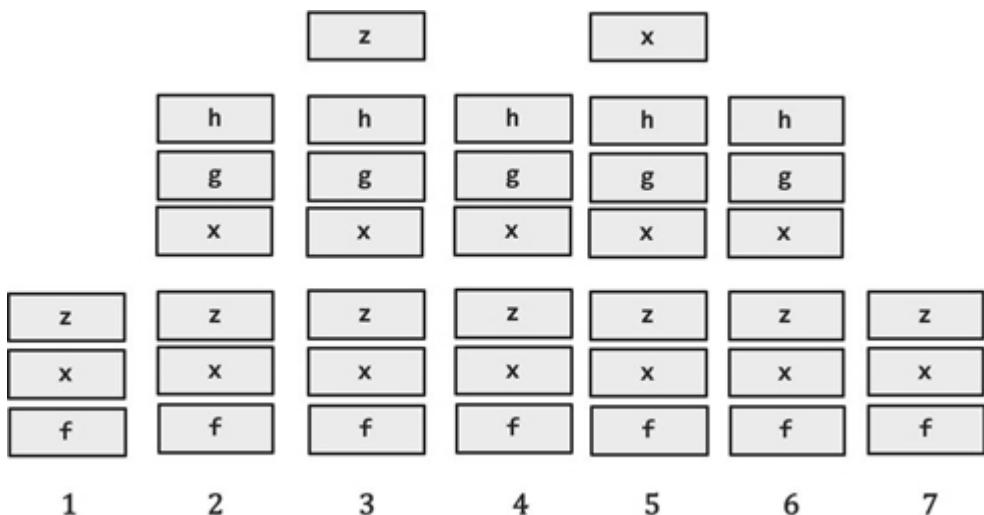
In Python, you can always determine the scope of a name by looking at the program text. This is called **static** or **lexical scoping**. [Figure 4-5](#) contains an example illustrating Python's scope rules. The history of the stack frames associated with the code is depicted in [Figure 4-6](#).

```
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 4-5](#) Nested scopes

The first column in [Figure 4-6](#) 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 4-6](#) 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 `x`, 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 a formal parameter of the function or a variable 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` in this case) 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. Because of this “last in first out” (**LIFO**) behavior, we refer to it as a **stack**. (Imagine cooking a stack of pancakes. When the first one comes off the griddle, the chef places it on a serving plate. As each successive pancake comes off the griddle, it is stacked on top of the pancakes already on the serving plate. When it comes time to eat the pancakes, the first pancake served will be the one on top of the stack, the last one added—leaving the penultimate pancake added to the stack as the new top pancake, and the next one to be served.)



Returning to our Python example, `g` is now 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 4-5](#) 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.<sup>39</sup> Consider 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.

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 only want to use variables that are local to a function, and the subtleties of scoping will be irrelevant. In fact, if your program depends upon some subtle scoping rule, you might consider rewriting to avoid doing so.

---

## 4.2 Specifications

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 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 parameters. For example, the specification of `find_root` might require that `power` be a positive integer.
- **Guarantees:** These describe conditions that must be met by the function, provided it has been called in a way that satisfies the assumptions. For example, the specification of `find_root` might guarantee that it returns `None` if asked to find a root that doesn't exist (e.g., the square root of a negative number).

Functions are a way of creating computational elements that we can think of as primitives. They provide 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.<sup>31</sup> 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 25 lectures. One way to do this would be to recruit 25 professors and ask each of them to prepare a one-hour lecture on their favorite topic. Though you might get 25 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 15 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, you need to let everyone know what everyone else is doing, without generating so much work that nobody is willing to participate. This is where abstraction comes in. You could write 25 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 use teams of programmers to get things done. Given a specification of a module, a programmer can work on implementing that module without worrying about what the other programmers on the team are doing. Moreover, the other programmers can use the specification to start writing code that uses the module without worrying about how the module will be implemented.

[Figure 4-7](#) adds a specification to the implementation of `find_root` in [Figure 4-3](#).

```

def find_root(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"""
    # Find interval containing answer
    if x < 0 and power%2 == 0:
        return None
    low = min(-1, x)
    high = max(1, x)
    # Use bisection search
    ans = (high + low)/2
    while abs(ans**power - x) >= epsilon:
        if ans**power < x:
            low = ans
        else:
            high = ans
        ans = (high + low)/2
    return ans

```

[Figure 4-7](#) A function definition with a specification

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`.

One of the nice things about Python IDEs is that they provide an interactive tool for asking about the built-in objects. If you want to know what a specific function does, you need only type `help(object)` into the console window. For example, `help(abs)` produces the text

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

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

This tells us that `abs` is a function that maps a single argument to its absolute value. (The `/` in the argument list means that the argument must be positional.) If you enter `help()`, an interactive help session is started, and the interpreter will present the prompt `help>` in the console window. An advantage of the interactive mode is that you can get help on Python constructs that are not objects. E.g.,

```
help> if
The "if" statement
*****
```

The "if" statement is used for conditional execution:

```
if_stmt ::= "if" expression ":" suite
          ("elif" expression ":" suite)*
          ["else" ":" suite]
```

It selects exactly one of the suites by evaluating the expressions one by one until one is found to be true (see section Boolean operations for the definition of true and false); then that suite is executed (and no other part of the "if" statement is executed or evaluated). If all expressions are false, the suite of the "else" clause, if present, is executed.

Related help topics: TRUTHVALUE

Interactive help can be exited by entering `quit`.

If the code in [Figure 4-4](#) had been loaded into an IDE, typing `help(find_root)` in the shell would display

```
find_root(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
```

The specification of `find_root` is an abstraction of all the possible implementations that meet the specification. Clients of `find_root` can assume that the implementation meets the specification, but they should assume nothing more. For example, clients can assume that the call `find_root(4, 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 is a perfect square, the value returned might not be 2 or -2. Crucially, if the assumptions of the specification are not satisfied, nothing can be assumed about the

effect of calling the function. For example, the call `find_root(8, 3, 0)` could return `2`. But it could also crash, run forever, or return some number nowhere near the cube root of `8`.

**Finger exercise:** Using the algorithm of Figure 3-6, write a function that satisfies the specification

```
def log(x, base, epsilon):
    """Assumes x and epsilon int or float, base an int,
       x > 1, epsilon > 0 & power >= 1
       Returns float y such that base**y is within epsilon
       of x."""

```

---

## 4.3 Using Functions to Modularize Code

So far, all of the functions we have implemented have been small. They fit nicely on a single page. As we implement more complicated functionality, it is convenient to split functions into multiple functions, each of which does one simple thing. To illustrate this idea we, somewhat superfluously, split `find_root` into three separate functions, as shown in [Figure 4-8](#). Each of the functions has its own specification and each makes sense as a stand-alone entity. The function `find_root_bounds` finds an interval in which the root must lie, `bisection_solve` uses bisection search to search this interval for an approximation to the root, and `find_root` simply calls the other two and returns the root.

Is this version of `find_root` easier to understand than the original monolithic implementation? Probably not. A good rule of thumb is that if a function fits comfortably on a single page, it probably doesn't need to be subdivided to be easily understood.

```

def find_root_bounds(x, power):
    """x a float, power a positive int
       return low, high such that low**power <=x and high**power >= x
    """
    low = min(-1, x)
    high = max(1, x)
    return low, high

def bisection_solve(x, power, epsilon, low, high):
    """x, epsilon, low, high are floats
       epsilon > 0
       low <= high and there is an ans between low and high s.t.
           ans**power is within epsilon of x
       returns ans s.t. ans**power within epsilon of x"""
    ans = (high + low)/2
    while abs(ans**power - x) >= epsilon:
        if ans**power < x:
            low = ans
        else:
            high = ans
        ans = (high + low)/2
    return ans

def find_root(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:
        return None #Negative number has no even-powered roots
    low, high = find_root_bounds(x, power)
    return bisection_solve(x, power, epsilon, low, high)

```

[Figure 4-8](#) Splitting `find_root` into multiple functions

---

## 4.4 Functions as Objects

In Python, functions are **first-class objects**. That means 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 returned by functions; etc.

Using functions as arguments allows a style of coding called **higher-order programming**. It allows us to write functions that are more generally useful. For example, the function `bisection_solve` in [Figure 4-8](#) can be rewritten so that it can be applied to tasks other than root finding, as shown in [Figure 4-9](#).

```
def bisection_solve(x, eval_ans, epsilon, low, high):
    """x, epsilon, low, high are floats
    epsilon > 0
    eval_ans a function mapping a float to a float
    low <= high and there is an ans between low and high s.t.
        eval(ans) is within epsilon of x
    returns ans s.t. eval(ans) within epsilon of x"""
    ans = (high + low)/2
    while abs(eval_ans(ans) - x) >= epsilon:
        if eval_ans(ans) < x:
            low = ans
        else:
            high = ans
        ans = (high + low)/2
    return ans
```

[Figure 4-9](#) Generalizing `bisection_solve`

We start by replacing the integer parameter `power` by a function, `eval_ans`, that maps floats to floats. We then replace every instance of the expression `ans**power` by the function call `eval_ans(ans)`.

If we wanted to use the new `bisection_solve` to print an approximation to the square root of 99, we could run the code

```
def square(ans):
    return ans**2
low, high = find_root_bounds(99, 2)
print(bisection_solve(99, square, 0.01, low, high))
```

Going to the trouble of defining a function to do something as simple as squaring a number seems a bit silly. Fortunately, 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, we could replace the above call to `bisection_solve` with

```
print(bisection_solve(99, lambda ans: ans**2, 0.01, low,
high))
```

**Finger exercise:** Write a lambda expression that has two numeric parameters. If the second argument equals zero, it should return `None`. Otherwise it should return the value of dividing the first argument by the second argument. Hint: use a conditional expression.

Since functions are first-class objects, they can be created and returned within functions. For example, given the function definition

```
def create_eval_ans():
    power = input('Enter a positive integer: ')
    return lambda ans: ans**int(power)
```

the code

```
eval_ans = create_eval_ans()
print(bisection_solve(99, eval_ans, 0.01, low, high))
```

will print an approximation to the  $n^{\text{th}}$  root of 99, where  $n$  is a number entered by a user.

The way we have generalized `bisection_solve` means that it can now be used not only to search for approximations to roots, but to search for approximations to any monotonic<sup>32</sup> function that maps floats to floats. For example, the code in [Figure 4-10](#) uses `bisection_solve` to find approximations to logarithms.

```

def log(x, base, epsilon):
    """Assumes x and epsilon int or float, base an int,
       x > 1, epsilon > 0 & power >= 1
       Returns float y such that base**y is within epsilon of x."""
    def find_log_bounds(x, base):
        upper_bound = 0
        while base**upper_bound < x:
            upper_bound += 1
        return upper_bound - 1, upper_bound
    low, high = find_log_bounds(x, base)
    return bisection_solve(x, lambda ans: base**ans, epsilon, low, high)

```

[Figure 4-10](#) Using `bisection_solve` to approximate logs

Notice that the implementation of `log` includes the definition of a local function, `find_log_bounds`. This function could have been defined outside of `log`, but since we don't expect to use it in any other context, it seemed better not to.

---

## 4.5 Methods, Oversimplified

Methods are function-like objects. They can be called with parameters, they can return values, and they can have side effects. They do differ from functions in some important ways, which we will discuss in Chapter 10.

For now, think of methods as providing a peculiar syntax for a function call. Instead of putting the first argument inside parentheses following the function name, we use **dot notation** to place that argument before the function name. We introduce methods here because many useful operations on built-in types are methods, and therefore invoked using dot notation. For example, if `s` is a string, the `find` method can be used to find the index of the first occurrence of a substring in `s`. So, if `s` were '`abcabc`', the invocation `s.find('bc')` would return `1`. Attempting to treat `find` as a function, e.g., invoking `find(s, 'bc')`, produces the error message `NameError: name 'find' is not defined`.

**Finger exercise:** What does `s.find(sub)` return if `sub` does not occur in `s`?

**Finger exercise:** Use `find` to implement a function satisfying the specification

```
def find_last(s, sub):
    """s and sub are non-empty strings
       Returns the index of the last occurrence of sub in s.
       Returns None if sub does not occur in s"""

```

---

## 4.6 Terms Introduced in Chapter

function definition

formal parameter

actual parameter

argument

function invocation (function call)

return statement

point of execution

lambda abstraction

test function

debugging

positional argument

keyword argument

default parameter value

unpacking operator (\*)

name space

scope

local variable

symbol table  
stack frame  
static (lexical) scoping  
stack (LIFO)  
specification  
client  
assumption  
guarantee  
decomposition  
abstraction  
docstring  
help function  
first-class object  
higher-order programming  
lambda expression  
method  
dot notation

---

- 25** Recall that italic is used to denote concepts rather than actual code.
- 26** In practice, you would probably use the built-in function `max`, rather than define your own function.
- 27** As we will see later, this notion of function is far more general than what mathematicians call a function. It was first popularized by the programming language Fortran 2 in the late 1950s.
- 28** Later in the book, we discuss two other mechanisms for exiting a function, `raise` and `yield`.

- 29 The name “lambda abstraction” is derived from mathematics developed by Alonzo Church in the 1930s and 1940s. The name is derived from Church's use of the Greek letter lambda ( $\lambda$ ) to denote function abstraction. During his lifetime, Church gave different explanations of why he chose the symbol lambda. My favorite is, “eeny, meeny, miny, moe.”
- 30 The wisdom of this language design decision is debatable.
- 31 “Where ignorance is bliss, 'tis folly to be wise.”—Thomas Gray
- 32 A function from numbers to numbers is monotonically increasing (decreasing) if the value returned by the function increases (decreases) as the value of its argument increases (decreases).

# 5

## STRUCTURED TYPES AND MUTABILITY

The programs we have looked at so 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. We 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 simple generalization of `str`. The other three—`list`, `range`, and `dict`—are more interesting. We also return to the topic of higher-order programming 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 think that the tuple containing the single value `1` would be written `(1)`. But, to quote H.R. Haldeman quoting Richard Nixon, “it would be wrong.”<sup>33</sup> Since parentheses are used to group expressions, `(1)` is 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 `'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. And the `in` operator can be used to test if a tuple contains a specific value. For example, the following code

```
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
print(intersect((1, 'a', 2), ('b', 2, 'a')))
```

prints `('a', 2)`.

### 5.1.1 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, the statement `x, y = (3, 4)`, will bind `x` 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 with functions that return multiple value. Consider the function definition

```
def find_extreme_divisors(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 & n2. If no
       common divisor,
               other than 1, returns (None, None)"""
    min_val, max_val = None, None
    for i in range(2, min(n1, n2) + 1):
        if n1%i == 0 and n2%i == 0:
            if min_val == None:
                min_val = i
            max_val = i
    return min_val, max_val
```

## The multiple assignment statement

```
min_divisor, max_divisor = find_extreme_divisors(100, 200)
```

will bind `min_divisor` to 2 and `max_divisor` to 200.

---

## 5.2 Ranges and Iterables

As discussed in Section 2.6, the function `range` produces an object of type `range`. Like strings and tuples, objects of type `range` are immutable. 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.

In Python 3, `range` is a special case of an **iterable object**. All iterable types have a method,<sup>34</sup> `__iter__` that returns an object of **type iterator**. The iterator can then be used in a `for` loop to return a sequence of objects, one at a time. For example, tuples are iterable, and the `for` statement

```
for elem in (1, 'a', 2, (3, 4)):
```

creates an iterator that will return the elements of the tuple one at a time. Python has many built-in iterable types, including strings, lists, and dictionaries.

Many useful built-in functions operate on iterables. Among the more useful are `sum`, `min`, and `max`. The function `sum` can be applied to iterables of numbers. It returns the sum of the elements. The

functions `max` and `min` can be applied to iterables for which there is a well-defined ordering on the elements.

**Finger exercise:** Write an expression that evaluates to the mean of a tuple of numbers. Use the function `sum`.

---

## 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.

Since lists are iterables, we can use a `for` statement to iterate over the elements in `list`. So, for example, the code

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

produces the output,

```
I did it all
4
Love
```

We can also index into lists and slice lists, just as we can for tuples. For example, the code

```
L1 = [1, 2, 3]
L2 = L1[-1::-1]
for i in range(len(L1)):
    print(L1[i]*L2[i])
```

prints

```
3
4
3
```

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

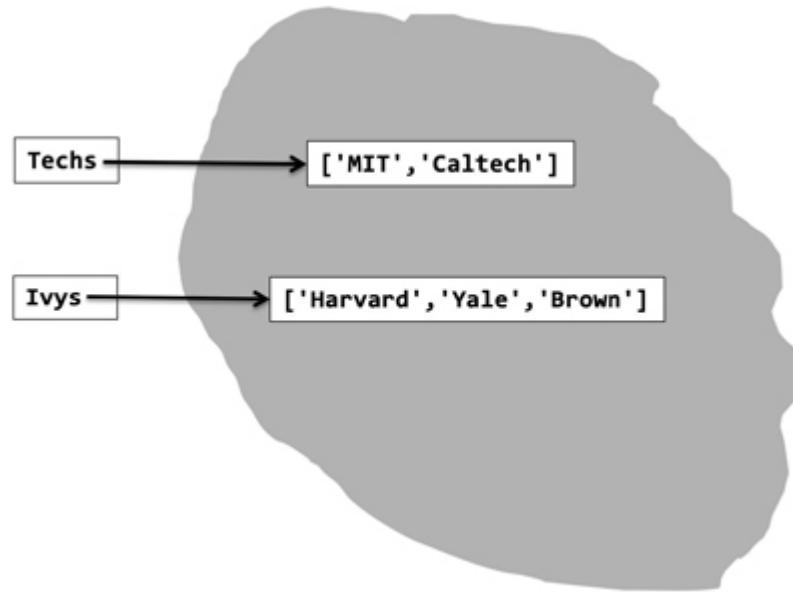
Lists differ from tuples in one hugely important way: lists are **mutable**. In contrast, tuples and strings are **immutable**. Many operators can be used to create objects of immutable types, and variables can be bound to objects of these types. But objects of immutable types cannot be modified after they are created. On the other hand, objects of mutable types 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. And perhaps the following set of examples will also help.

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']]
```

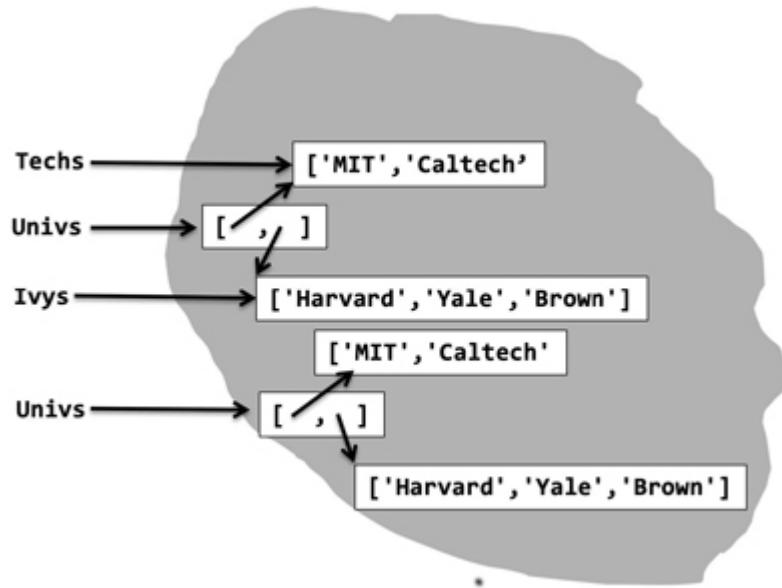
also create new lists and bind variables to them. The elements of these lists are themselves lists. The three print statements

```
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** by comparing their `id`. A simpler way to test for object equality is to use the `is` operator. When we run the code

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

it prints

```
True
False
False
Id of Univs = 4946827936
Id of Univs1 = 4946612464
```

(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]))
```

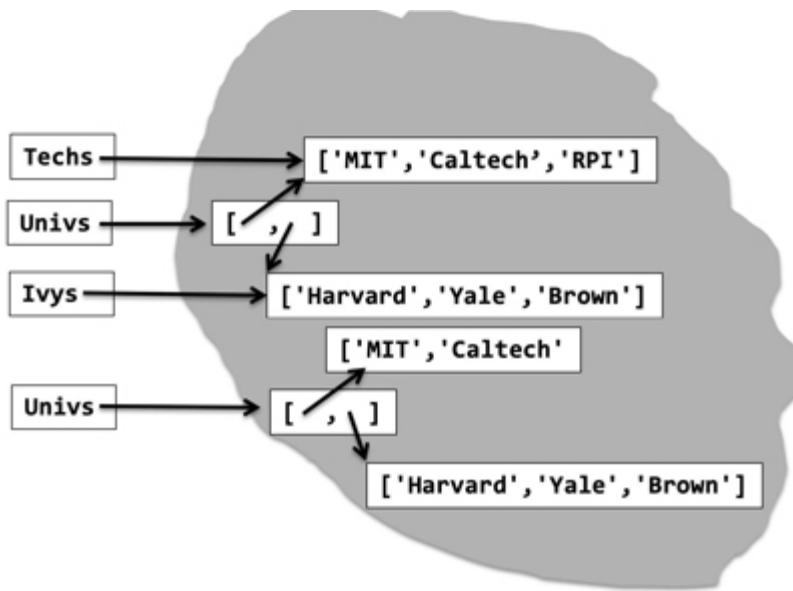
which prints

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

Why the big fuss about the difference between value and object equality? It matters because lists are mutable. Consider the code

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

The `append` method for lists has a **side effect**. Rather than create a new list, it mutates the existing list, `Techs`, by adding a new element, the string '`RPI`' in this example, 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 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. We 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. For example, what do you think is printed by

```
L1 = [[]]*2
L2 = [[], []]
for i in range(len(L1)):
    L1[i].append(i)
    L2[i].append(i)
print('L1 =', L1, 'but', 'L2 =', L2)
```

It prints `L1 = [[0, 1], [0, 1]]` but `L2 = [[0], [1]]`. Why? Because the first assignment statement creates a list with two elements, each of which is the same object, whereas the second assignment statement creates a list with two different objects, each of which is initially equal to an empty list.

**Finger exercise:** What does the following code print?

```
L = [1, 2, 3]
L.append(L)
print(L is L[-1])
```

The interaction of aliasing and mutability with default parameter values is something to watch out for. Consider the code

```

def append_val(val, list_1 = []):
    list_1.append(val)
    print(list_1)

append_val(3)
append_val(4)

```

You might think that the second call to `append_val` would print the list `[4]` because it would have appended `4` to the empty list. In fact, it will print `[3, 4]`. This happens because, at function definition time, a new object of type `list` is created, with an initial value of the empty list. Each time `append_val` is invoked without supplying a value for the formal parameter `list_1`, the object created at function definition is bound to `list_1`, mutated, and then printed. So, the second call to `append_val` mutates and then prints a list that was already mutated by the first call to that function.

When we append one list to another, e.g., `Techs.append(Ivys)`, the original structure is maintained. 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 (using the `+` operator) 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 mutate `L1`.

[Figure 5-4](#) briefly describes 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 9) 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](#) Common methods associated with lists

### 5.3.1 Cloning

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

```
def remove_dups(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]
Remove_dups(L1, L2)
print('L1 =', L1)
```

You might be surprised to discover that this prints

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

During a `for` loop, 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 you 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**<sup>35</sup> (i.e., make a copy of) the list and write `for e1 in L1[:]`. Notice that writing

```
new_L1 = L1
for e1 in new_L1:
```

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 `L.copy()` has the same value as `L[:]`. Both slicing and `copy` perform what is known as a **shallow copy**. A shallow copy creates a new list and then inserts the objects (not copies of the objects) of the list to be copied into the new list. The code

```
L = [2]
L1 = [L]
L2 = L1[:]
L.append(3)
print(f'L1 = {L1}, L2 = {L2}')
```

prints `L1 = [[2, 3]] L2 = [[2, 3]]` because both `L1` and `L2` contain the object that was bound to `L` in the first assignment statement.

If the list to be copied contains mutable objects that you also want to copy, import the standard library module `copy` and use the function `copy.deepcopy` to make a **deep copy**. The method `deepcopy` creates a new list and then inserts copies of the objects in the list to be copied into the new list. If we replace the third line in the above code by `L2 = copy.deepcopy(L1)`, it will print `L1 = [[2, 3]], L2 = [[2]]`, because `L1` would not contain the object to which `L` is bound.

Understanding `copy.deepcopy` is tricky if the elements of a list are lists containing lists (or any mutable type). Consider

```
L1 = [2]
L2 = [[L1]]
L3 = copy.deepcopy(L2)
L1.append(3)
```

The value of `L3` will be `[[[2]]]` because `copy.deepcopy` creates a new object not only for the list `L1`, but also for the list `L1`. I.e., it makes copies all the way to the bottom—most of the time. Why “most of the time?” The code

```
L1 = [2]
L1.append(L1)
```

creates a list that contains itself. An attempt to make copies all the way to the bottom would never terminate. To avoid this problem, `copy.deepcopy` makes exactly one copy of each object, and then uses that copy for each instance of the object. This matters even when lists do not contain themselves. For example,

```
L1 = [2]
L2 = [L1, L1]
L3 = copy.deepcopy(L2)
L3[0].append(3)
print(L3)
```

prints `[[2, 3], [2, 3]]` because `copy.deepcopy` makes one copy of `L1` and uses it both times `L1` occurs in `L2`.

### 5.3.2 List Comprehension

**List comprehension** provides a concise way to apply an operation to the sequence values provided by iterating over an iterable value. It creates a new list in which each element is the result of applying a given operation to a value from an iterable (e.g., the elements in another list). It is an expression of the form

```
[expr for elem in iterable if test]
```

Evaluating the expression is equivalent to invoking the function

```

def f(expr, old_list, test = lambda x: True):
    new_list = []
    for e in iterable:
        if test(e):
            new_list.append(expr(e))
    return new_list

```

For example, `[e**2 for e in range(6)]` evaluates to `[0, 1, 4, 9, 16, 25]`, `[e**2 for e in range(8) if e%2 == 0]` evaluates to `[0, 4, 16, 36]`, and `[x**2 for x in [2, 'a', 3, 4.0] if type(x) == int]` evaluates to `[4, 9]`.

List comprehension provides a convenient way to initialize lists. For example, `[] for _ in range(10)` generates a list containing 10 distinct (i.e., non-aliased) empty lists. The variable name `_` indicates that the values of that variable are not used in generating the elements of list, i.e., it is merely a placeholder. This convention is common in Python programs.

Python allows multiple `for` statements within a list comprehension. Consider the code

```

L = [(x, y)
      for x in range(6) if x%2 == 0
      for y in range(6) if y%3 == 0]

```

The Python interpreter starts by evaluating the first `for`, assigning to `x` the sequence of values `0, 2, 4`. For each of these three values of `x`, it evaluates the second `for` (which generates the sequence of values `0, 3` each time). It then adds to the list being generated the tuple `(x, y)`, producing the list

```

[(0, 0), (0, 3), (2, 0), (2, 3), (4, 0), (4, 3)]

```

Of course, we can produce the same list without list comprehension, but the code is considerably less compact:

```

L = []
for x in range(6):
    if x%2 == 0:
        for y in range(6):
            if y%3 == 0:
                L.append((x, y))

```

The following code is an example of nesting a list comprehension within a list comprehension.

```
print([[x, y) for x in range(6) if x%2 == 0]
      for y in range(6) if y%3 == 0])
```

It prints `[(0, 0), (2, 0), (4, 0)], [(0, 3), (2, 3), (4, 3)]`.

It takes practice to get comfortable with nested list comprehensions, but they can be quite useful. Let's use nested list comprehensions to generate a list of all prime numbers less than 100. The basic idea is to use one comprehension to generate a list of all of the candidate numbers (i.e., 2 through 99), a second comprehension to generate a list of the remainders of dividing a candidate prime by each potential divisor, and the built-in function `all` to test if any of those remainders is 0.

```
[x for x in range(2, 100) if all(x % y != 0 for y in
range(3, x))]
```

Evaluating the expression is equivalent to invoking the function

```
def gen_primes():
    primes = []
    for x in range(2, 100):
        is_prime = True
        for y in range(3, x):
            if x%y == 0:
                is_prime = False
        if is_prime:
            primes.append(x)
    return primes
```

**Finger exercise:** Write a list comprehension that generates all non-primes between 2 and 100.

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 rarely a desirable property for a program.

---

## 5.4 Higher-Order Operations on Lists

In Section 4.4 we introduced the notion of higher-order programming. It can be particularly convenient with lists, as shown in [Figure 5-5](#).

```
def apply_to_each(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.')
apply_to_each(L, abs)
print('L =', L)
print('Apply int to each element of', L)
apply_to_each(L, int)
print('L =', L)
print('Apply squaring to each element of', L)
apply_to_each(L, lambda x: x**2)
print('L =', L)
```

[Figure 5-5](#) Applying a function to elements of a list

The function `apply_to_each` 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. And the third time it is called, it replaces each element by the result of applying a function defined using `lambda`. 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 squaring to each element of [1, 2, 3].
L = [1, 4, 9]
```

Python has a built-in higher-order function, `map`, that is similar to, but more general than, the `apply_to_each` function defined in [Figure 5-5](#). 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. It is frequently used in lieu of a list comprehension. For example, `list(map(str, range(10)))` is equivalent to `[str(e) for e in range(10)]`.

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

```
for i in map(lambda x: x**2, [2, 6, 4]):  
    print(i)
```

prints

```
4  
36  
16
```

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)
```

prints

```
1  
28  
9
```

**Finger exercise:** Implement a function satisfying the following specification. Hint: it will be convenient to use `lambda` in the body of the implementation.

```
def f(L1, L2):  
    """L1, L2 lists of same length of numbers  
    returns the sum of raising each element in L1  
    to the power of the element at the same index in L2  
    For example, f([1,2], [2,3]) returns 9"""
```

---

## 5.5 Strings, Tuples, Ranges, and Lists

We have looked at four iterable sequence types: `str`, `tuple`, `range`, and `list`. They are similar in that objects 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  $i^{\text{th}}$  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
<code>str</code>	characters	<code>'', 'a', 'abc'</code>	No
<code>tuple</code>	any type	<code>((), (3,), ('abc', 4))</code>	No
<code>range</code>	integers	<code>range(10), range(1, 10, 2)</code>	No
<code>list</code>	any type	<code>[], [3], ['abc', 4]</code>	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.

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

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 useful built-in methods. [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.

**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 whitespace 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

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).<sup>36</sup>

---

## 5.6 Sets

**Sets** are yet another kind of collection type. They are similar to the notion of a set in mathematics in that they are unordered collections of unique elements. They are denoted using what programmers call curly braces and mathematicians call set braces, e.g.,

```
baseball_teams = {'Dodgers', 'Giants', 'Padres', 'Rockies'}
football_teams = {'Giants', 'Eagles', 'Cardinals',
'Cowboys'}
```

Since the elements of a set are unordered, attempting to index into a set, e.g., evaluating `baseball_teams[0]`, generates a runtime error. We can use a `for` statement to iterate over the elements of a set, but unlike the other collection types we have seen, the order in which the elements are produced is undefined.

Like lists, sets are mutable. We add a single element to a set using the `add` method. We add multiple elements to a set by passing a collection of elements (e.g., a list) to the `update` method. For example, the code

```
baseball_teams.add('Yankees')
football_teams.update(['Patriots', 'Jets'])
print(baseball_teams)
print(football_teams)
```

prints

```
{'Dodgers', 'Yankees', 'Padres', 'Rockies', 'Giants'}  
{'Jets', 'Eagles', 'Patriots', 'Cowboys', 'Cardinals',  
'Giants'}
```

(The order in which the elements appear is not defined by the language, so you might get a different output if you run this example.)

Elements can be removed from a set using the `remove` method, which raises an error if the element is not in the set, or the `discard` method, which does not raise an error if the element is not in the set.

Membership in a set can be tested using the `in` operator. For example, `'Rockies' in baseball_teams` returns `True`. The binary methods `union`, `intersection`, `difference`, and `issubset` have their usual mathematical meanings. For example,

```
print(baseball_teams.union({1, 2}))  
print(baseball_teams.intersection(football_teams))  
print(baseball_teams.difference(football_teams))  
print({'Padres', 'Yankees'}.issubset(baseball_teams))
```

prints

```
{'Padres', 'Rockies', 1, 2, 'Giants', 'Dodgers', 'Yankees'}  
{'Giants'}  
{'Padres', 'Rockies', 'Dodgers', 'Yankees'}  
True
```

One of the nice things about sets is that there are convenient infix operators for many of the methods, including `|` for `union`, `&` for `intersect`, `-` for `difference`, `<=` for `subset`, and `>=` for `superset`. Using these operators makes code easier to read. Compare, for example,

```
print(baseball_teams | {1, 2})  
print(baseball_teams & football_teams)  
print(baseball_teams - football_teams)  
print({'Padres', 'Yankees'} <= baseball_teams)
```

to the code presented earlier, which uses dot notation to print the same values.

Not all types of objects can be elements of sets. All objects in a set must be **hashable**. An object is hashable if it has

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

All objects of Python's scalar immutable types are hashable, and no object of Python's built-in mutable types is hashable. An object of a non-scalar immutable type (e.g., a tuple) is hashable if all of its elements are hashable.

## 5.7 Dictionaries

Objects of type `dict` (short for **dictionary**) are like lists except that we index them using **keys** rather than integers. Any hashable object can be used as a key. 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,

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

will print

```
{'Jan': 1, 'Feb': 2, 'Mar': 3, 'Apr': 4, 'May': 5, 1: 'Jan',
2: 'Feb', 3: 'Mar', 4: 'Apr', 5: 'May'}
The third month is Mar
Apr and Jan are 3 months apart
```

The entries in a `dict` cannot be accessed using an index. That's why `month_numbers[1]` unambiguously refers to the entry with the key `1` rather than the second entry. Whether a key is defined in a dictionary can be tested using the `in` operator.

Like lists, dictionaries are mutable. We can add an entry by writing, for example, `month_numbers['June'] = 6` or change an entry by writing, for example, `month_numbers['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.

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, be careful about side effects. For example,

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

will print

```
I wood of wine red.
```

```

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 translate_word(word, dictionary):
    if word in dictionary:
        return dictionary[word]
    elif word != '':
        return "'' + word + ''
    return word

def translate(phrase, dicts, direction):
    UC_letters = 'ABCDEFGHIJKLMNPQRSTUVWXYZ'
    LC_letters = 'abcdefghijklmnopqrstuvwxyz'
    punctuation = ',.;:;'
    letters = UC_letters + LC_letters
    dictionary = dicts[direction]
    translation = ''
    word = ''
    for c in phrase:
        if c in letters:
            word = word + c
        elif word != '':
            if c in punctuation:
                c = c + ''
            translation = (translation +
                           translate_word(word, dictionary) + c)
            word = ''
    return f'{translation} {translate_word(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)

Many 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 tuple representing a key/value pair. We can then write a simple function that does the associative retrieval, e.g.,

```

def key_search(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 fast. It uses a technique called hashing, described in Chapter 12, to do the lookup in time that is nearly independent of the size of the dictionary.

There are multiple ways to use a `for` statement to iterate over the entries in a dictionary. If `d` is a dictionary, a loop of the form `for k in d` iterates over the keys of `d`. The order in which the keys are chosen is the order in which the keys were inserted in the dictionary.<sup>37</sup> For example,

```

capitals = {'France': 'Paris', 'Italy': 'Rome', 'Japan':
'Kyoto'}
for key in capitals:
    print('The capital of', key, 'is', capitals[key])

```

`prints`

```

The capital of France is Paris
The capital of Italy is Rome
The capital of Japan is Kyoto

```

To iterate over the values in a dictionary, we can use the method `values`. For example,

```

cities = []
for val in capitals.values():
    cities.append(val)
print(cities, 'is a list of capital cities')

```

`prints` `['Paris', 'Rome', 'Kyoto']` is a list of capital cities.

The method `values` returns an object of type `dict_values`. This is an example of a **view object**. 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, the code

```
cap_vals = capitals.values()
print(cap_vals)
capitals['Japan'] = 'Tokyo'
print(cap_vals)
```

prints

```
dict_values(['Paris', 'Rome', 'Kyoto'])
dict_values(['Paris', 'Rome', 'Toyko'])
```

Similarly, the method `keys` returns a view object of type `dict_keys`. View objects can be converted to lists, e.g., `list(capitals.keys())` returns a list of the keys in `capitals`.

To iterate over key/value pairs, we use the method `items`. This method returns a view object of type `dict_items`. Each element of an object of type `dict_items` is a tuple of a key and its associated value. For example, the code

```
for key, val in capitals.items():
    print(val, 'is the capital of', key)
```

prints

```
Paris is the capital of France
Rome is the capital of Italy
Tokyo is the capital of Japan
```

**Finger exercise:** Implement a function that meets the specification

```
def get_min(d):
    """d a dict mapping letters to ints
       returns the value in d with the key that occurs first
       in the
       alphabet. E.g., if d = {x = 11, b = 12}, get_min
       returns 12."""

```

It is often convenient to use tuples as keys. Imagine, for example, using a tuple of the form `(flight_number, day)` to represent airline flights. It would then be easy to use such tuples as keys in a dictionary implementing a mapping from flights to arrival times. A list cannot be used as key, because objects of type `list` are not hashable.

As we have seen, 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.

`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`.  
`d.items()` returns a view of the (key, value) pairs in `d`.  
`d.update(d1)` updates `d` with the (key, value) pairs in `d1`, overwriting existing keys.  
`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`.

[Figure 5-10](#) Some common operations on dicts

---

## 5.8 Dictionary Comprehension

**Dictionary comprehension** is similar to list comprehension. The general form is

```
{key: value for id1, id2 in iterable if test}
```

The key difference (other than the use of set braces rather than square braces) is that it uses two values to create each element of the dictionary, and allows (but does not require) the iterable to return two values at a time. Consider a dictionary mapping some decimal digits to English words:

```
number_to_word = {1: 'one', 2: 'two', 3: 'three', 4: 'four',
10: 'ten'}
```

We can easily use dictionary comprehension to produce a dictionary that maps words to digits with

```
word_to_number = {w: d for d, w in number_to_word.items() }
```

If we decide that we only want single digit numbers in `word_to_number`, we can use the comprehension

```
word_to_number = {w: d for d, w in number_to_word.items() if  
d < 10}
```

Now, let's try something more ambitious. A cipher is an algorithm that maps a plain text (a text that can be easily read by a human) to a crypto text. The simplest ciphers are substitution ciphers that replace each character in the plain text with a unique string. The mapping from the original characters to the string that replaces them is called a key (by analogy with the kind of key used to open a lock, not the kind of key used in Python dictionaries). In Python, dictionaries provide a convenient way to implement mappings that can be used to code and decode text.

A **book cipher** is a cipher for which the key is derived from a book. For example, it might map each character in the plain text to the numeric index of the first occurrence of that character in the book (or on a page of the book). The assumption is that the sender and receiver of the coded message have previously agreed on the book, but an adversary who intercepts the coded message does not know what book was used to code it.

The following function definition uses dictionary comprehension to create a dictionary that can be used for encoding a plain text using a book cipher.

```
gen_code_keys = (lambda book, plain_text:  
    {c: str(book.find(c)) for c in plain_text}))
```

If `plain_text` were “no is no” and `book` started with “Once upon a time, in a house in a land far away,” the call `gen_code_keys(book, plain_text)` would return

```
{'n': '1', 'o': '7', ' ': '4', 'i': '13', 's': '26'}
```

Notice, by the way, that o gets mapped to seven rather than zero because o and O are different characters. If `book` were the text of *Don*

*Quixote*,<sup>38</sup> the call `gen_code_keys(book, plain_text)` would return

```
{'n': '1', 'o': '13', ' ': '2', 'i': '6', 's': '57'}
```

Now that we have our coding dictionary, we can use list comprehension to define a function that uses it to encrypt a plain text

```
encoder = (lambda code_keys, plain_text:  
    ''.join(['*' + code_keys[c] for c in plain_text])[1:])
```

Since characters in the plain text might be replaced by multiple characters in the cipher text, we use `*` to separate characters in the cipher text. The `.join` operator is used to turn the list of strings into a single string.

The function `encrypt` uses `gen_code_keys` and `encoder` to encrypt a plain text

```
encrypt = (lambda book, plain_text:  
    encoder(gen_code_keys(book, plain_text), plain_text))
```

The call `encrypt(Don_Quixote, 'no is no')` returns

```
1*13*2*6*57*2*1*13
```

Before we can decode the cipher text, we need to build a decoding dictionary. The easy thing to do would be to invert the coding dictionary, but that would be cheating. The whole point of a book cipher is that the sender sends an encrypted message, but not any information about the keys. The only thing the receiver needs to decode the message is access to the book that the encoder used. The following function definition uses dictionary comprehension to build a decoding key from the book and the coded message.

```
gen_decode_keys = (lambda book, cipher_text:  
    {s: book[int(s)] for s in cipher_text.split('*'))})
```

The call `gen_decode_keys(Don_Quixote, '1*13*2*6*57*2*1*13')` would produce the decrypting key

```
{'1': 'n', '13': 'o', '2': ' ', '6': 'i', '57': 's'}
```

If a character occurs in the plain text but not in the book, something bad happens. The `code_keys` dictionary will map each such character to `-1`, and `decode_keys` will map `-1` to whatever the last character in the book happens to be.

**Finger exercise:** Remedy the problem described in the previous paragraph. Hint: a simple way to do this is to create a new book by appending something to the original book.

**Finger exercise:** Using `encoder` and `encrypt` as models, implement the functions `decoder` and `decrypt`. Use them to decrypt the message

```
22*13*33*137*59*11*23*11*1*57*6*13*1*2*6*57*2*6*1*22*13*33*1  
37*59*11*23*11*1*57*6*173*7*11
```

which was encrypted using the opening of *Don Quixote*.

---

## 5.9 Terms Introduced in Chapter

tuple

multiple assignment

iterable object

type iterator

list

mutable type

immutable type

`id` function

object equality

side effect

aliasing

cloning

shallow copy

deep copy  
list comprehension  
higher-order function  
whitespace character  
set  
hashable type  
dictionary  
keys  
value  
view object  
dictionary comprehension  
book cipher

---

- 33 Haldeman was Nixon's chief of staff during the Watergate hearings. Haldeman asserted that this was Nixon's response to a proposal to pay hush money. The taped record suggests otherwise.
- 34 Recall that, for now, you should think of a method simply as a function that is invoked using dot notation.
- 35 The cloning of humans raises a host of technical, ethical, and spiritual conundrums. Fortunately, the cloning of Python objects does not.
- 36 Since Python strings support Unicode, the complete list of whitespace characters is much longer (see [https://en.wikipedia.org/wiki/Whitespace\\_character](https://en.wikipedia.org/wiki/Whitespace_character)).
- 37 Until Python 3.7, the semantics of the language did not define the ordering of the keys.
- 38 The book begins, “In a village of La Mancha, the name of which I have no desire to call to mind, there lived not long since one of

those gentlemen that keep a lance in the lance-rack, an old buckler, a lean hack, and a greyhound for coursing."

# 6

## RECURSION AND GLOBAL VARIABLES

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 an 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 “birthright” citizenship. Roughly speaking, the definition is as follows

- Any child born inside the United States or
- Any child born in wedlock outside the United States, one of whose parents is a citizen of the United States.

The first part is simple; if you are born in the United States, you are a birthright citizen (such as Barack Obama). If you are not born in the U.S., it depends upon whether your parents were U.S. citizens at the time of your birth. And whether your parents were U.S. citizens might depend upon whether their parents were U.S. citizens, 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** (case 2 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. It is the presence of a base case that keeps a recursive definition from being a circular definition.<sup>39</sup>

The world's simplest recursive definition is probably the factorial function (typically written in mathematics using  $!$ ) on natural numbers.<sup>40</sup> The classic **inductive definition** is

$$\begin{aligned} 1! &= 1 \\ (n + 1)! &= (n + 1) * n! \end{aligned}$$

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 6-1](#) contains both an iterative (`fact_iter`) and a recursive (`fact_rec`) implementation of factorial.

```
def fact_iter(n):
    """Assumes n an int > 0
    Returns n!””
    result = 1
    for i in range(1, n+1):
        result *= i
    return result

def fact_rec(n):
    """Assumes n an int > 0
    Returns n!””
    if n == 1:
        return n
    else:
        return n*fact_rec(n - 1)
```

[Figure 6-1](#) Iterative and recursive implementations of factorial

### [Figure 6-1](#) Iterative and recursive implementations of factorial

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

It almost seems like cheating to implement `fact_rec` by calling `fact_rec` from within the body of `fact_rec`. It works for the same reason that the iterative implementation works. We know that the iteration in `fact_iter` 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 `fact_rec` 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 `fact_rec(1)`.

**Finger exercise:** The harmonic sum of an integer,  $n > 0$ , can be calculated using the formula  $1 + \frac{1}{2} + \dots + \frac{1}{n}$ . Write a recursive function that computes this.

---

## 6.1 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.<sup>41</sup>

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 can 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 (not a property of any known breed of rabbit), 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 6-2](#).

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

[Figure 6-2](#) Growth in population of female rabbits

Notice that for month  $n > 1$ ,  $\text{females}(n) = \text{females}(n-1) + \text{females}(n-2)$ . 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 in month  $n-1$  to get the number of females in month  $n$ .

[Figure 6-2](#) Growth in population of female rabbits

The growth in population is described naturally by the **recurrence**<sup>42</sup>

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

This definition is different from the recursive definition of factorial:

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

[Figure 6-3](#) contains a straightforward implementation of the Fibonacci recurrence,<sup>43</sup> 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 test_fib(n):
    for i in range(n+1):
        print('fib of', i, '=', fib(i))

```

[Figure 6-3](#) Recursive implementation of Fibonacci sequence

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 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 24 rabbits from England to be used as targets in hunts. Some escaped. 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  $120^{\text{th}}$  Fibonacci number.<sup>44</sup>

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 common in nature. For example, for most flowers the number of petals is a Fibonacci number.

**Finger exercise:** When the implementation of `fib` in [Figure 6-3](#) is used to compute `fib(5)`, how many times does it compute the value of `fib(2)` on the way to computing `fib(5)`?

---

## 6.2 Palindromes

Recursion is also useful for many problems that do not involve numbers. [Figure 6-4](#) contains a function, `is_palindrome`, that checks whether a string reads the same way backwards and forwards.

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

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

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

    return is_pal(to_chars(s))
```

[Figure 6-4](#) Palindrome testing

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

The helper function `to_chars` 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.

The helper function `is_pal` 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<sup>45</sup> 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 `is_palindrome` 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 12.) The problem-solving principle is to conquer a hard problem by breaking it into a set of subproblems with the properties

- The subproblems are easier to solve than the original problem.
- Solutions of the subproblems can be combined to solve the original problem.

Divide-and-conquer is an 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 the logical operator `and`. [Figure 6-5](#) contains some code that can be used to visualize how this works.

```

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

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

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

    return is_pal(to_chars(s))

```

[Figure 6-5](#) Code to visualize palindrome testing

## Executing the code

```

print('Try dogGod')
print(is_palindrome('dogGod'))
print('Try doGood')
print(is_palindrome('doGood'))

```

prints

```

Try dogGod
    is_pal called with doggod
    is_pal called with oggo
    is_pal called with gg
    is_pal 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

```

```
is_pal called with dogood
is_pal called with ogoo
is_pal called with go
About to return False for go
About to return False for ogoo
About to return False for dogood
False
```

---

## 6.3 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 11 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. Once in a while, however, global variables come in handy. Consider the code in [Figure 6-6](#).

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

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

[Figure 6-6](#) Using a global variable

In each function, the line of code `global num_fib_calls` tells Python that the name `num_fib_calls` should be defined outside of the function in which the line of code appears. Had we not included the code `global num_fib_calls`, the name `num_fib_calls` would have been local to each of the functions `fib` and `test_fib`, because `num_fib_calls` occurs on the left-hand side of an assignment statement in both `fib` and `test_fib`. The functions `fib` and `test_fib` both have unfettered access to the object referenced by the variable `num_fib_calls`. The function `test_fib` binds `num_fib_calls` to 0 each time it calls `fib`, and `fib` increments the value of `num_fib_calls` each time `fib` is entered.

The call `test_fib(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.
```

We introduce the topic of global variables with some trepidation. Since the 1970s, card-carrying computer scientists have inveighed against them, for good reason. The indiscriminate use of global variables can lead to lots of problems. The key to making programs readable is locality. People read programs 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, their sloppy use can destroy locality. Nevertheless, there are a few times when they are just what is needed. The most common use of global variables is probably to define a **global constant** that will be used in many places. For example, someone writing a physics-related program might want to define the speed of light, C, once, and then use it in multiple functions.

---

## 6.4 Terms Introduced in Chapter

recursion  
base case  
recursive (inductive) case  
inductive definition  
recurrence  
helper functions  
short-circuit evaluation  
divide-and-conquer  
global variable  
global constant

- 
- 39.** A circular definition is a definition that is circular.
  - 40.** The exact definition of “natural” number is subject to debate. Some define it as the positive integers (arguing that zero, like negative numbers, is merely a convenient mathematical abstraction) and others as the nonnegative integers (arguing that while it is impossible to have -5 apples it is certainly possible to have no apples). That's why we were explicit about the possible values of  $n$  in the docstrings in Figure 6-1.
  - 41.** 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 Acharya Pingala.

- 42 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.
- 43 While obviously correct, this is a terribly inefficient implementation of the Fibonacci function. There is a simple iterative implementation that is much better.
- 44 The damage done by the descendants of those 24 rabbits has been estimated to be \$600 million per year, and they are in the process of eating many native plants into extinction.
- 45 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**.

# 7

## MODULES AND FILES

So far, we have operated under the assumptions that 1) our entire program is stored in one file, 2) our programs do not depend upon previously written code (other than the code implementing Python), and 3) our programs do not access previously gathered data nor do they store their results in a way that allows them to be accessed after the program is finished running.

The first assumption 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. In Section 7.1, we discuss a mechanism, Python modules, that allow us to easily construct a program from code in multiple files.

The second and third assumptions are reasonable for exercises designed to help people learn to program, but rarely reasonable when writing programs designed to accomplish something useful. In Section 7.2, we show how to take advantage of library modules that are part of the standard Python distribution. We use a couple of these modules in this chapter, and many others later in the book. Section 7.3 provides a brief introduction to reading from and writing data to files.

---

### 7.1 Modules

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 7-1](#).

```
pi = 3.14159

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

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

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

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

[Figure 7-1](#) 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.sphere_surface(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.<sup>46</sup> 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.

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 a console, import a module, and then change the contents of that module, the interpreter will still be using the original version of the module. This can lead to puzzling behavior when debugging. When in doubt, start a new shell.

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
```

Many Python programmers frown upon using this kind of “wild card” `import`. They believe that it makes code more difficult to read because it is no longer obvious where a name (for example `pi` in the above code) is defined.

A commonly used variant of the `import` statement is

```
import module_name as new_name
```

This instructs the interpreter to import the module named `module_name`, but rename it to `new_name`. This is useful if

*module\_name* is already being used for something else in the importing program. The most common reason programmers use this form is to provide an abbreviation for a long name.

---

## 7.2 Using Predefined Packages

Lots of useful module packages come as part of the **standard Python library**; we will use a number of them later in this book. Moreover, most Python distributions come with packages beyond those in the standard library. The Anaconda distribution for Python 3.8 comes with over 600 packages! We will use a few of these later in this book.

In this section, we introduce two standard packages, `math` and `calendar`, and give a few simple examples of their use. By the way, these packages, like all of the standard modules, use Python mechanisms that we have not yet covered (e.g., exceptions, which are covered in Chapter 9).

In previous chapters, we presented various ways to approximate logarithms. But we did not show you the easiest way. The easiest way is to simply import the module `math`. For example, to print the log of  $x$  base 2, all you need to write is

```
import math  
print(math.log(x, 2))
```

In addition to containing approximately 50 useful mathematical functions, the `math` module contains several useful floating-point constants, e.g., `math.pi` and `math.inf` (positive infinity).

The standard library modules designed to support mathematical programming represent a minority of the modules in the standard library.

Imagine, for example, that you wanted to print a textual representation of the days of the week of March 1949, something akin to the picture on the right. You could go online and find what the calendar looked like that month and year. Then, with sufficient patience and lots of trial and error, you might manage to write a print statement that would get the job done. Alternatively, you could simply write

March 1949						
Mo	Tu	We	Th	Fr	Sa	Su
1	2	3	4	5	6	
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			

```
import calendar as cal
cal_english = cal.TextCalendar()
print(cal_english.formatmonth(1949, 3))
```

Or, if you preferred to see the calendar in French, Polish, and Danish, you could write

```
print(cal.LocaleTextCalendar(locale='fr_FR').formatmonth(204
9, 3))
print(cal.LocaleTextCalendar(locale='pl_PL').formatmonth(204
9, 3))
print(cal.LocaleTextCalendar(locale='da_dk').formatmonth(204
9, 3))
```

which would produce

mars 2049						
Lu	Ma	Me	Je	Ve	Sa	Di
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31				

marca 2049						
po	wt	śr	cz	pt	so	nd
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31				

Marts 1949						
Ma	Ti	On	To	Fr	Lø	Sø
1	2	3	4	5	6	
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			

Suppose you wanted to know on what day of the week Christmas will fall in 2033. The line

```
print(cal.day_name[cal.weekday(2033, 12, 25)])
```

will answer the question. The invocation of `cal.weekday` will return an integer representing the day of the week,<sup>47</sup> which is then used to index into `cal.day_name`—a list of the days of the week in English.

Now, suppose you wanted to know on what day American Thanksgiving fell in 2011. The day of the week is easy, because American Thanksgiving is always on the fourth Thursday of

November.<sup>48</sup> Finding the actual date is slightly more complex. First, we use `cal.monthcalendar` to get a list representing the weeks of the month. Each element of the list contains seven integers, representing the day of the month. If the day does not occur in that month, the first element of the list for the week will be `0`. For example, if a month with 31 days starts on a Tuesday, the first element of the list will be the list `[0, 1, 2, 3, 4, 5, 6]` and the last element of the list will be `[30, 31, 0, 0, 0, 0, 0]`.

We use the list returned by `calendar.monthcalendar` to check to see if there is a Thursday in the first week. If so, the fourth Thursday is in the fourth week of the month (which is at index 3); otherwise it is in the fifth week.

```
def find_thanksgiving(year):
    month = cal.monthcalendar(year, 11)
    if month[0][cal.THURSDAY] != 0:
        thanksgiving = month[3][cal.THURSDAY]
    else:
        thanksgiving = month[4][cal.THURSDAY]
    return thanksgiving
print('In 2011', 'U.S. Thanksgiving was on November',
      find_thanksgiving(2011))
```

**Finger exercise:** Write a function that meets the specification

```
def shopping_days(year):
    """year a number >= 1941
       returns the number of days between U.S. Thanksgiving
       and
       Christmas in year"""

```

**Finger exercise:** Since 1958, Canadian Thanksgiving has occurred on the second Monday in October. Write a function that takes a year ( $>1957$ ) as a parameter, and returns the number of days between Canadian Thanksgiving and Christmas.

By convention, Python programmers usually

1. Import one module per line.
2. Place all imports at the start of a program.
3. Import standard modules first, followed by third-party modules (e.g., the modules provided through Anaconda), and

finally application-specific modules.

Occasionally, placing all imports at the start of a program leads to a problem. An import statement is an executable line of code, and the Python interpreter executes it when it is encountered. Some modules contain code that gets executed when the module is imported. Typically, this code initializes some objects used by the module. Since some of this code might access shared resources (e.g., the file system on your computer), where in a program the import is executed might matter. The good news is that this is unlikely to be a problem for the modules you are likely to use.

---

## 7.3 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 macOS) 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

```
name_handle = 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. (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**.) Finally, the code **closes** the file. Remember to close a file when the program is finished using it. Otherwise there is a risk that some or all of the writes may not be saved.

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

You can ensure that you don't forget to close a file by opening it using a `with` statement. Code of the form

```
with open(file_name) as name_handle:  
    code_block
```

opens a file, binds a local name to it that can be used in the `code_block`, and then closes the file when `code_block` is exited.

The following code opens a file for **reading** (using the argument '`r`'), and prints 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.

```
with open('kids', 'r') as name_handle:  
    for line in name_handle:  
        print(line)
```

If we type 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 the extra line by writing `print(line[:-1])`.

The code

```
name_handle = open('kids', 'w')  
name_handle.write('Michael')  
name_handle.write('Mark')  
name_handle.close()  
name_handle = open('kids', 'r')  
for line in name_handle:  
    print(line)
```

will print the single line MichaelMark.

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

```
name_handle = open('kids', 'a')  
name_handle = open('kids', 'a')  
name_handle.write('David')
```

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

it will print the line MichaelMarkDavidAndrea.

**Finger exercise:** Write a program that first stores the first ten numbers in the Fibonacci sequence to a file named `fib_file`. Each number should be on a separate line in the file. The program should then read the numbers from the file and print them.

Some of the common operations on files are summarized in [Figure 7-2](#).

`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 7-2](#) Common functions for accessing files

---

## 7.4 Terms Introduced in Chapter

module  
import statement  
fully qualified names  
standard Python library  
files  
file handle  
writing to and reading  
from files  
newline character  
opening and closing files  
with statement  
appending to files

---

- 46 Superficially, this may seem unrelated to the use of dot notation in method invocation. However, as we will see in Chapter 10, there is a deep connection.
- 47 John Conway's "Doomsday rule" provides an interesting algorithm for computing the day of the week for a date. It relies on the fact that each year the dates 4/4, 6/6, 8/8, 10/10, 12/12, and the last day of February occur on the same day of the week as each other. The algorithm is sufficiently simple that some people can execute it in their head.
- 48 It wasn't always the fourth Thursday of November. Abraham Lincoln signed a proclamation declaring that the last Thursday in November should be a national Thanksgiving Day. His successors continued the tradition until 1939, when Franklin Roosevelt declared that the holiday should be celebrated on the penultimate

Thursday of the month (to allow more time for shopping between Thanksgiving and Christmas). In 1941, Congress passed a law establishing the current date. It was not the most consequential thing it did that year.

# 8

## TESTING AND DEBUGGING

We hate to bring this up, but Dr. Pangloss<sup>49</sup> 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. Sometimes cheaters do win (see Houston Astros). 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 complete that next problem set on time, this chapter provides a highly condensed discussion of the topic. While all of the programming examples are in Python, the general principles apply to getting any complex system to work.

**Testing** is the process of running a program to try and ascertain whether 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 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.

---

## 8.1 Testing

The purpose of testing 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!”<sup>50</sup> 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 is_smaller(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, `is_smaller(x, y)`. The set of

possible inputs is all pairwise combinations of integers. One way to partition this set is into these nine subsets:

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

If we tested the implementation on at least one value from each of these subsets, we would have a good chance (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 on exploring paths through the code fall into a class called **glass-box** (or **white-box**) **testing**. Heuristics based on exploring paths through the specification fall into a class called **black-box testing**.

### 8.1.1 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. They too develop black-box test suites.

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. Looking at an argument of type list often means looking at the empty list, a list with exactly one element, a list with immutable elements, a list with mutable elements, 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  $\text{epsilon}$  similar to those in [Figure 8-1](#).

<b>x</b>	<b>epsilon</b>
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 8-1](#) Testing boundary conditions

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.

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 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.

### 8.1.2 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 is_prime(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 `is_prime(2)`, and would therefore not discover that the function call `is_prime(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, including many in this book, 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, a few rules of thumb are usually worth following:

- Exercise both branches of all `if` statements.
- Make sure that each `except` clause (see Chapter 9) 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.
  - 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.

### 8.1.3 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 groups of units function properly when combined. Finally, **functional testing** is used to check if the program as a whole behaves as intended. In practice, testers cycle through these phases, since failures during integration or functional testing lead to making changes to individual units.

Functional testing is almost always the most challenging phase. The intended behavior of an entire program is 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 the subsystem that counts the number of characters in a document. Problems of scale can also make functional testing difficult. It is not unusual for functional 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 an SQA 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 and functional testing.

In industry, the testing process is often highly automated. Testers<sup>51</sup> 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 units) to test.
- Invoke the program (or units) to test with a predefined or automatically generated sequence of inputs.
- Save the results of these invocations.
- Check the acceptability of test results.
- 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.
- Return values consistent with the specification.

Building adequate stubs is often a challenge. If the unit the stub replaces 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 set of arguments accepted by the stub, and create a table that contains the values to return 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, or maybe many things, 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.

---

## 8.2 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 8-2](#) is of a page, from September 9, 1947, in a laboratory book from the group working on the Mark II Aiken Relay Calculator at Harvard University. Notice the moth taped to the page and the phrase “First actual case of bug being found” below it.

92

9/9

0800 Auton started  
 1900 stopped - auton ✓  
 13' UC (032) MP-MC { 1.2700 9.032 847 025  
 2.13047645 9.037 846 995 const  
 033 PRO 2 2.13047645

const  
 Relays 6-2 in 033 fault special speed test  
 in relay " 10,000 test.

1100 Started Cosine Tape (Sine check)  
 1525 Started Multi Adder Test.

1545 Relay #70 Panel F  
 (Moth) in relay.

~~1600~~ Auton started.  
 1700 closed down.

First actual case of bug being found.



[Figure 8-2](#) 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,<sup>52</sup> 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.”<sup>53</sup> 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 the bug is overt can depend upon how carefully you examine 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 16, we will look at programs that model situations in which randomness plays a role. In programs of that kind, intermittent bugs are common.

The best kind 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 the time would be far more dangerous than one that makes obvious 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 an undetected fallacious answer for long periods of time. Such programs can, and have, caused a lot of damage.<sup>54</sup> 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. Software in a flight management computer can make the difference between an aircraft remaining airborne or not.<sup>55</sup> 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.

### 8.2.1 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 debugging tools are built into all of the popular Python IDEs. (If you haven't already, give the debugging tool in Spyder a try.) These tools can help. But what's much more important is how you approach the problem. Many experienced programmers don't even bother with debugging tools, relying instead on 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 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 working because I forgot about the possibility of aliasing in multiple places.”

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 loop. If these are always paired, then the hypothesis that a loop is causing nontermination has been refuted. Decide before running the experiment how you would interpret various possible results. All humans are subject to what psychologists call **confirmation bias**—we interpret information in a way that reinforces what we want to believe. If you wait until after you run the experiment to think about what the results should be, you are more likely to fall prey to wishful thinking.

Finally, 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, you can 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.”<sup>56</sup>

### 8.2.2 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 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 you might go about debugging it. Imagine that you wrote the palindrome-checking code in [Figure 8-3](#).

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

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 is_pal(result):
        print('Yes')
    else:
        print('No')
```

[Figure 8-3](#) 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 by entering the 3,116,480 letters in the *Bible*, and your program printed `Yes`. Yet any fool can see that the *Bible* is not a palindrome. Fix it! (Your program, not the *Bible*.)”

You could try and test it on the *Bible*. 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 millions of characters. 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 some easily examined intermediate values 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, or a few units if you are testing a large system.

Looking at `silly`, the halfway point is around the line `if is_pal(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 the value `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 is_pal(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 `is_pal`. Insert the line

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

before the `return` statement. 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 `is_pal` 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.

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 the bug is to replace the first assignment statement in `is_pal` by `temp = x[:]`, which makes a copy of `x`.

The corrected version of `is_pal` is

```

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

```

### 8.2.3 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.”<sup>57</sup> But he never debugged a piece of software. This subsection contains a few pragmatic hints about what to do when the debugging gets tough.

- *Look for the usual suspects.* 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 to test for object equality (e.g., `id(L1) == id(L2)`)?
  - 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?
  - 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 have found it long ago. One practical way to decide 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.”<sup>58</sup>
- *Try to explain the problem to somebody else.* We all develop blind spots. Merely attempting to explain the problem to someone will often lead you to see things you have missed. You can also try to explain why the bug cannot be in certain places.

- *Don't believe everything you read.*<sup>59</sup> 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 than if you had stuck with it, but you will probably spend 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!)

#### 8.2.4 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 pause. 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 take care of the bug in concert with other changes. Suppose, for example, that you have discovered that the bug is the result of accidentally mutating a list. You could circumvent the problem locally, perhaps by making a copy of the list. Alternatively, 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. Nothing is 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 your starting point. 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 a better way

to organize your program or maybe a simpler algorithm that will be easier to implement correctly.

---

### **8.3 Terms Introduced in Chapter**

testing  
debugging  
test suite  
partition of inputs  
glass-box testing  
black-box testing  
path-complete testing  
unit testing  
integration testing  
functional testing  
software quality assurance (SQA)  
test driver  
test stub  
regression testing  
bug  
overt bug  
covert bug  
persistent bug  
intermittent bug  
defensive programming  
debuggers  
confirmation bias

## bisection search

---

49. Dr. Pangloss appears in Voltaire's *Candide*, as a teacher of metaphysico-theologico-cosmo-lonigology. Pangloss' optimistic view of the world is the limit of the central thesis of Gottfried Leibniz's *Essays of Theodicy on the Goodness of God, the Freedom of Man and the Origin of Evil*.
50. "Notes On Structured Programming," Technical University Eindhoven, T.H. Report 70-WSK-03, April 1970.
51. Or, for that matter, those who grade problem sets in very large programming courses.
52. Grace Murray Hopper played a prominent role in the early development of programming languages and language processors. Despite never having been involved in combat, she rose to the rank of Rear Admiral in the U.S. Navy.
53. *Webster's New World College Dictionary*.
54. On August 1, 2012, Knight Capital Group, Inc. deployed a new piece of stock-trading software. Within 45 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." No human could have made that many mistakes that quickly.
55. Inadequate software was a contributing factor in the loss of 346 lives in two commercial airline crashes between October 2018 and March 2019.
56. This quotation is drawn from Rita Mae Brown's *Sudden Death*. However, it has been variously attributed to many other sources—including Albert Einstein.
57. 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."

- 58 Arthur Conan Doyle, “The Sign of the Four.”
59. This suggestion is of general utility, and should be extended to things you see on television or hear on the radio.

# 9

## 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 circumstances. You’ve already seen some exceptions.

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 already have encountered many of these. Among the most common types of exceptions are `TypeError`, `IndexError`, `NameError`, and `ValueError`.

---

## 9.1 Handling Exceptions

Up to now, we have treated exceptions as terminal 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.

Python provides a convenient mechanism, **try-except**, for **catching** and handling exceptions. The general form is

```
try
    code block
except (list of exception names):
    code block
else:
    code block
```

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

```
success_failure_ratio = num_successes/num_failures
print('The success/failure ratio is', success_failure_ratio)
```

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

It is better to write something along the lines of

```

try:
    success_failure_ratio = num_successes/num_failures
    print('The success/failure ratio is',
    success_failure_ratio)
except ZeroDivisionError:
    print('No failures, so the success/failure ratio is
undefined.')

```

Upon entering the `try` block, the interpreter attempts to evaluate the expression `num_successes/num_failures`. If expression evaluation is successful, the program assigns the value of the expression to the variable `success_failure_ratio`, executes the `print` statement at the end of the `try` block, and then proceeds to execute whatever code follows the `try-except` block. 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 following the `try-except` block.

**Finger exercise:** Implement a function that meets the specification below. Use a `try-except` block. Hint: before starting to code, you might want to type something like `1 + 'a'` into the shell to see what kind of exception is raised.

```

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

```

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. Consider the function definition in [Figure 9-1](#).

```
def get_ratios(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('get_ratios called with bad arguments')
    return ratios
```

[Figure 9-1](#) 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 `get_ratios` should respect the assumptions in the specification of `get_ratios`. 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 `get_ratios`. 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. When the code

```
try:  
    print(get_ratios([1, 2, 7, 6], [1, 2, 0, 3]))  
    print(get_ratios([], []))  
    print(get_ratios([1, 2], [3]))  
except ValueError as msg:  
    print(msg)
```

is executed it prints

```
[1.0, 1.0, nan, 2.0]  
[]  
get_ratios called with bad arguments
```

For comparison, [Figure 9-2](#) contains an implementation of the same specification, but without using a `try-except`. The code in [Figure 9-2](#) is longer and more difficult to read than the code in [Figure 9-1](#). It is also less efficient. (The code in [Figure 9-2](#) could be shortened by eliminating the local variables `vect1_elem` and `vect2_elem`, but only at the cost of introducing yet more inefficiency by indexing into the lists repeatedly.)

```
def get_ratios(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('get_ratios called with bad arguments')  
    for index in range(len(vect1)):  
        vect1_elem = vect1[index]  
        vect2_elem = vect2[index]  
        if (type(vect1_elem) not in (int, float))\  
            or (type(vect2_elem) not in (int, float)):  
            raise ValueError('get_ratios called with bad arguments')  
        if vect2_elem == 0:  
            ratios.append(float('NaN')) #NaN = Not a Number  
        else:  
            ratios.append(vect1_elem/vect2_elem)  
    return ratios
```

[Figure 9-2](#) Control flow without a try-except

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 read_int():
    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 read_val(val_type, request_msg, error_msg):
    while True:
        val = input(request_msg + ' ')
        try:
            return(val_type(val)) #convert str to val_type
        except ValueError:
            print(val, error_msg)
```

The function `read_val` 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 = read_val(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 that something troublesome has happened. (And, as we discussed in Chapter 8, overt bugs are much better than covert bugs.) Moreover, it gives someone debugging the program a clear indication of where things went awry.

---

## 9.2 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 is 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 10) 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 find_an_even(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"""

```

Let's look at one more example, [Figure 9-3](#). The function `get_grades` 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 `get_grades` deal with it, but that would have provided less information to the calling code about what went wrong. The code that calls `get_grades` either uses the returned value

to compute another value or handles the exception and prints an informative error message.

```
def get_grades(fname):
    grades = []
    try:
        with open(fname, 'r') as grades_file:
            for line in grades_file:
                try:
                    grades.append(float(line))
                except:
                    raise ValueError('Cannot convert line to float')
    except IOError:
        raise ValueError('get_grades could not open ' + fname)
    return grades

try:
    grades = get_grades('quiz1grades.txt')
    grades.sort()
    median = grades[len(grades)//2]
    print('Median grade is', median)
except ValueError as error_msg:
    print('Whoops.', error_msg)
```

[Figure 9-3](#) Get grades

---

### 9.3 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. They can be used, for example, to confirm that intermediate values have the expected values or that a function returns an acceptable value.

---

## **9.4 Terms Introduced in Chapter**

exceptions

raising an exception

unhandled exception

handled exception

try-except construct

catch (an exception)

polymorphic functions

first-class objects

raise statement

assertions

# 10

## 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 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 about 50 years old, and have been widely accepted and practiced over the last 30 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 object-oriented programming really took off in practice.

We have been implicitly relying on object-oriented programming throughout most of this book. Back in Section 2.2.1 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 `float` and `str` 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.

---

## 10.1 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 programmers 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. Two powerful mechanisms are 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. We 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 we start the programming process by devising types that will be relevant months and even decades later, we have 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.*”<sup>60</sup>

In Python, we implement data abstractions using **classes**. 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

Consider the following tiny (and totally useless) **class definition**.

```
class Toy(object):
    def __init__(self):
        self._elems = []
    def add(self, new_elems):
        """new_elems is a list"""
        self._elems += new_elems
    def size(self):
        return len(self._elems)
```

The first line indicates that `Toy` is a subclass of `object`. For now, ignore what it means to be a subclass. We will get to that shortly.

A class definition creates an object of type `type` and associates with that class object a set of objects called **attributes**. In this example, the three attributes associated with the class are `__init__`, `add`, and `size`. Each is of type `function`. Consequently, the code,

```
print(type(Toy))
print(type(Toy.__init__), type(Toy.add), type(Toy.size))
```

prints

```
<class 'type'>
<class 'function'> <class 'function'> <class 'function'>
```

As we will see, Python has a number of special function names that start and end with two underscores. These are often referred to as **magic methods**.<sup>61</sup> The first of these we will look at is `__init__`. Whenever a class is instantiated, a call is made to the `__init__` function defined in that class. When the line of code

```
s = Toy()
```

is executed, the interpreter will create a new **instance** of type `Toy`, and then call `Toy.__init__` with the newly created object as the actual parameter that is bound to the formal parameter `self`. When invoked, `Toy.__init__` creates the list object `_elems`, which becomes part of the newly created instance of type `Toy`. (The list is created using the by now familiar notation `[]`, which is simply an

abbreviation for `list()`.) The list `_elems` is called a **data attribute** of the instance of `Toy`. The code

```
t1 = Toy()
print(type(t1))
print(type(t1.add))
t2 = Toy()
print(t1 is t2) #test for object identity
```

prints

```
<class '__main__.Toy'>
<class 'method'>
False
```

Notice that `t1.add` is of type `method`, whereas `Toy.add` is of type `function`. Because `t1.add` is a method, we can invoke it (and `t1.size`) using dot notation.

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:

- Class attributes are defined in a class definition; for example `Toy.size` is an attribute of the class `Toy`. When the class is instantiated, e.g., by the statement `t = Toy()`, instance attributes, e.g., `t.size`, are created.
- While `t.size` is initially bound to the `size` function defined in the class `Toy`, that binding can be changed during the course of a computation. For example, you could (but definitely should not!) change the binding by executing `t.size = 3`.
- 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, `_elems` is an instance variable because for each instance of class `Toy`, `_elems` is bound to a different list. So far, we haven't seen a class variable. We will use one in [Figure 10-4](#).

Now, consider the code

```
t1 = Toy()
t2 = Toy()
t1.add([3, 4])
t2.add([4])
print(t1.size() + t2.size())
```

Since each instance of `Toy` is a different object, each instance of type `Toy` will have a different `_elems` attribute. Therefore, the code prints 3.

At first blush, something appears to be inconsistent in this code. It looks as if each method is being called with one argument too few. For example, `add` has two formal parameters, but we appear to be calling it with only one actual parameter. This is an artifact of using dot notation to invoke a method associated with an instance of a class. 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.

Another common convention is to start the name of data attributes with an underscore. As we discuss in detail in Section 10.3, we use the leading `_` to indicate that the attribute is private to the class, i.e., should not be directly accessed outside the class.

Now, let's look at a more interesting example. [Figure 10-1](#) contains a **class definition** that provides a straightforward implementation of a set-of-integers abstraction called `Int_set`. (Given that Python has a built-in type `set`, this implementation is both unnecessary and unnecessarily complicated. However, it is pedagogically useful.)

```

class Int_set(object):
    """An Int_set is a set of integers"""
    #Information about the implementation (not the abstraction):
    #Value of a set is represented by a list of ints, self._vals.
    #Each int in a 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 get_members(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"""
        if self._vals == []:
            return '{}'
        self._vals.sort()
        result = ''
        for e in self._vals:
            result = result + str(e) + ','
        return f'{{{result[:-1]}}}'

```

[Figure 10-1](#) Class Int\_set

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 subclasses (see Section 10.2) of the class, not at programmers who might want to use the abstraction.

As we have seen, methods associated with an instance of a class can be invoked using dot notation. For example, the code,

```
s = Int_set()  
s.insert(3)  
print(s.member(3))
```

creates a new instance of `Int_set`, inserts the integer 3 into that `Int_set`, and then prints `True`.

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
- 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 `Int_set` is that `vals` contains no duplicates. The implementation of `__init__` is responsible for establishing the invariant (which holds for 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. The `__str__` method of a class is invoked when a program converts an instance of that class to a string by calling `str`. Therefore, when the `print` command is used, the `__str__`

function associated with the object to be printed is invoked. For example, the code

```
s = Int_set()
s.insert(3)
s.insert(4)
print(str(s))
print('The value of s is', s)
```

will print

```
{3, 4}
The value of s is {3, 4}
```

(If no `__str__` method were defined, executing `print(s)` would cause something like `<__main__.Int_set object at 0x1663510>` to be printed.)

**Finger exercise:** Add a method satisfying the specification below to the `Int_set` class.

```
def union(self, other):
    """other is an Int_set
       mutates self so that it contains exactly the
       elements in self
       plus the elements in other."""
    pass
```

### 10.1.1 Magic Methods and Hashable Types

One of the design goals for Python was to allow programmers to use classes to define new types that are as easy to use as the built-in types of Python. Using magic methods to provide class-specific definitions of built-in functions such as `str` and `len` plays an important role in achieving this goal.

Magic methods can also be used to provide class-specific definitions for infix operators such as `==` and `+`. The names of the methods available for infix operators are

<code>+: __add__</code>	<code>*: __mul__</code>	<code>/: __truediv__</code>
<code>-: __sub__</code>	<code>//: __floordiv__</code>	<code>%: __mod__</code>
<code>**: __pow__</code>	<code> : __or__</code>	<code>&lt;: __lt__</code>
<code>&lt;&lt;: __lshift__</code>	<code>^: __xor__</code>	<code>&gt;: __gt__</code>

```

>>: __rshsift__ ==: __eq__           <=: __le__
&: __and__      !=: __ne__           >=: __ge__

```

You can associate any implementation you want with these operators. If you wanted, you could implement + as subtraction, < as exponentiation, etc. We recommend, however, that you resist the opportunity to be imaginative, and stick to implementations consistent with the conventional meanings of these operators.

Returning to our toy example, consider the code in [Figure 10-2](#).

```

class Toy(object):
    def __init__(self):
        self._elems = []
    def add(self, new_elems):
        """new_elems is a list"""
        self._elems += new_elems
    def __len__(self):
        return len(self._elems)
    def __add__(self, other):
        new_toy = Toy()
        new_toy._elems = self._elems + other._elems
        return new_toy
    def __eq__(self, other):
        return self._elems == other._elems
    def __str__(self):
        return str(self._elems)
    def __hash__(self):
        return id(self)

t1 = Toy()
t2 = Toy()
t1.add([1, 2])
t2.add([3, 4])
t3 = t1 + t2
print('The value of t3 is', t3)
print('The length of t3 is', len(t3))
d = {t1: 'A', t2: 'B'}
print('The value', d[t1], 'is associated with the key t1 in d.')

```

[Figure 10-2](#) Using magic methods

When the code in [Figure 10-2](#) is run it prints

```
The value of t3 is [1, 2, 3, 4]
The length of t3 is 4
The value A is associated with the key t1 in d.
```

We can use instances of `Toy` as dictionary keys because we defined a `__hash__` function for the class. Had we defined an `__eq__` function and not defined a `__hash__` function, the code would have generated the error message `unhashable type: 'Toy'` when we attempted to create a dictionary using `t1` and `t2` as keys. When a user-defined `__hash__` is provided, it should ensure that the hash value of an object is constant throughout the lifetime of that object.

All instances of user-defined classes that do not explicitly define `__eq__` use object identity for `==` and are hashable. If no `__hash__` method is provided, the hash value of the object is derived from the object's identity (see Section 5.3).

**Finger exercise:** Replace the `union` method you added to `Int_set` by a method that allows clients of `Int_set` to use the `+` operator to denote set union.

### 10.1.2 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, a maturity date, and a price 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 us 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 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.

### 10.1.3 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 data could all be kept in a 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 10-3](#) contains a class that incorporates two common attributes (name and birthday) of humans. It uses the standard Python library module `datetime`, which provides many convenient methods for creating and manipulating dates.

```

class Person(object):

    def __init__(self, name):
        """Assumes name a string. Create a person"""
        self._name = name
        try:
            last_blank = name.rindex(' ')
            self._last_name = name[last_blank+1:]
        except:
            self._last_name = name
        self.birthday = None

    def get_name(self):
        """Returns self's full name"""
        return self._name

    def get_last_name(self):
        """Returns self's last name"""
        return self._last_name

    def set_birthday(self, birthdate):
        """Assumes birthdate is of type datetime.date
           Sets self's birthday to birthdate"""
        self._birthday = birthdate

    def get_age(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):
        """Assume other a Person
           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._last_name == other._last_name:
            return self._name < other._name
        return self._last_name < other._last_name

    def __str__(self):
        """Returns self's name"""
        return self._name

```

[Figure 10-3](#) Class Person

The following code uses `Person` and `datetime`.

```

me = Person('Michael Guttag')
him = Person('Barack Hussein Obama')
her = Person('Madonna')
print(him.get_last_name())
him.set_birthday(datetime.date(1961, 8, 4))
her.set_birthday(datetime.date(1958, 8, 16))
print(him.get_name(), 'is', him.get_age(), '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.

Executing the above code creates three instances of class `Person`. We can access information about these instances using the methods associated with them. For example, `him.get_last_name()` returns '`Obama`'. The expression `him._last_name` 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 `get_birthday` 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` provides a `Person`-specific definition for 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 `p_list` is a list composed

of elements of type `Person`, the call `p_list.sort()` will sort that list using the `__lt__` method defined in class `Person`. Therefore, the code

```
pList = [me, him, her]
for p in pList:
    print(p)
pList.sort()
for p in pList:
    print(p)
```

will print

```
Michael Guttag
Barack Hussein Obama
Madonna
Michael Guttag
Madonna
Barack Hussein Obama
```

---

## 10.2 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 runtime 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 `MIT_person` in [Figure 10-4](#) 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, `MIT_person` 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 `MIT_person` has added the class variable `_next_id_num`, the instance variable

`_id_num`, and the method `get_id_num`.

- **Override**, i.e., replace, attributes of the superclass. For example, `MIT_person` has overridden `__init__` and `__lt__`. When a method has been overridden, the version of the method that is executed is based on the object 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 `MIT_person.__init__` first uses `super().__init__(name)` to invoke the `__init__` function of its super class (`Person`). This initializes the inherited instance variable `self.name`. It then initializes `self._id_num`, which is an instance variable that instances of `MIT_person` have but instances of `Person` do not.

The instance variable `self._id_num` is initialized using a **class variable**, `_next_id_num`, that belongs to the class `MIT_person`, rather than to instances of the class. When an instance of `MIT_person` is created, a new instance of `next_id_num` is not created. This allows `__init__` to ensure that each instance of `MIT_person` has a unique `_id_num`.

```
class MIT_person(Person):

    _next_id_num = 0 #identification number

    def __init__(self, name):
        super().__init__(name)
        self._id_num = MIT_person._next_id_num
        MIT_person._next_id_num += 1

    def get_id_num(self):
        return self._id_num

    def __lt__(self, other):
        return self._id_num < other._id_num
```

[Figure 10-4](#) Class `MIT_person`

Consider the code

```
p1 = MIT_person('Barbara Beaver')
print(str(p1) + '\'s id number is ' + str(p1.get_id_num()))
```

The first line creates a new `MIT_person`. The second line is 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 `MIT_person`. Since there is not, it next checks to see if there is an `__str__` method associated with the immediate superclass, `Person`, of `MIT_person`. There is, so it uses that. When the runtime system attempts to evaluate the expression `p1.get_id_num()`, it first checks to see if there is a `get_id_num` method associated with class `MIT_person`. 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 = MIT_person('Mark Guttag')
p2 = MIT_person('Billy Bob Beaver')
p3 = MIT_person('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 `MIT_person`, 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 `MIT_person`, the interpreter will use the `__lt__` method defined in class `MIT_person` 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 `MIT_person`. This will lead to the exception

```
AttributeError: 'Person' object has no attribute '_id_num'
```

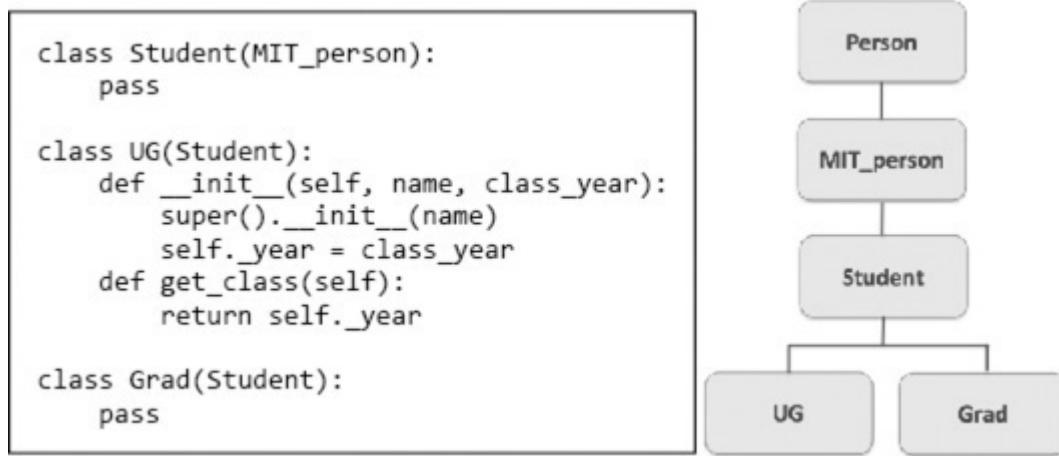
because the object to which `p4` is bound does not have an attribute `_id_num`.

**Finger exercise:** Implement a subclass of `Person` that meets the specification

```
class Politician(Person):
    """ A politician is a person who can belong to a
    political party"""
    def __init__(self, name, party = None):
        """name and party are strings"""
    def get_party(self):
        """returns the party to which self belongs"""
    def might_agree(self, other):
        """returns True if self and other belong to the same
        part
                or at least one of them does not belong to a
        party"""
```

### 10.2.1 Multiple Levels of Inheritance

[Figure 10-5](#) adds another couple of levels of inheritance to the class hierarchy.



[Figure 10-5](#) 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 anyone ever want to create a class with no new attributes?

By introducing the class `Grad`, we gain the ability to create two 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 subtler. Consider going back to `class MIT_person` and adding the method

```
def is_student(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` or a tuple of objects of type `type`. The function returns `True` if and only if the first argument is an instance of the second argument (or, if the second argument is a tuple, an instance of one of the types in the tuple). For example, the value of `isinstance([1,2], list)` is `True`.

Returning to our example, the code

```
print(p5, 'is a student is', p5.is_student())
print(p6, 'is a student is', p6.is_student())
print(p3, 'is a student is', p3.is_student())
```

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 an instance of class `Student` (as well as an instance of `MIT_person` and `Person`).

Since there are only two kinds of students, we could have implemented `is_student` as,

```
def is_student(self):
    return type(self) == Grad or type(self) == UG
```

However, if a new type of student were added later, it would be necessary to go back and edit the code implementing `is_student`. By introducing the intermediate class `Student` and using `isinstance`, we avoid this problem. For example, if we were to add

```
class Transfer_student(Student):
    def __init__(self, name, from_school):
        MIT_person.__init__(self, name)
        self._from_school = from_school
```

```
def get_old_school(self):
    return self._from_school
```

no change needs to be made to `is_student`.

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.

**Finger exercise:** What is the value of the following expression?

```
isinstance('ab', str) == isinstance(str, str)
```

### 10.2.2 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 (hence the phrase substitution principle) 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`.<sup>62</sup>

Conversely, there is no reason to expect that code written to work for `TransferStudent` should work for arbitrary types of `Student`.

---

## 10.3 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 10-6](#) 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.

```
class Grades(object):

    def __init__(self):
        """Create empty grade book"""
        self._students = []
        self._grades = {}
        self._is_sorted = True

    def add_student(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.get_id_num()] = []
        self._is_sorted = False

    def add_grade(self, student, grade):
        """Assumes: grade is a float
           Add grade to the list of grades for student"""
        try:
            self._grades[student.get_id_num()].append(grade)
        except:
            raise ValueError('Student not in mapping')

    def get_grades(self, student):
        """Return a list of grades for student"""
        try:
            return self._grades[student.get_id_num()][:]
        except:
            raise ValueError('Student not in mapping')

    def get_students(self):
        """Return a sorted list of the students in the grade book"""
        if not self._is_sorted:
            self._students.sort()
            self._is_sorted = True
        return self._students[:]
```

[Figure 10-6](#) Class Grades

Notice that `get_grades` returns a copy of the list of grades associated with a student, and `get_students` 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

```
course = Grades()  
course.add_student(Grad('Bernie'))  
all_students = course.get_students()  
all_students.append(Grad('Liz'))
```

If `get_students` returned `self._students`, the last line of code would have the (probably unexpected) side effect of changing the set of students in `course`.

The instance variable `_is_sorted` is used to keep track of whether the list of students has been sorted since the last time a student was added to it. This allows the implementation of `get_students` to avoid sorting an already sorted list.

[Figure 10-7](#) contains a function that uses class `Grades` to produce a grade report for some students taking a course named `six_hundred`.

```

def grade_report(course):
    """Assumes course is of type Grades"""
    report = ''
    for s in course.get_students():
        tot = 0.0
        num_grades = 0
        for g in course.get_grades(s):
            tot += g
            num_grades += 1
        try:
            average = tot/num_grades
            report = f'{report}\n{s}'s mean grade is {average}'
        except ZeroDivisionError:
            report = f'{report}\n{s} has no grades'
    return report

ug1 = UG('Jane Doe', 2021)
ug2 = UG('Pierce Addison', 2041)
ug3 = UG('David Henry', 2003)
g1 = Grad('Billy Buckner')
g2 = Grad('Bucky F. Dent')
six_hundred = Grades()
six_hundred.add_student(ug1)
six_hundred.add_student(ug2)
six_hundred.add_student(g1)
six_hundred.add_student(g2)
for s in six_hundred.get_students():
    six_hundred.add_grade(s, 75)
six_hundred.add_grade(g1, 25)
six_hundred.add_grade(g2, 100)
six_hundred.add_student(ug3)
print(grade_report(six_hundred))

```

[Figure 10-7](#) Generating a grade report

When run, the code in the figure prints

```

Jane Doe's mean grade is 75.0
Pierce Addison'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 = MIT_person('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 `_` (double underscore) but does not end with `__`, that attribute is not visible outside the class. Consider the class in [Figure 10-8](#).

```
class info_hiding(object):
    def __init__(self):
        self.visible = 'Look at me'
        self.__also_visible__ = 'Look at me too'
        self.__invisible = 'Don\'t look at me directly'

    def print_visible(self):
        print(self.visible)

    def print_invisible(self):
        print(self.__invisible)

    def __print_invisible__(self):
        print(self.__invisible)

    def __print_invisible__(self):
        print(self.__invisible)
```

[Figure 10-8](#) Information hiding in classes

## When we run the code

```
test = info_hiding()  
print(test.visible)  
print(test.__also_visible__)  
print(test.__invisible)
```

it prints

```
Look at me  
Look at me too
```

and then raises the exception

```
AttributeError: 'info_hiding' object has no attribute  
'__invisible'
```

The code

```
test = info_hiding()  
test.print_invisible()  
test.__print_invisible__()  
test.__print_invisible()
```

prints

```
Don't look at me directly  
Don't look at me directly
```

and then raises the exception

```
AttributeError: 'info_hiding' object has no attribute  
'__print_invisible'
```

And the code

```
class Sub_class(info_hiding):  
    def new_print_invisible(self):  
        print(self.__invisible)  
test_sub = Sub_class()  
test_sub.new_print_invisible()
```

prints

```
AttributeError: 'Sub_class' object has no attribute  
'__Sub_class_invisible'
```

Notice that when a subclass attempts to use a hidden attribute of its superclass, an `AttributeError` occurs. This can make using information hiding using `_` 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._last_name` rather than `Rafael.get_last_name()`. We discourage this sort of bad behavior by placing a single `_` at the start of the attribute to indicate that we would prefer that clients not access it directly.

We avoid directly accessing data attributes because it is dangerous for client code to rely upon something that is not part of the specification, 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 client code that directly accessed `_last_name` 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.get_age` 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.get_id_num()
```

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.

### 10.3.1 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 or 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 `grade_report` in [Figure 10-7](#). The invocation of `course.get_students` 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 `grade_report` to directly access the instance variable `course.students`, but that would violate information hiding. Fortunately, there is a better solution.

The code in [Figure 10-9](#) replaces the `get_students` function in class `Grades` with a function that uses a kind of statement we have not yet seen: 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 with `for` statements, as in

```
for s in course.get_students():
```

in [Figure 10-7](#).

```
def get_students(self):
    """Return the students in the grade book one at a time
    in alphabetical order"""
    if not self._is_sorted:
        self._students.sort()
        self._is_sorted = True
    for s in self._students:
        yield s
```

[Figure 10-9](#) New version of `get_students`

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.<sup>63</sup>

The version of `get_students` in [Figure 10-9](#) 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.add_student(Grad('Julie'))
book.add_student(Grad('Lisa'))
for s in book.get_students():
    print(s)
```

prints

```
Julie
Lisa
```

Thus the loop in [Figure 10-7](#) that starts with

```
for s in course.get_students():
```

does not have to be altered to take advantage of the version of class `Grades` that contains the new implementation of `get_students`. (Of course, most code that depended upon `get_students` returning a list would no longer work.) The same `for` loop can iterate over the values provided by `get_students` regardless of whether `get_students` 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.

**Finger exercise:** Add to `Grades` a generator that meets the specification

```
def get_students_above(self, grade):
    """Return the students a mean grade > g one at a time"""
```

---

## 10.4 An Extended Example

A collapse in U.S. housing prices helped trigger an international economic meltdown in the fall of 2008. One of the contributing factors was that too many homeowners had taken on mortgages that ended up having unexpected consequences.<sup>64</sup>

In the beginning, mortgages were relatively simple beasts. Buyers borrowed money from a bank and made a fixed-size payment each month for the life of the mortgage, which typically ranged from 15 to 30 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.<sup>65</sup>

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 mortgages:

- A fixed-rate mortgage with no points
- A fixed-rate mortgage with points
- 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 10-10](#) 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 `find_payment` 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, however, 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.

```

def find_payment(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, ann_rate, months):
        """Assumes: loan and ann_rate are floats, months an int
           Creates a new mortgage of size loan, duration months, and
           annual rate ann_rate"""
        self._loan = loan
        self._rate = ann_rate/12
        self._months = months
        self._paid = [0.0]
        self._outstanding = [loan]
        self._payment = find_payment(loan, self._rate, months)
        self._legend = None #description of mortgage

    def make_payment(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 get_total_paid(self):
        """Return the total amount paid so far"""
        return sum(self._paid)

    def __str__(self):
        return self._legend

```

[Figure 10-10](#) Mortgage base class

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, 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 `find_payment`), 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.

The method `make_payment` 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 `make_payment` updates both `self.paid` and `self.outstanding`.

The method `get_total_paid` 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.

[Figure 10-11](#) contains classes implementing three types of mortgages. The classes `Fixed` and `Fixed_with_pts` override `__init__` and inherit the other three methods from `Mortgage`. The class `Two_rate` 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 `make_payment` compares `len(self.paid)` to `self.teaser_months + 1`.)

[Figure 10-11](#) also 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.

We are now, finally, ready to compare different mortgages:

```
compare_mortgages(amt=200000, years=30, fixed_rate=0.035,
                    pts = 2, pts_rate=0.03, var_rate1=0.03,
                    var_rate2=0.05, var_months=60)
```

Notice that we used keyword rather than positional arguments in the invocation of `compare_mortgages`. We did this because `compare_mortgages` has a large number of formal parameters of the same type, and using keyword arguments makes it easier to ensure that we are supplying the intended actual values to each of the formals.

When the code is run, it prints

```
Fixed, 3.5%
Total payments = $323,312
```

Fixed, 3.0%, 2 points  
 Total payments = \$307,555  
 3.0% for 60 months, then 5.0%  
 Total payments = \$362,435

```

class Fixed(Mortgage):
    def __init__(self, loan, r, months):
        Mortgage.__init__(self, loan, r, months)
        self._legend = f'Fixed, {r*100:.1f}%'

class Fixed_with_pts(Mortgage):
    def __init__(self, loan, r, months, pts):
        Mortgage.__init__(self, loan, r, months)
        self._pts = pts
        self._paid = [loan*(pts/100)]
        self._legend = f'Fixed, {r*100:.1f}%, {pts} points'

class Two_rate(Mortgage):
    def __init__(self, loan, r, months, teaser_rate, teaser_months):
        Mortgage.__init__(self, loan, teaser_rate, months)
        self._teaser_months = teaser_months
        self._teaser_rate = teaser_rate
        self._nextRate = r/12
        self._legend = (f'{100*teaser_rate:.1f}% for ' +
                       f'{self._teaser_months} months, then {100*r:.1f}%')

    def make_payment(self):
        if len(self._paid) == self._teaser_months + 1:
            self._rate = self._nextRate
            self._payment = find_payment(self._outstanding[-1],
                                         self._rate,
                                         self._months - self._teaser_months)
        Mortgage.make_payment(self)

    def compare_mortgages(amt, years, fixed_rate, pts, pts_rate,
                           var_rate1, var_rate2, var_months):
        tot_months = years*12
        fixed1 = Fixed(amt, fixed_rate, tot_months)
        fixed2 = Fixed_with_pts(amt, pts_rate, tot_months, pts)
        two_rate = Two_rate(amt, var_rate2, tot_months, var_rate1,
                            var_months)
        morts = [fixed1, fixed2, two_rate]
        for m in range(tot_months):
            for mort in morts:
                mort.make_payment()
        for m in morts:
            print(m)
            print(f' Total payments = ${m.get_total_paid():,.0f}')
  
```

[Figure 10-11](#) Mortgage subclasses

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 13.2.

---

## 10.5 Terms Introduced in Chapter

object-oriented programming

abstract data type

interface

abstraction barrier

decomposition

abstraction

class

class definition

method attribute

class attributes

class instance

attribute reference

\_\_ methods

magic (dunder) methods

data attribute

class variable

instance variable  
class definition  
representation invariant  
inheritance  
subclass  
superclass  
overriding  
isinstance  
substitution principle  
encapsulation  
information hiding  
private  
generator  
abstract class

---

[60](#) “Good heavens, for more than 60 pages we have been using ADTs without knowing it.”

[61](#) They are also referred to as “dunder methods.” The word dunder is derived from “double underscore,” not from the Scottish word for a noise like thunder or the lees produced in the distillation of rum. And it is certainly not intended to imply that only dunderheads use them.

[62](#) This **substitution principle** was nicely described by Barbara Liskov and Jeannette Wing in their 1994 paper, “A behavioral notion of subtyping.”

[63](#) This explanation of generators doesn't give the whole story. To fully understand generators, you need to understand how built-in iterators are implemented in Python, which is not covered in this book.

- 64.** 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.)
- 65** The London Interbank Offered Rate (LIBOR) is probably the most commonly used index.

# 11

## 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.

---

### 11.1 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
- 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 microseconds, 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.<sup>66</sup> 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 can depend not only upon the sizes of the inputs but also upon their values. Consider, for example, the linear search algorithm implemented by

```

def linear_search(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 `linear_search(L, 3)`. If the first element in `L` is `3`, `linear_search` will return `True` almost immediately. On the other hand, if `3` is not in `L`, `linear_search` 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. That is, the **best-case** running time is the minimum running time over all the possible inputs of a given size. For `linear_search`, 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 `linear_search`, 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 when 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 positive int
       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 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 11-1](#) and [Figure 11-2](#).

```

def square_root_exhaustive(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 11-1](#) Using exhaustive enumeration to approximate square root

```

def square_root_bi(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 11-2](#) Using bisection search to approximate square root

We saw that exhaustive enumeration was so slow as to be impractical for many combinations of values for `x` and `epsilon`. For example, evaluating `square_root_exhaustive(100, 0.0001)` requires roughly one billion iterations of the `while` loop. In contrast, evaluating `square_root_bi(100, 0.0001)` takes roughly 20 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. That is, the multiplicative constants are irrelevant.

---

## 11.2 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 11-3](#).

```
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 11-3](#) Asymptotic complexity

If we assume 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.<sup>67</sup> 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.

Many computer scientists will abuse Big O notation by making statements like, “the complexity of `f(x)` is  $O(x^2)$ .” By this they mean that in the worst case `f` will take no more than  $O(x^2)$  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 not preclude the worst-case running time of  $f$  from being considerably less than  $O(x^2)$ . To avoid this kind of confusion we will use **Big Theta** ( $\theta$ ) when we are describing something that is both an upper and a **lower bound** on the asymptotic worst-case running time. This is called a **tight bound**.

**Finger exercise:** What is the asymptotic complexity of each of the following functions?

```
def g(L, e):
    """L a list of ints, e is an int"""
    for i in range(100):
        for el in L:
            if el == e:
                return True
    return False

def h(L, e):
    """L a list of ints, e is an int"""
    for i in range(e):
        for el in L:
            if el == e:
                return True
    return False
```

---

### 11.3 Some Important Complexity Classes

Some of the most common instances of Big O (and  $\theta$ ) 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.

### 11.3.1 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.

### 11.3.2 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 12.) 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 example,  $O(\log_2(x)) = O(\log_2(10) * \log_{10}(x))$ . There are lots of interesting functions with logarithmic complexity. Consider

```
def int_to_str(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 `//` (floor division) to divide `i` by `10` before getting

a result of 0. So, the complexity of `int_to_str` is in  $O(\log(i))$ . More precisely, it is order  $\theta(\log(i))$ , because  $\log(i)$  is a tight bound.

What about the complexity of

```
def add_digits(n):
    """Assumes n is a nonnegative int
       Returns the sum of the digits in n"""
    string_rep = int_to_str(n)
    val = 0
    for c in string_rep:
        val += int(c)
    return val
```

The complexity of converting `n` to a string using `int_to_str` is order  $\theta(\log(n))$ , and `int_to_str` returns a string of length  $\log(n)$ . The `for` loop will be executed order  $\theta(\text{len(string\_rep)})$  times, i.e., order  $\theta(\log(n))$  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  $\theta(\log(n)) + \theta(\log(n))$ , which makes it order  $\theta(\log(n))$ .

### 11.3.3 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 add_digits(s):
    """Assumes s is a string of digits
       Returns the sum of the digits in s"""
    val = 0
    for c in string_rep:
        val += int(c)
    return val
```

This function is linear in the length of `s`, i.e.,  $\theta(\text{len}(s))$ .

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 are 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 order  $\theta(x)$ .

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 in  $\Theta(x)$ .

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.

#### 11.3.4 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 order  $\Theta(n \log(n))$ , where  $n$  is the length of the list being sorted. We will look at that algorithm and analyze its complexity in Chapter 12.

### 11.3.5 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 11-4](#), which implements a subset test.

```
def is_subset(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 11-4](#) Implementation of subset test

Each time the inner loop is reached it is executed order  $\theta(\text{len}(L2))$  times. The function `is_subset` will execute the outer loop order  $\theta(\text{len}(L1))$  times, so the inner loop will be reached order  $\theta(\text{len}(L1) * \text{len}(L2))$ .

Now consider the function `intersect` in [Figure 11-5](#). The running time for the part of the code building the list that might contain duplicates is clearly order  $\theta(\text{len}(L1) * \text{len}(L2))$ . 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.

```

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 11-5](#) Implementation of list intersection

Evaluating the expression `e not in result` potentially involves looking at each element in `result`, and is therefore  $\theta(\text{len}(\text{result}))$ ; consequently the second part of the implementation is order  $\theta(\text{len}(\text{tmp}) * \text{len}(\text{result}))$ . 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  $\theta(\text{len}(\text{L1}) * \text{len}(\text{L2}))$ .

### 11.3.6 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 11-6](#).

```

def get_binary_rep(n, num_digits):
    """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) > num_digits:
        raise ValueError('not enough digits')
    for i in range(num_digits - len(result)):
        result = '0' + result
    return result

def gen_powerset(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)):
        bin_str = get_binary_rep(i, len(L))
        subset = []
        for j in range(len(L)):
            if bin_str[j] == '1':
                subset.append(L[j])
        powerset.append(subset)
    return powerset

```

[Figure 11-6](#) Generating the power set

The function `gen_powerset(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  $2^n - 1$ .
- For each of these  $2^n$  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  $['x', 'y', 'z']$  and  $b$  is 101, generate the list  $['x', 'z']$ .

Try running `gen_powerset` on a list containing the first 10 letters of the alphabet. It will finish quite quickly and produce a list with 1024 elements. Next, try running `gen_powerset` on the first 20 letters of the alphabet. It will take more than a bit of time to run, and will return a list with about a million elements. If you try running `gen_powerset` on all 26 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 `gen_powerset` on a list containing all uppercase and lowercase letters. Step 1 of the algorithm generates order  $\theta(2^{\text{len}(L)})$  binary numbers, so the algorithm is exponential in  $\text{len}(L)$ .

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 exact solutions on some instances of the problem. But that is a subject for later chapters.

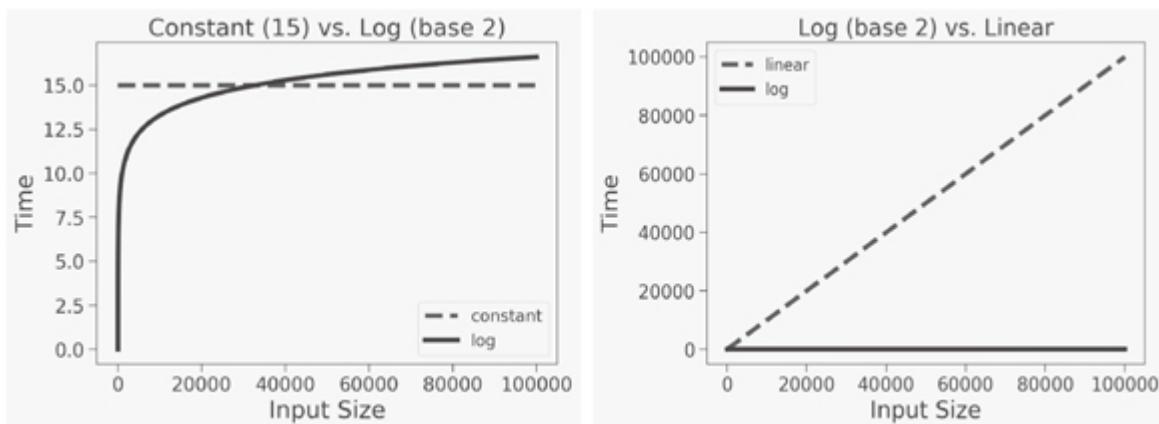
### 11.3.7 Comparisons of Complexity Classes

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 11-7](#) 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 30,000 for the two of them to cross, even for the very small constant of 15. When the size of the input is 100,000, 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.

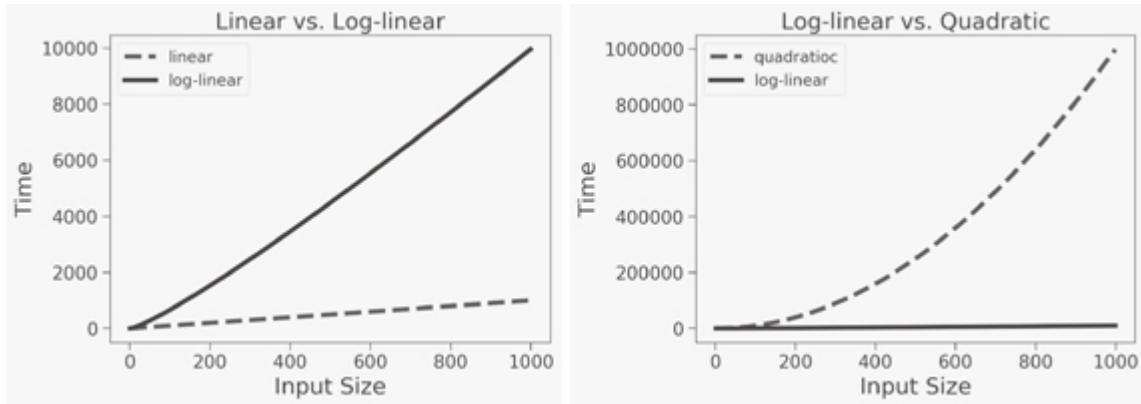
The plot on the right of [Figure 11-7](#) illustrates the dramatic difference between logarithmic algorithms and linear algorithms.

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, much of the time a linear algorithm is acceptably efficient.



[Figure 11-7](#) Constant, logarithmic, and linear growth

The plot on the left in [Figure 11-8](#) shows that there is a significant difference between  $\mathcal{O}(n)$  and  $\mathcal{O}(n \log(n))$ . Given how slowly  $\log(n)$  grows, this may seem surprising, but keep in mind that it is a multiplicative factor. Also keep in mind that in many practical situations,  $\mathcal{O}(n \log(n))$  is fast enough to be useful. On the other hand, as the plot on the right in [Figure 11-8](#) suggests, there are many situations in which a quadratic rate of growth is prohibitive.

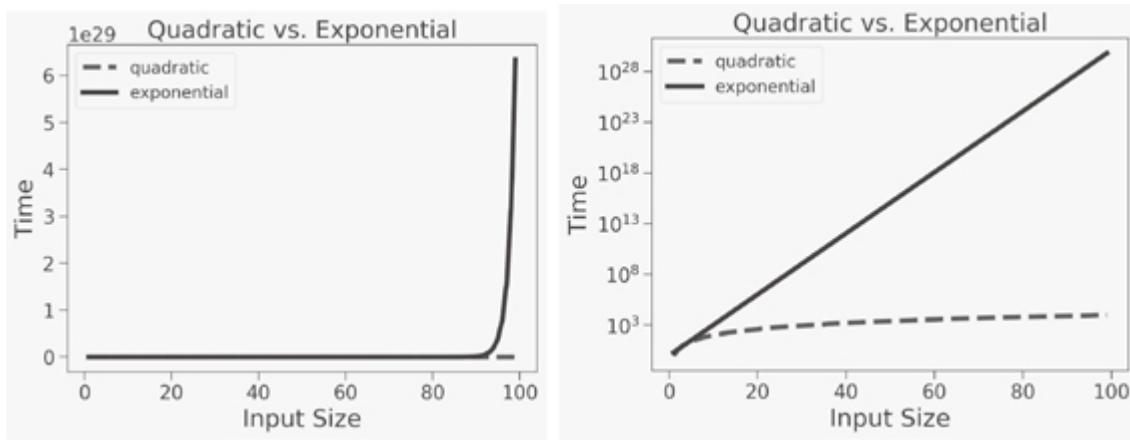


[Figure 11-8](#) Linear, log-linear, and quadratic growth

The plots in [Figure 11-9](#) are about exponential complexity. In the plot on the left of [Figure 11-9](#), the numbers to the left of the y-axis run from 0 to 6. However, the notation  $1e29$  on the top left means that each tick on the y-axis should be multiplied by  $10^{29}$ . So, the plotted y-values range from 0 to roughly  $6.6 \times 10^{29}$ . The curve for quadratic growth is almost invisible in the plot on the left in [Figure 11-9](#). 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 11-9](#) 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 11-9](#) Quadratic and exponential growth

---

## 11.4 Terms Introduced in Chapter

- conceptual complexity
- computational complexity
- random access machine
- computational step
- best-case complexity
- worst-case complexity
- average-case complexity
- expected-case complexity
- upper bound
- asymptotic notation
- Big O notation
- order of growth
- Big Theta notation
- lower bound
- tight bound

constant time  
logarithmic time  
linear time  
log-linear time  
polynomial time  
quadratic time  
exponential time

---

- [66](#) 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.
- [67](#) 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 nineteenth century to denote a related concept.

# 12

## SOME SIMPLE ALGORITHMS AND DATA STRUCTURES

Though we expend a fair number of pages in this book talking about efficiency, the goal is not to make you expert in designing efficient programs. There are many long books (and even some good long books) devoted exclusively to that topic.<sup>68</sup> In Chapter 11, we introduced some of the basic concepts underlying complexity analysis. In this chapter, we use those concepts to look at the complexity of a few classic algorithms. The goal of this chapter is to help you develop some general intuitions about how to approach questions of efficiency. By the time you get through this chapter, you should understand why some programs complete in the blink of an eye, why some need to run overnight, and why some wouldn't complete in your lifetime.

The first algorithms we looked at in this book were based on brute-force exhaustive enumeration. We argued that modern computers are so fast that it is often the case that employing clever algorithms is a waste of time. Writing code that is simple and obviously correct is often the right way to go.

We then looked at some problems (e.g., finding an approximation to the roots of a polynomial) where the search space was too large to make brute force practical. This led us to consider more efficient algorithms such as bisection search and Newton–Raphson. The major point was that the key to efficiency is a good algorithm, not clever coding tricks.

In the sciences (physical, life, and social), programmers often start by quickly coding a simple algorithm to test the plausibility of a hypothesis about a data set, and then run it on a small amount of

data. If this yields encouraging results, the hard work of producing an implementation that can be run (perhaps over and over again) on large data sets begins. Such implementations need to be based on efficient algorithms.

Efficient algorithms are hard to invent. Successful professional computer scientists might invent one algorithm during their whole career—if they are lucky. Most of us never invent a novel algorithm. What we do instead is learn to reduce the most complex aspects of the problems we are faced with to previously solved problems.

More specifically, we

- Develop an understanding of the inherent complexity of the problem.
- Think about how to break that problem up into subproblems.
- Relate those subproblems to other problems for which efficient algorithms already exist.

This chapter contains a few examples intended to give you some intuition about algorithm design. Many other algorithms appear elsewhere in the book.

Keep in mind that the most efficient algorithm is not always the algorithm of choice. A program that does everything in the most efficient possible way is often needlessly difficult to understand. It is often a good strategy to start by solving the problem at hand in the most straightforward manner possible, instrument it to find any computational bottlenecks, and then look for ways to improve the computational complexity of those parts of the program contributing to the bottlenecks.

---

## 12.1 Search Algorithms

A **search algorithm** is a method for finding an item or group of items with specific properties within a collection of items. We refer to the collection of items as a **search space**. The search space might be something concrete, such as a set of electronic medical records, or something abstract, such as the set of all integers. A large number of

problems that occur in practice can be formulated as search problems.

Many of the algorithms presented earlier in this book can be viewed as search algorithms. In Chapter 3, we formulated finding an approximation to the roots of a polynomial as a search problem and looked at three algorithms—exhaustive enumeration, bisection search, and Newton–Raphson—for searching the space of possible answers.

In this section, we will examine two algorithms for searching a list. Each meets the specification

```
def search(L, e):
    """Assumes L is a list.
    Returns True if e is in L and False otherwise"""

```

The astute reader might wonder if this is not semantically equivalent to the Python expression `e in L`. The answer is yes, it is. And if you are unconcerned about the efficiency of discovering whether `e` is in `L`, you should simply write that expression.

### 12.1.1 Linear Search and Using Indirection to Access Elements

Python uses the following algorithm to determine if an element is in a list:

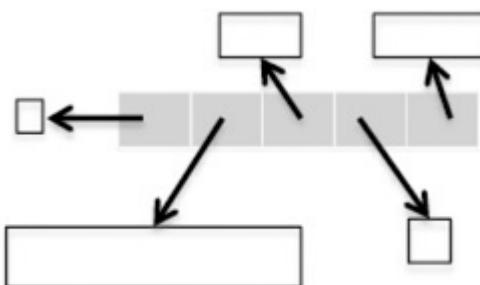
```
for i in range(len(L)):
    if L[i] == e:
        return True
return False
```

If the element `e` is not in the list, the algorithm will perform  $\theta(\text{len}(L))$  tests, i.e., the complexity is at best linear in the length of `L`. Why “at best” linear? It will be linear only if each operation inside the loop can be done in constant time. That raises the question of whether Python retrieves the  $i^{\text{th}}$  element of a list in constant time. Since our model of computation assumes that fetching the contents of an address is a constant-time operation, the question becomes whether we can compute the address of the  $i^{\text{th}}$  element of a list in constant time.

Let's start by considering the simple case where each element of the list is an integer. This implies that each element of the list is the same size, e.g., four units of memory (four 8-bit bytes<sup>69</sup>). Assuming that the elements of the list are stored contiguously, the address in memory of the  $i^{\text{th}}$  element of the list is simply `start + 4*i`, where `start` is the address of the start of the list. Therefore we can assume that Python could compute the address of the  $i^{\text{th}}$  element of a list of integers in constant time.

Of course, we know that Python lists can contain objects of types other than `int`, and that the same list can contain objects of many types and sizes. You might think that this would present a problem, but it does not.

In Python, a list is represented as a length (the number of objects in the list) and a sequence of fixed-size **pointers**<sup>70</sup> to objects. [Figure 12-1](#) illustrates the use of these pointers.



[Figure 12-1](#) Implementing lists

The shaded region represents a list containing four elements. The leftmost shaded box contains a pointer to an integer indicating the length of the list. Each of the other shaded boxes contains a pointer to an object in the list.

If the length field occupies four units of memory, and each pointer (address) occupies four units of memory, the address of the  $i^{\text{th}}$  element of the list is stored at the address `start + 4 + 4*i`. Again, this address can be found in constant time, and then the value stored at that address can be used to access the  $i^{\text{th}}$  element. This access too is a constant-time operation.

This example illustrates one of the most important implementation techniques used in computing: **indirection**.<sup>71</sup> Generally speaking, indirection involves accessing something by first accessing something else that contains a reference to the thing initially sought. This is what happens each time we use a variable to refer to the object to which that variable is bound. When we use a variable to access a list and then a reference stored in that list to access another object, we are going through two levels of indirection.<sup>72</sup>

### 12.1.2 Binary Search and Exploiting Assumptions

Getting back to the problem of implementing `search(L, e)`, is  $\theta(\text{len}(L))$  the best we can do? Yes, if we know nothing about the relationship of the values of the elements in the list and the order in which they are stored. In the worst case, we have to look at each element in `L` to determine whether `L` contains `e`.

But suppose we know something about the order in which elements are stored, e.g., suppose we know that we have a list of integers stored in ascending order. We could change the implementation so that the search stops when it reaches a number larger than the number for which it is searching, as in [Figure 12-2](#).

```
def search(L, e):
    """Assumes L is a list, the elements of which are in
       ascending order.
    Returns True if e is in L and False otherwise"""
    for i in range(len(L)):
        if L[i] == e:
            return True
        if L[i] > e:
            return False
    return False
```

[Figure 12-2](#) Linear search of a sorted list

This would improve the average running time. However, it would not change the worst-case complexity of the algorithm, since in the worst case each element of `L` is examined.

We can, however, get a considerable improvement in the worst-case complexity by using an algorithm, **binary search**, that is similar to the bisection search algorithm used in Chapter 3 to find an approximation to the square root of a floating-point number. There we relied upon the fact that there is an intrinsic total ordering on floating-point numbers. Here we rely on the assumption that the list is ordered.

The idea is simple:

1. Pick an index,  $i$ , that divides the list  $L$  roughly in half.
2. Ask if  $L[i] == e$ .
3. If not, ask whether  $L[i]$  is larger or smaller than  $e$ .
4. Depending upon the answer, search either the left or right half of  $L$  for  $e$ .

Given the structure of this algorithm, it is not surprising that the most straightforward implementation of binary search uses recursion, as shown in [Figure 12-3](#).

The outer function in [Figure 12-3](#), `search(L, e)`, has the same arguments and specification as the function defined in [Figure 12-2](#). The specification says that the implementation may assume that  $L$  is sorted in ascending order. The burden of making sure that this assumption is satisfied lies with the caller of `search`. If the assumption is not satisfied, the implementation has no obligation to behave well. It could work, but it could also crash or return an incorrect answer. Should `search` be modified to check that the assumption is satisfied? This might eliminate a source of errors, but it would defeat the purpose of using binary search, since checking the assumption would itself take  $O(\text{len}(L))$  time.

```

def search(L, e):
    """Assumes L is a list, the elements of which are in
       ascending order.
    Returns True if e is in L and False otherwise"""

    def bin_search(L, e, low, high):
        #Decrements high - low
        if high == low:
            return L[low] == e
        mid = (low + high)//2
        if L[mid] == e:
            return True
        elif L[mid] > e:
            if low == mid: #nothing left to search
                return False
            else:
                return bin_search(L, e, low, mid - 1)
        else:
            return bin_search(L, e, mid + 1, high)

    if len(L) == 0:
        return False
    else:
        return bin_search(L, e, 0, len(L) - 1)

```

[Figure 12-3](#) Recursive binary search

Functions such as `search` are often called **wrapper functions**. The function provides a nice interface for client code, but is essentially a pass-through that does no serious computation. Instead, it calls the helper function `bSearch` with appropriate arguments. This raises the question of why not eliminate `search` and have clients call `bin_search` directly? The reason is that the parameters `low` and `high` have nothing to do with the abstraction of searching a list for an element. They are implementation details that should be hidden from those writing programs that call `search`.

Let us now analyze the complexity of `bin_search`. We showed in the last section that list access takes constant time. Therefore, we can see that excluding the recursive call, each instance of `bSearch` is  $\theta(1)$ . Therefore, the complexity of `bin_search` depends only upon the number of recursive calls.

If this were a book about algorithms, we would now dive into a careful analysis using something called a recurrence relation. But since it isn't, we will take a much less formal approach that starts

with the question “How do we know that the program terminates?” Recall that in Chapter 3 we asked the same question about a `while` loop. We answered the question by providing a decrementing function for the loop. We do the same thing here. In this context, the decrementing function has the properties:

- It maps the values to which the formal parameters are bound to a nonnegative integer.
- When its value is 0, the recursion terminates.
- For each recursive call, the value of the decrementing function is less than the value of the decrementing function on entry to the instance of the function making the call.

The decrementing function for `bin_search` is `high-low`. The `if` statement in `search` ensures that the value of this decrementing function is at least 0 the first time `bSearch` is called (decrementing function property 1).

When `bin_search` is entered, if `high-low` is exactly 0, the function makes no recursive call—simply returning the value `L[low] == e` (satisfying decrementing function property 2).

The function `bin_search` contains two recursive calls. One call uses arguments that cover all the elements to the left of `mid`, and the other call uses arguments that cover all the elements to the right of `mid`. In either case, the value of `high-low` is cut in half (satisfying decrementing function property 3).

We now understand why the recursion terminates. The next question is how many times can the value of `high-low` be cut in half before `high-low == 0`? Recall that  $\log_y(x)$  is the number of times that  $y$  has to be multiplied by itself to reach  $x$ . Conversely, if  $x$  is divided by  $y$   $\log_y(x)$  times, the result is 1. This implies that `high-low` can be cut in half using floor division at most  $\log_2(\text{high-low})$  times before it reaches 0.

Finally, we can answer the question, what is the algorithmic complexity of binary search? Since when `search` calls `bSearch` the value of `high-low` is equal to `len(L)-1`, the complexity of `search` is  $\theta(\log(\text{len}(L)))$ .<sup>73</sup>

**Finger exercise:** Why does the code use `mid+1` rather than `mid` in the second recursive call?

---

## 12.2 Sorting Algorithms

We have just seen that if we happen to know a list is sorted, we can exploit that information to greatly reduce the time needed to search a list. Does this mean that when asked to search a list we should first sort it and then perform the search?

Let  $\theta(\text{sortComplexity}(L))$  be a tight bound on the complexity of sorting a list. Since we know that we can search an unsorted list in  $\theta(\text{len}(L))$  time, the question of whether we should first sort and then search boils down to the question, is `sortComplexity(L) + log(len(L))` less than `len(L)`? The answer, sadly, is no. It is impossible sort a list without looking at each element in the list at least once, so it is not possible to sort a list in sub-linear time.

Does this mean that binary search is an intellectual curiosity of no practical import? Happily, no. Suppose that we expect to search the same list many times. It might well make sense to pay the overhead of sorting the list once, and then **amortize** the cost of the sort over many searches. If we expect to search the list  $k$  times, the relevant question becomes, is `sortComplexity(L) + k*log(len(L))` less than  $k*\text{len}(L)$ ?

As  $k$  becomes large, the time required to sort the list becomes increasingly irrelevant. How big  $k$  needs to be depends upon how long it takes to sort a list. If, for example, sorting were exponential in the size of the list,  $k$  would have to be quite large.

Fortunately, sorting can be done rather efficiently. For example, the standard implementation of sorting in most Python implementations runs in roughly  $O(n * \log(n))$  time, where  $n$  is the length of the list. In practice, you will rarely need to implement your own sort function. In most cases, the right thing to do is to use either Python's built-in `sort` method (`L.sort()` sorts the list `L`) or its built-in function `sorted` (`sorted(L)` returns a list with the same elements as `L`, but does not mutate `L`). We present sorting algorithms here primarily to provide some practice in thinking about algorithm design and complexity analysis.

We begin with a simple but inefficient algorithm, **selection sort**. Selection sort, [Figure 12-4](#), works by maintaining the **loop invariant** that, given a partitioning of the list into a prefix ( $L[0:i]$ ) and a suffix ( $L[i+1:len(L)]$ ), the prefix is sorted and no element in the prefix is larger than the smallest element in the suffix.

We use induction to reason about loop invariants.

- Base case: At the start of the first iteration, the prefix is empty, i.e., the suffix is the entire list. Therefore, the invariant is (trivially) true.
- Induction step: At each step of the algorithm, we move one element from the suffix to the prefix. We do this by appending a minimum element of the suffix to the end of the prefix. Because the invariant held before we moved the element, we know that after we append the element the prefix is still sorted. We also know that since we removed the smallest element in the suffix, no element in the prefix is larger than the smallest element in the suffix.
- Termination: When the loop is exited, the prefix includes the entire list, and the suffix is empty. Therefore, the entire list is now sorted in ascending order.

```
def sel_sort(L):
    """Assumes that L is a list of elements that can be
       compared using >.
       Sorts L in ascending order"""
    suffix_start = 0
    while suffix_start != len(L):
        #look at each element in suffix
        for i in range(suffix_start, len(L)):
            if L[i] < L[suffix_start]:
                #swap position of elements
                L[suffix_start], L[i] = L[i], L[suffix_start]
        suffix_start += 1
```

[Figure 12-4](#). Selection sort

It's hard to imagine a simpler or more obviously correct sorting algorithm. Unfortunately, it is rather inefficient.<sup>74</sup> The complexity of the inner loop is  $\theta(\text{len}(L))$ . The complexity of the outer loop is also  $\theta(\text{len}(L))$ . So, the complexity of the entire function is  $\theta(\text{len}(L)^2)$ . I.e., it is quadratic in the length of  $L$ .<sup>75</sup>

### 12.2.1 Merge Sort

Fortunately, we can do a lot better than quadratic time using a **divide-and-conquer algorithm**. The basic idea is to combine solutions of simpler instances of the original problem. In general, a divide-and-conquer algorithm is characterized by

- A threshold input size, below which the problem is not subdivided
- The size and number of sub-instances into which an instance is split
- The algorithm used to combine sub-solutions.

The threshold is sometimes called the **recursive base**. For item 2, it is usual to consider the ratio of initial problem size to the sub-instance size. In most of the examples we've seen so far, the ratio was 2.

**Merge sort** is a prototypical divide-and-conquer algorithm. It was invented in 1945, by John von Neumann, and is still widely used. Like many divide-and-conquer algorithms it is most easily described recursively:

1. If the list is of length 0 or 1, it is already sorted.
2. If the list has more than one element, split the list into two lists, and use merge sort to sort each of them.
3. Merge the results.

The key observation made by von Neumann is that two sorted lists can be efficiently merged into a single sorted list. The idea is to look at the first element of each list and move the smaller of the two to the end of the result list. When one of the lists is empty, all that

remains is to copy the remaining items from the other list. Consider, for example, merging the two lists  $L_1 = [1, 5, 12, 18, 19, 20]$  and  $L_2 = [2, 3, 4, 17]$ :

<b>Remaining in <math>L_1</math></b>	<b>Remaining in <math>L_2</math></b>	<b>Result</b>
[1, 5, 12, 18, 19, 20]	[2, 3, 4, 17]	[]
[5, 12, 18, 19, 20]	[2, 3, 4, 17]	[1]
[5, 12, 18, 19, 20]	[3, 4, 17]	[1, 2]
[5, 12, 18, 19, 20]	[4, 17]	[1, 2, 3]
[5, 12, 18, 19, 20]	[17]	[1, 2, 3, 4]
[12, 18, 19, 20]	[17]	[1, 2, 3, 4, 5]
[18, 19, 20]	[17]	[1, 2, 3, 4, 5, 12]
[18, 19, 20]	[]	[1, 2, 3, 4, 5, 12, 17]
[]	[]	[1, 2, 3, 4, 5, 12, 17, 18, 19, 20]

What is the complexity of the merge process? It involves two constant-time operations, comparing the values of elements and copying elements from one list to another. The number of comparisons is order  $\theta(\text{len}(L))$ , where  $L$  is the longer of the two lists. The number of copy operations is order  $\theta(\text{len}(L_1) + \text{len}(L_2))$ , because each element is copied exactly once. (The time to copy an element depends on the size of the element. However, this does not affect the order of the growth of sort as a function of the number of elements in the list.) Therefore, merging two sorted lists is linear in the length of the lists.

[Figure 12-5](#) contains an implementation of the merge sort algorithm.

```

def merge(left, right, compare):
    """Assumes left and right are sorted lists and
       compare defines an ordering on the elements.
       Returns a new sorted (by compare) list containing the
       same elements as (left + right) would contain."""

    result = []
    i, j = 0, 0
    while i < len(left) and j < len(right):
        if compare(left[i], right[j]):
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    while (i < len(left)):
        result.append(left[i])
        i += 1
    while (j < len(right)):
        result.append(right[j])
        j += 1
    return result

def merge_sort(L, compare = lambda x, y: x < y):
    """Assumes L is a list, compare defines an ordering
       on elements of L
       Returns a new sorted list with the same elements as L"""
    if len(L) < 2:
        return L[:]
    else:
        middle = len(L)//2
        left = merge_sort(L[:middle], compare)
        right = merge_sort(L[middle:], compare)
        return merge(left, right, compare)

```

[Figure 12-5](#) Merge sort

Notice that we have made the comparison operator a parameter of the `merge_sort` function and written a lambda expression to supply a default value. So, for example, the code

```
L = [2,1,4,5,3]
print(merge_sort(L), merge_sort(L, lambda x, y: x > y))
```

prints

```
[1, 2, 3, 4, 5] [5, 4, 3, 2, 1]
```

Let's analyze the complexity of `merge_sort`. We already know that the time complexity of `merge` is order  $\theta(\text{len}(L))$ . At each level of recursion the total number of elements to be merged is  $\text{len}(L)$ . Therefore, the time complexity of `merge_sort` is order  $\theta(\text{len}(L))$  multiplied by the number of levels of recursion. Since `merge_sort` divides the list in half each time, we know that the number of levels of recursion is order  $\theta(\log(\text{len}(L)))$ . Therefore, the time complexity of `merge_sort` is  $\theta(n * \log(n))$ , where  $n$  is  $\text{len}(L)$ .<sup>76</sup>

This is a lot better than selection sort's  $\theta(\text{len}(L)^2)$ . For example, if  $L$  has 10,000 elements,  $\text{len}(L)^2$  is 100 million but  $\text{len}(L) * \log_2(\text{len}(L))$  is about 130,000.

This improvement in time complexity comes with a price. Selection sort is an example of an **in-place** sorting algorithm. Because it works by swapping the place of elements within the list, it uses only a constant amount of extra storage (one element in our implementation). In contrast, the merge sort algorithm involves making copies of the list. This means that its space complexity is order  $\theta(\text{len}(L))$ . This can be an issue for large lists.<sup>77</sup>

Suppose we want to sort a list of names written as first name followed by last name, e.g., the list `['Chris Terman', 'Tom Brady', 'Eric Grimson', 'Gisele Bundchen']`. [Figure 12-6](#) defines two ordering functions, and then uses these to sort a list in two different ways. Each function uses the `split` method of type `str`.

```

def last_name_first_name(name1, name2):
    arg1 = name1.split(' ')
    arg2 = name2.split(' ')
    if arg1[1] != arg2[1]:
        return arg1[1] < arg2[1]
    else: #last names the same, sort by first name
        return arg1[0] < arg2[0]
def first_name_last_name(name1, name2):
    arg1 = name1.split(' ')
    arg2 = name2.split(' ')
    if arg1[0] != arg2[0]:
        return arg1[0] < arg2[0]
    else: #first names the same, sort by last name
        return arg1[1] < arg2[1]

L = ['Tom Brady', 'Eric Grimson', 'Gisele Bundchen']
newL = merge_sort(L, last_name_first_name)
print('Sorted by last name =', newL)
newL = merge_sort(L, first_name_last_name)
print('Sorted by first name =', newL)

```

[Figure 12-6](#) Sorting a list of names

When the code in [Figure 12-6](#) is run, it prints

```

Sorted by last name = ['Tom Brady', 'Gisele Bundchen', 'Eric
Grimson']
Sorted by first name = ['Eric Grimson', 'Gisele Bundchen',
'Tom Brady']

```

**Finger exercise:** Use `merge_sort` to sort a list to tuples of integers. The sorting order should be determined by the sum of the integers in the tuple. For example, `(5, 2)` should precede `(1, 8)` and follow `(1, 2, 3)`.

### 12.2.2 Sorting in Python

The sorting algorithm used in most Python implementations is called **timsort**.<sup>78</sup> The key idea is to take advantage of the fact that in a lot of data sets, the data are already partially sorted. Timsort's worst-case performance is the same as merge sort's, but on average it performs considerably better.

As mentioned earlier, the Python method `list.sort` takes a list as its first argument and modifies that list. In contrast, the Python

`function sorted` takes an iterable object (e.g., a list or a view) as its first argument and returns a new sorted list. For example, the code

```
L = [3,5,2]
D = {'a':12, 'c':5, 'b':'dog'}
print(sorted(L))
print(L)
L.sort()
print(L)
print(sorted(D))
D.sort()
```

will print

```
[2, 3, 5]
[3, 5, 2]
[2, 3, 5]
['a', 'b', 'c']
AttributeError: 'dict' object has no attribute 'sort'
```

Notice that when the `sorted` function is applied to a dictionary, it returns a sorted list of the keys of the dictionary. In contrast, when the `sort` method is applied to a dictionary, it causes an exception to be raised since there is no method `dict.sort`.

Both the `list.sort` method and the `sorted` function can have two additional parameters. The `key` parameter plays the same role as `compare` in our implementation of merge sort: it supplies the comparison function to be used. The `reverse` parameter specifies whether the list is to be sorted in ascending or descending order relative to the comparison function. For example, the code

```
L = [[1,2,3], (3,2,1,0), 'abc']
print(sorted(L, key = len, reverse = True))
```

sorts the elements of `L` in reverse order of length and prints

```
[(3, 2, 1, 0), [1, 2, 3], 'abc']
```

Both the `list.sort` method and the `sorted` function provide **stable sorts**. This means that if two elements are equal with respect to the comparison (`len` in this example) used in the sort, their relative ordering in the original list (or other iterable object) is preserved in the final list. (Since no key can occur more than once in

a `dict`, the question of whether `sorted` is stable when applied to a `dict` is moot.)

**Finger exercise:** Is `merge_sort` a stable sort?

---

## 12.3 Hash Tables

If we put merge sort together with binary search, we have a nice way to search lists. We use merge sort to preprocess the list in order  $\theta(n^*\log(n))$  time, and then we use binary search to test whether elements are in the list in order  $\theta(\log(n))$  time. If we search the list  $k$  times, the overall time complexity is order  $\theta(n^*\log(n) + k^*\log(n))$ .

This is good, but we can still ask, is logarithmic the best that we can do for search when we are willing to do some preprocessing?

When we introduced the type `dict` in Chapter 5, we said that dictionaries use a technique called hashing to do the lookup in time that is nearly independent of the size of the dictionary. The basic idea behind a **hash table** is simple. We convert the key to an integer, and then use that integer to index into a list, which can be done in constant time. In principle, values of any type can be easily converted to an integer. After all, we know that the internal representation of each object is a sequence of bits, and any sequence of bits can be viewed as representing an integer. For example, the internal representation of the string '`abc`' is the sequence of bits `011000010110001001100011`, which can be viewed as a representation of the decimal integer  $6,382,179$ . Of course, if we want to use the internal representation of strings as indices into a list, the list is going to have to be pretty darn long.

What about situations where the keys are already integers? Imagine, for the moment, that we are implementing a dictionary all of whose keys are U.S. Social Security numbers, which are nine-digit integers. If we represented the dictionary by a list with  $10^9$  elements and used Social Security numbers to index into the list, we could do lookups in constant time. Of course, if the dictionary contained entries for only ten thousand ( $10^4$ ) people, this would waste quite a lot of space.

Which gets us to the subject of hash functions. A **hash function** maps a large space of inputs (e.g., all natural numbers) to a smaller space of outputs (e.g., the natural numbers between 0 and 5000). Hash functions can be used to convert a large space of keys to a smaller space of integer indices.

Since the space of possible outputs is smaller than the space of possible inputs, a hash function is a **many-to-one mapping**, i.e., multiple different inputs may be mapped to the same output. When two inputs are mapped to the same output, it is called a **collision**—a topic we will return to shortly. A good hash function produces a **uniform distribution**; i.e., every output in the range is equally probable, which minimizes the probability of collisions.

[Figure 12-7](#) uses a simple hash function (recall that `i % j` returns the remainder when the integer `i` is divided by the integer `j`) to implement a dictionary with integers as keys.

The basic idea is to represent an instance of class `Int_dict` by a list of **hash buckets**, where each bucket is a list of key/value pairs implemented as tuples. By making each bucket a list, we handle collisions by storing all of the values that hash to the same bucket in the list.

The hash table works as follows: The instance variable `buckets` is initialized to a list of `num_buckets` empty lists. To store or look up an entry with key `key`, we use the hash function `%` to convert `key` into an integer and use that integer to index into `buckets` to find the hash bucket associated with `key`. We then search that bucket (which, remember, is a list) linearly to see if there is an entry with the key `key`. If we are doing a lookup and there is an entry with the key, we simply return the value stored with that key. If there is no entry with that key, we return `None`. If a value is to be stored, we first check if there is already an entry with that key in the hash bucket. If so, we replace the entry with a new tuple; otherwise we append a new entry to the bucket.

There are many other ways to handle collisions, some considerably more efficient than using lists. But this is probably the simplest mechanism, and it works fine if the hash table is large enough relative to the number of elements stored in it, and the hash function provides a good enough approximation to a uniform distribution.

```

class Int_dict(object):
    """A dictionary with integer keys"""

    def __init__(self, num_buckets):
        """Create an empty dictionary"""
        self.buckets = []
        self.num_buckets = num_buckets
        for i in range(num_buckets):
            self.buckets.append([])

    def add_entry(self, key, dict_val):
        """Assumes key an int. Adds an entry."""
        hash_bucket = self.buckets[key%self.num_buckets]
        for i in range(len(hash_bucket)):
            if hash_bucket[i][0] == key:
                hash_bucket[i] = (key, dict_val)
                return
        hash_bucket.append((key, dict_val))

    def get_value(self, key):
        """Assumes key an int.
           Returns value associated with key"""
        hash_bucket = self.buckets[key%self.num_buckets]
        for e in hash_bucket:
            if e[0] == key:
                return e[1]
        return None

    def __str__(self):
        result = '{'
        for b in self.buckets:
            for e in b:
                result += f'{e[0]}:{e[1]},'
        return result[:-1] + '}' #result[:-1] omits the last comma

```

[Figure 12-7](#) Implementing dictionaries using hashing

Notice that the `__str__` method produces a representation of a dictionary that is unrelated to the order in which elements were added to it, but is instead ordered by the values to which the keys happen to hash.

The following code first constructs an `Int_dict` with 17 buckets and 20 entries. The values of the entries are the integers 0 to 19. The keys are chosen at random, using `random.choice`, from integers in the range 0 to  $10^5 - 1$ . (We discuss the `random` module in Chapters 16 and

17.) The code then prints the `Int_dict` using the `__str__` method defined in the class.

```
import random
D = Int_dict(17)
for i in range(20):
    #choose a random int in the range 0 to 10**5 - 1
    key = random.choice(range(10**5))
    D.add_entry(key, i)
print('The value of the Int_dict is:')
print(D)
```

When we ran this code it printed<sup>79</sup>

```
The value of the Int_dict is:
{99740:6, 61898:8, 15455:4, 99913:18, 276:19, 63944:13, 79618:17, 5
1093:15, 8271:2, 3715:14, 74606:1, 33432:3, 58915:7, 12302:12, 5672
3:16, 27519:11, 64937:5, 85405:9, 49756:10, 17611:0}
```

The following code prints the individual hash buckets by iterating over `D.buckets`. (This is a terrible violation of information hiding, but pedagogically useful.)

```
print('The buckets are:')
for hash_bucket in D.buckets: #violates abstraction barrier
    print(' ', hash_bucket)
```

When we ran this code it printed

```
The buckets are:
[]
[(99740, 6), (61898, 8)]
[(15455, 4)]
[]
[(99913, 18), (276, 19)]
[]
[]
[(63944, 13), (79618, 17)]
[(51093, 15)]
[(8271, 2), (3715, 14)]
[(74606, 1), (33432, 3), (58915, 7)]
[(12302, 12), (56723, 16)]
[]
[(27519, 11)]
[(64937, 5), (85405, 9), (49756, 10)]
```

```
[ ]  
[(17611, 0)]
```

When we violate the abstraction barrier and peek at the representation of the `Int_dict`, we see that some of the hash buckets are empty. Others contain one or more entries—depending upon the number of collisions that occurred.

What is the complexity of `get_value`? If there were no collisions it would be `constant time` because each hash bucket would be of length 0 or 1. But, of course, there might be collisions. If everything hashed to the same bucket, it would be `linear` in the number of entries in the dictionary, because the code would perform a linear search on that hash bucket. By making the hash table large enough, we can reduce the number of collisions sufficiently to allow us to treat the complexity as `constant time`. That is, we can trade space for time. But what is the tradeoff? To answer this question, we need to use a tiny bit of probability, so we defer the answer to Chapter 17.

---

## 12.4 Terms Introduced in Chapter

search algorithm

search space

pointer

indirection

binary search

wrapper function

amortized complexity

selection sort

loop invariant

divide and conquer algorithms

recursive base

merge sort

in-place sort  
quicksort  
timsort  
stable sort  
hash table  
hash function  
many-to-one mapping  
collision  
uniform distribution  
hash bucket

---

- 68 *Introduction to Algorithms*, by Cormen, Leiserson, Rivest, and Stein, is an excellent source for those of you not intimidated by a fair amount of mathematics.
- 69. The number of bits used to store an integer, often called the word size, is typically dictated by the hardware of the computer.
- 70 Of size 32 bits in some implementations and 64 bits in others.
- 71 My dictionary defines the noun “indirection” as “lack of straightforwardness and openness: deceitfulness.” In fact, the word generally had a pejorative implication until about 1950, when computer scientists realized that it was the solution to many problems.
- 72 It has often been said that “any problem in computing can be solved by adding another level of indirection.” Following three levels of indirection, we attribute this observation to David J. Wheeler. The paper “Authentication in Distributed Systems: Theory and Practice,” by Butler Lampson *et al.*, contains the observation. It also contains a footnote saying that “Roger Needham attributes this observation to David Wheeler of Cambridge University.”

- 73 Recall that when looking at orders of growth the base of the logarithm is irrelevant.
- 74 But not the most inefficient of sorting algorithms, as suggested by a successful candidate for the U.S. Presidency. See [http://www.youtube.com/watch?v=k4RRi\\_ntQc8](http://www.youtube.com/watch?v=k4RRi_ntQc8).
- 75 See <https://www.youtube.com/watch?v=o7pMRILvSws> for a video demonstrating selection sort.
- 76 See <https://www.youtube.com/watch?v=VP5CTeUp2kg> for a dramatization of merge sort.
- 77 **Quicksort**, which was invented by C.A.R. Hoare in 1960, is conceptually similar to merge sort, but considerably more complex. It has the advantage of needing only  $\log(n)$  additional space. Unlike merge sort, its running time depends upon the way the elements in the list to be sorted are ordered relative to each other. Though its worst-case running time is  $O(n^2)$ , its expected running time is only  $O(n * \log(n))$ .
- 78 Timsort was invented by Tim Peters in 2002 because he was unhappy with the previous algorithm used in Python.
- 79 Since the integers were chosen at random, you will probably get different results if you run it.

# 13

## PLOTTING AND MORE ABOUT CLASSES

Often text is the best way to communicate information, but sometimes there is a lot of truth to the Chinese proverb, 圖片的意義可以表達近萬字. Yet most programs rely on textual output to communicate with their users. Why? Because in many programming languages, presenting visual data is too hard. Fortunately, it is simple to do in Python.

---

### 13.1 Plotting Using Matplotlib

**Matplotlib** is a Python library module that provides **plotting** facilities very much like those in MATLAB, “a high-level technical computing language and interactive environment for algorithm development, data visualization, data analysis, and numeric computation.”<sup>80</sup> Later in the book we will look other Python libraries that provide other MATLAB-like capabilities. In this chapter, we focus on some simple ways of plotting data. A complete user's guide to the plotting capabilities of Matplotlib is at the website

[plt.sourceforge.net/users/index.html](http://plt.sourceforge.net/users/index.html)

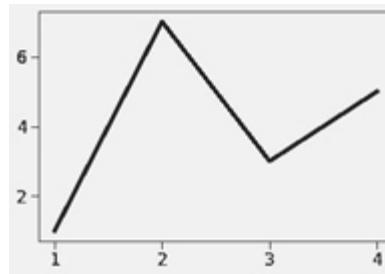
We will not try to provide a user's guide or a complete tutorial here. Instead, in this chapter we will merely provide a few example plots and explain the code that generated them. We introduce many other plotting features in later chapters.

Let's start with a simple example that uses `plt.plot` to produce a single plot. Executing

```
import matplotlib.pyplot as plt  
plt.plot([1,2,3,4], [1,7,3,5]) #draw on current figure
```

will produce a plot similar to, but not identical to, the one in [Figure 13-1](#). Your plot will probably have a colored line.<sup>81</sup> Also, if you run this code with the default parameter settings of most installations of Matplotlib, the line will probably not be as thick as the line in [Figure 13-1](#). We have used nonstandard default values for line width and font sizes so that the figures will reproduce better in black and white. We discuss how this is done later in this section.

Where the plot appears on your monitor depends upon the Python environment you are using. In the version of Spyder used to produce this edition of this book, it appears, by default, in something called the “Plots pane.”



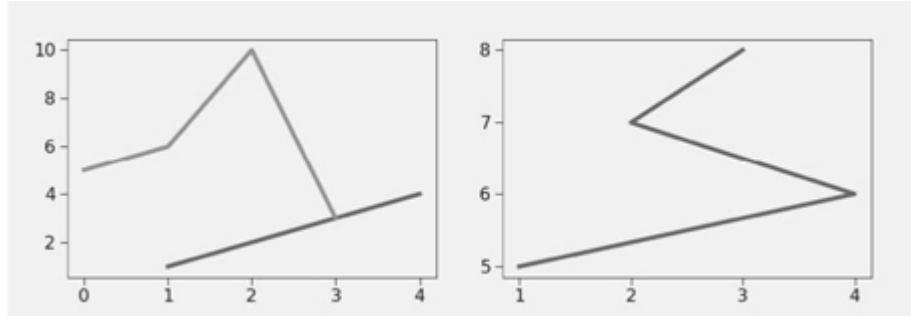
[Figure 13-1](#) A simple plot

It is possible to produce multiple **figures** and to write them to files. These files can have any name you like. By default, they will all have the file extension `.png`, but you can change this to other formats (e.g., `.jpg`) using the keyword parameter `format`.

The code

```
plt.figure(1)                      #create figure 1
plt.plot([1,2,3,4], [1,2,3,4])    #draw on figure 1
plt.figure(2)                      #create figure 2
plt.plot([1,4,2,3], [5,6,7,8])    #draw on figure 2
plt.savefig('Figure-Addie')        #save figure 2
plt.figure(1)                      #go back to working on figure
1
plt.plot([5,6,10,3])              #draw again on figure 1
plt.savefig('Figure-Jane')         #save figure 1
```

produces and saves to files named `Figure-Jane.png` and `Figure-Addie.png` the two plots in [Figure 13-2](#).



[Figure 13-2](#) Contents of Figure-Jane.png (left) and Figure-Addie.png (right)

Observe that the last call to `plt.plot` is passed only one argument. This argument supplies the `y` values. The corresponding `x` values default to the sequence yielded by `range(len([5, 6, 10, 3]))`, which is why they range from 0 to 3 in this case.

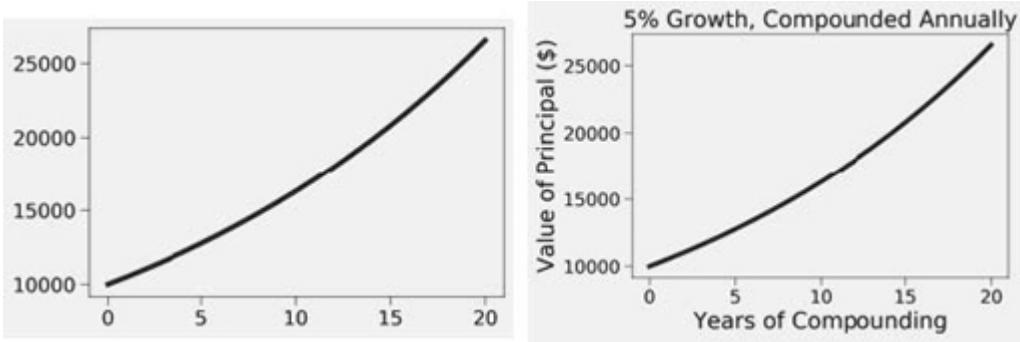
Matplotlib has a notion of **current figure**. Executing `plt.figure(x)` sets the current figure to the figure numbered `x`. Subsequently executed calls of plotting functions implicitly refer to that figure until another invocation of `plt.figure` occurs. This explains why the figure written to the file `Figure-Addie.png` was the second figure created.

Let's look at another example. The code on the left side of [Figure 13-3](#) produces the plot on the left in [Figure 13-4](#).

```
principal = 10000
interest_rate = 0.05
years = 20
values = []
for i in range(years + 1):
    values.append(principal)
    principal += principal*interest_rate
plt.plot(values)

plt.title('5% Growth, Compounded Annually')
plt.xlabel('Years of Compounding')
plt.ylabel('Value of Principal ($)')
```

[Figure 13-3](#) Produce plots showing compound growth

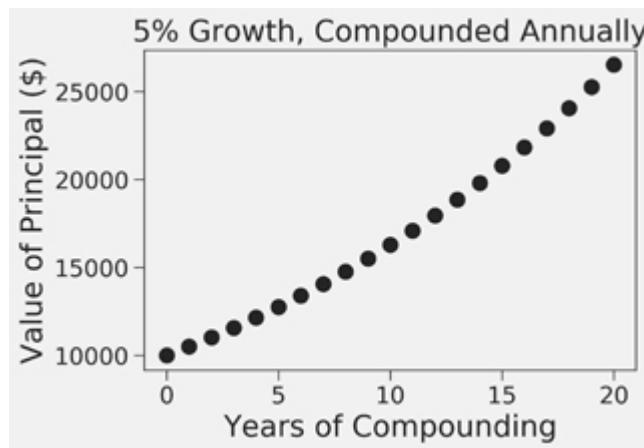


[Figure 13-4](#) Plots showing compound growth

If we look at the code, we can deduce that this is a plot showing the growth of an initial investment of \$10,000 at an annually compounded interest rate of 5%. However, this cannot be easily inferred by looking only at the plot itself. That's a bad thing. All plots should have informative titles, and all axes should be labeled. If we add to the end of our code the lines on the right of [Figure 13-3](#), we get the plot on the right in [Figure 13-4](#).

For every plotted curve, there is an optional argument that is a format string indicating the color and line type of the plot. The letters and symbols of the format string are derived from those used in MATLAB and are composed of a color indicator followed by an optional line-style indicator. The default format string is '`b-`', which produces a solid blue line. To plot the growth in principal with black circles, replace the call `plt.plot(values)` by `plt.plot(values, 'ko')`, which produces the plot in [Figure 13-5](#). For a complete list of color and line-style indicators, see

[http://plt.org/api/pyplot\\_api.html# plt.pyplot.plot](http://plt.org/api/pyplot_api.html# plt.pyplot.plot)

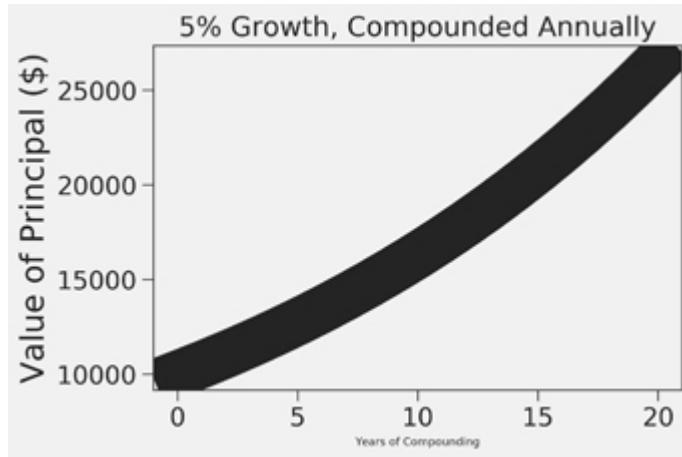


[Figure 13-5](#) Another plot of compound growth

It is also possible to change the type size and line width used in plots. This can be done using keyword arguments in individual calls to functions. For example, the code

```
principal = 10000 #initial investment
interestRate = 0.05
years = 20
values = []
for i in range(years + 1):
    values.append(principal)
    principal += principal*interestRate
plt.plot(values, '-k', linewidth = 30)
plt.title('5% Growth, Compounded Annually',
          fontsize = 'xx-large')
plt.xlabel('Years of Compounding', fontsize = 'x-small')
plt.ylabel('Value of Principal ($)')
```

produces the intentionally bizarre-looking plot in [Figure 13-6](#).



[Figure 13-6](#) Strange-looking plot

It is also possible to change the default values, which are known as “rc settings.” (The name “rc” is derived from the `.rc` file extension used for runtime configuration files in Unix.) These values are stored in a dictionary-like variable that can be accessed via the name `plt.rcParams`. So, for example, you can set the default line width to 6 points<sup>82</sup> by executing the code

```
plt.rcParams['lines.linewidth'] = 6.
```

There are an enormous number of `rcParams` settings. A complete list can be found at

<http://matplotlib.org/users/customizing.html>

If you don't want to worry about customizing individual parameters, there are pre-defined style sheets. A description of these can be found at

[http://matplotlib.org/users/style\\_sheets.html#style-sheets](http://matplotlib.org/users/style_sheets.html#style-sheets)

The values used in most of the remaining examples in this book were set with the code

```
#set line width
plt.rcParams['lines.linewidth'] = 4
#set font size for titles
plt.rcParams['axes.titlesize'] = 20
#set font size for labels on axes
```

```
plt.rcParams['axes.labelsize'] = 20
#set size of numbers on x-axis
plt.rcParams['xtick.labelsizes'] = 16
#set size of numbers on y-axis
plt.rcParams['ytick.labelsizes'] = 16
#set size of ticks on x-axis
plt.rcParams['xtick.major.size'] = 7
#set size of ticks on y-axis
plt.rcParams['ytick.major.size'] = 7
#set size of markers, e.g., circles representing points
plt.rcParams['lines.markersizes'] = 10
#set number of times marker is shown when displaying legend
plt.rcParams['legend.numpoints'] = 1
#Set size of type in legend
plt.rcParams['legend.fontsize'] = 14
```

If you are viewing plots on a color display, you will have little reason to change the default settings. We customized the settings so that it would be easier to read the plots when we shrank them to fit on the page and converted them to grayscale.

---

## 13.2 Plotting Mortgages, an Extended Example

In Chapter 10, we worked our way through a hierarchy of mortgages as a way of illustrating the use of subclassing. We concluded that chapter by observing that “our program should be producing plots designed to show how the mortgage behaves over time.” [Figure 13-7](#) enhances class `Mortgage` by adding methods that make it convenient to produce such plots. (The function `find_payment`, which appears in Figure 10-10, is discussed in Section 10.4.)

```

class Mortgage(object):
    """Abstract class for building different kinds of mortgages"""
    def __init__(self, loan, annRate, months):
        self._loan = loan
        self._rate = annRate/12.0
        self._months = months
        self._paid = [0.0]
        self._outstanding = [loan]
        self._payment = find_payment(loan, self._rate, months)
        self._legend = None #description of mortgage

    def make_payment(self):
        self._paid.append(self._payment)
        reduction = self._payment - self._outstanding[-1]*self._rate
        self._outstanding.append(self._outstanding[-1] - reduction)

    def get_total_paid(self):
        return sum(self._paid)
    def __str__(self):
        return self._legend

    def plot_payments(self, style):
        plt.plot(self._paid[1:], style, label = self._legend)

    def plot_balance(self, style):
        plt.plot(self._outstanding, style, label = self._legend)

    def plot_tot_pd(self, style):
        tot_pd = [self._paid[0]]
        for i in range(1, len(self._paid)):
            tot_pd.append(tot_pd[-1] + self._paid[i])
        plt.plot(tot_pd, style, label = self._legend)

    def plot_net(self, style):
        tot_pd = [self._paid[0]]
        for i in range(1, len(self._paid)):
            tot_pd.append(tot_pd[-1] + self._paid[i])
        equity_acquired = np.array([self._loan]*len(self._outstanding))
        equity_acquired = equity_acquired-np.array(self._outstanding)
        net = np.array(tot_pd) - equity_acquired
        plt.plot(net, style, label = self._legend)

```

[Figure 13-7](#) Class `Mortgage` with plotting methods

The nontrivial methods in class `Mortgage` are `plot_tot_paid` and `plot_net`. The method `plot_tot_paid` plots the cumulative total of the payments made. The method `plot_net` plots an approximation to the

total cost of the mortgage over time by plotting the cash expended minus the equity acquired by paying off part of the loan.[83](#)

The expression `np.array(self.outstanding)` in the function `plot_net` performs a type conversion. Thus far, we have been calling the plotting functions of Matplotlib with arguments of type `list`. Under the covers, Matplotlib has been converting these lists into a different type, `array`, which is part of the `numpy` module. The importation `import numpy as np` and the invocation `np.array` makes this explicit.

`Numpy` is a Python module that provides tools for scientific computing. In addition to providing multi-dimensional arrays, it provides a variety of mathematical capabilities. We will see more of `numpy` later in this book.

There are many convenient ways to manipulate arrays that are not readily available for lists. In particular, expressions can be formed using arrays and arithmetic operators. There are a number of ways to create arrays in `numpy`, but the most common one is to first create a list, and then convert it. Consider the code

```
import numpy as np
a1 = np.array([1, 2, 4])
print('a1 =', a1)
a2 = a1*2
print('a2 =', a2)
print('a1 + 3 =', a1 + 3)
print('3 - a1 =', 3 - a1)
print('a1 - a2 =', a1 - a2)
print('a1*a2 =', a1*a2)
```

The expression `a1*2` multiplies each element of `a1` by the constant 2. The expression `a1 + 3` adds the integer 3 to each element of `a1`. The expression `a1 - a2` subtracts each element of `a2` from the corresponding element of `a1` (If the arrays had been of different length, an error would have occurred.) The expression `a1*a2` multiplies each element of `a1` by the corresponding element of `a2`. When the above code is run it prints

```
a1 = [1 2 4]
a2 = [2 4 8]
a1 + 3 = [4 5 7]
3 - a1 = [ 2  1 -1]
```

```
a1 - a2 = [-1 -2 -4]
a1*a2 = [ 2   8  32]
```

[Figure 13-8](#) repeats the three subclasses of `Mortgage` from Figure 10-11. Each has a distinct `__init__` method that overrides the `__init__` method in `Mortgage`. The subclass `Two_rate` also overrides the `make_payment` method of `Mortgage`.

```
class Fixed(Mortgage):
    def __init__(self, loan, r, months):
        Mortgage.__init__(self, loan, r, months)
        self._legend = f'Fixed, {r*100:.1f}%'"

class Fixed_with_pts(Mortgage):
    def __init__(self, loan, r, months, pts):
        Mortgage.__init__(self, loan, r, months)
        self._pts = pts
        self._paid = [loan*(pts/100)]
        self._legend = f'Fixed, {r*100:.1f}%, {pts} points'

class Two_rate(Mortgage):
    def __init__(self, loan, r, months, teaser_rate, teaser_months):
        Mortgage.__init__(self, loan, teaser_rate, months)
        self._teaser_months = teaser_months
        self._teaser_rate = teaser_rate
        self._nextRate = r/12
        self._legend = (f'{100*teaser_rate:.1f}% for ' +
                       f'{self._teaser_months} months, then {100*r:.1f}%')

    def make_payment(self):
        if len(self._paid) == self._teaser_months + 1:
            self._rate = self._nextRate
            self._payment = find_payment(self._outstanding[-1],
                                         self._rate,
                                         self._months - self._teaser_months)
        Mortgage.make_payment(self)
```

[Figure 13-8](#) Subclasses of `Mortgage`

[Figure 13-9](#) and [Figure 13-10](#) contain functions that can be used to generate plots intended to provide insight about different kinds of mortgages.

The function `compare_mortgages`, [Figure 13-9](#), creates a list of different kinds of mortgages and simulates making a series of

payments on each. It then calls `plot_mortgages`, [Figure 13-10](#), to produce the plots.

```
def compare_mortgages(amt, years, fixed_rate, pts, pts_rate,
                      var_rate1, var_rate2, var_months):
    tot_months = years*12
    fixed1 = Fixed(amt, fixed_rate, tot_months)
    fixed2 = Fixed_with_pts(amt, pts_rate, tot_months, pts)
    two_rate = Two_rate(amt, var_rate2, tot_months, var_rate1, var_months)
    morts = [fixed1, fixed2, two_rate]
    for m in range(tot_months):
        for mort in morts:
            mort.make_payment()
    plot_mortgages(morts, amt)
```

[Figure 13-9](#) Compare mortgages

```

def plot_mortgages(morts, amt):
    def label_plot(figure, title, x_label, y_label):
        plt.figure(figure)
        plt.title(title)
        plt.xlabel(x_label)
        plt.ylabel(y_label)
        plt.legend(loc = 'best')
    styles = ['k-', 'k-.', 'k:']
    #Give names to figure numbers
    payments, cost, balance, net_cost = 0, 1, 2, 3
    for i in range(len(morts)):
        plt.figure(payments)
        morts[i].plot_payments(styles[i])
        plt.figure(cost)
        morts[i].plot_tot_pd(styles[i])
        plt.figure(balance)
        morts[i].plot_balance(styles[i])
        plt.figure(net_cost)
        morts[i].plot_net(styles[i])
    label_plot(payments, f'Monthly Payments of ${amt:,} Mortages',
               'Months', 'Monthly Payments')
    label_plot(cost, f'Cash Outlay of ${amt:,} Mortgages',
               'Months', 'Total Payments')
    label_plot(balance, f'Balance Remaining of ${amt:,} Mortages',
               'Months', 'Remaining Loan Balance of $')
    label_plot(net_cost, f'Net Cost of ${amt:,} Mortgages',
               'Months', 'Payments - Equity $')

```

[Figure 13-10](#) Generate mortgage plots

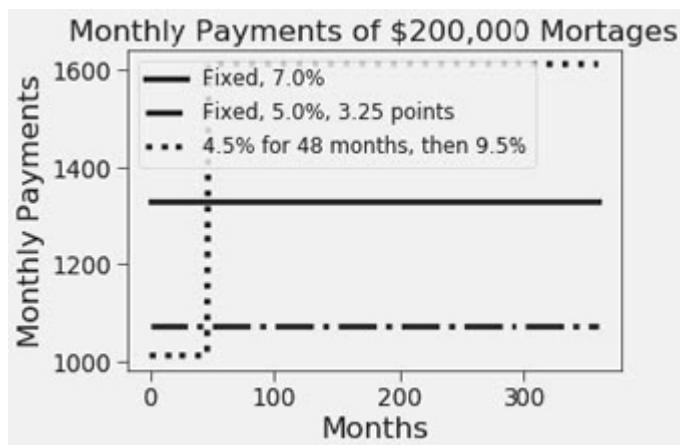
The function `plot_mortgages` in [Figure 13-10](#) uses the plotting methods in `Mortgage` to produce plots containing information about each of three kinds of mortgages. The loop in `plot_mortgages` uses the index `i` to select elements from the lists `morts` and `styles` so that different kinds of mortgages are represented in a consistent way across figures. For example, since the third element in `morts` is a variable-rate mortgage and the third element in `styles` is `'k:'`, the variable-rate mortgage is always plotted using a black dotted line. The local function `label_plot` is used to generate appropriate titles and axis labels for each plot. The calls of `plt.figure` ensure that titles and labels are associated with the appropriate plot.

The call

```
compare_mortgages(amt=200000, years=30, fixed_rate=0.07,
                  pts = 3.25, pts_rate=0.05, var_rate1=0.045,
                  var_rate2=0.095, var_months=48)
```

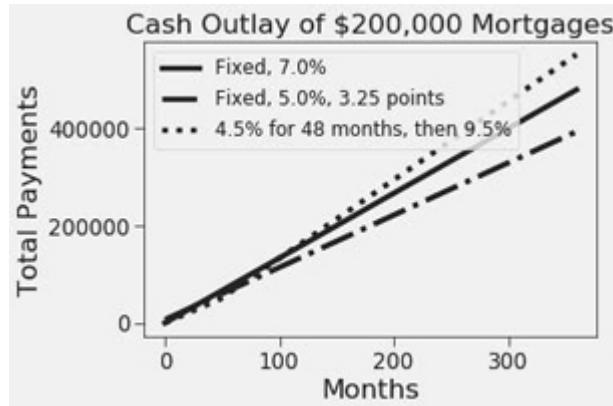
produces plots ([Figure 13-11 through 13-13](#)) that compare three kinds of mortgages.

The plot shown in [Figure 13-11](#), which was produced by invocations of `plot_payments`, simply plots each payment of each mortgage against time. The box containing the key appears where it does because of the value supplied to the keyword argument `loc` used in the call to `plt.legend`. When `loc` is bound to `'best'`, the location is chosen automatically. This plot makes it clear how the monthly payments vary (or don't) over time but doesn't shed much light on the relative costs of each kind of mortgage.



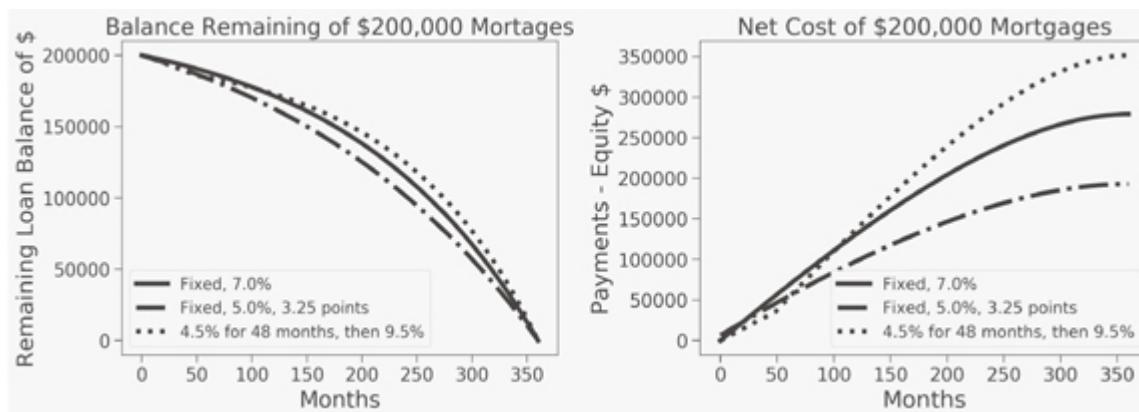
[Figure 13-11](#) Monthly payments of different kinds of mortgages

The plot in [Figure 13-12](#) was produced by invocations of `plot_tot_pd`. It compares the cost of each kind of mortgage by plotting the cumulative costs that have been incurred at the start of each month.



[Figure 13-12](#) Cost over time of different kinds of mortgages

The plots in [Figure 13-13](#) show the remaining debt (on the left) and the total net cost of having the mortgage (on the right).



[Figure 13-13](#) Balance remaining and net cost for different kinds of mortgages

### 13.3 An Interactive Plot for an Infectious Disease

As I put the final touches on this book, I am at home following “social distancing” restrictions related to restricting the spread of the Covid-19 disease. Like many respiratory viruses, the SARS-CoV-2 virus is spread primarily by human-to-human contact. Social distancing is designed to reduce contacts between humans, and thus limit the spread of the disease caused by the virus.

[Figure 13-14](#) contains a simplistic simulation of incidence of an infectious disease over time. The parameter `fixed` is a dictionary defining the initial values for key variables related to the spread of infections. The parameter `variable` is a dictionary defining variables related to social distancing. Later, we show how the value of `variable` can be changed in an interactive plot.

```
def simulation(fixed, variable):
    infected = [fixed['initial_infections']]
    new_infections = [fixed['initial_infections']]
    total_infections = fixed['initial_infections']

    for t in range(fixed['duration']):
        cur_infections = infected[-1]
        # remove people who are no longer contagious
        if len(new_infections) > fixed['days_spreading']:
            cur_infections -= new_infections[-fixed['days_spreading']: -1]
        # if social distancing, change number of daily contacts
        if t >= variable['red_start'] and t < variable['red_end']:
            daily_contacts = variable['red_daily_contacts']
        else:
            daily_contacts = fixed['init_contacts']
        # compute number of new cases
        total_contacts = cur_infections * daily_contacts
        susceptible = fixed['pop'] - total_infections
        risky_contacts = total_contacts * (susceptible/fixed['pop'])
        newly_infected = round(risky_contacts * fixed['contagiousness'])
        # update variables
        new_infections.append(newly_infected)
        total_infections += newly_infected
        infected.append(cur_infections + newly_infected)
```

[Figure 13-14](#) Simulation of spread of an infectious disease

Later in the book, we talk in detail about simulation models. Here, however, we are focused on interactive plotting, and the purpose of the simulation is to provide us with something interesting to plot. If you don't understand the details of the simulation, that's okay.

[Figure 13-15](#) contains a function that produces a static plot showing the number of infected people on each day. It also contains a **text box** showing the total number of infected people. The

statement starting with `txt_box = plt.text` instructs Python to start the text specified by the third argument of `plt.text` at a location specified by the first two arguments. The expression `plt.xlim()[1]/2` places the left edge of the text halfway between the left end of the x-axis (0 for this plot) and the right end of the x-axis. The expression `plt.ylim()[1]/1.25` places the text 80% of the way from the bottom of the y-axis (0 on this plot) to the top of the y-axis.

```
def plot_infections(infections, total_infections, fixed):
    infection_plot = plt.plot(infections, 'r', label = 'Infected')[0]
    plt.xticks(fontsize = 'large')
    plt.yticks(fontsize = 'large')
    plt.xlabel('Days Since First Infection', fontsize = 'xx-large')
    plt.ylabel('Number Currently Infected', fontsize = 'xx-large')
    plt.title('Number of Infections Assuming No Vaccine\n' +
              f'Pop = {fixed["pop"]}, ' +
              f'Contacts/Day = {fixed["init_contacts"]}, ' +
              f'Infectivity = {(100*fixed["contagiousness"]):.1f}%, ' +
              f'Days Contagious = {fixed["days_spreading"]}', 
              fontsize = 'xx-large')
    plt.legend(fontsize = 'xx-large')
    txt_box = plt.text(plt.xlim()[1]/2, plt.ylim()[1]/1.25,
                      f'Total Infections = {total_infections:.0f}',
                      fontdict = {'size':'xx-large', 'weight':'bold',
                                  'color':'red'})
    return infection plot, txt box
```

[Figure 13-15](#) Function to plot history of infection

[Figure 13-16](#) uses the functions in [Figure 13-14](#) and [Figure 13-15](#) to produce a plot, [Figure 13-17](#), showing the number of infected people—assuming no social distancing. The values in `fixed` are not based on a specific disease. It might seem surprising to assume that on average an individual comes into “contact” with 50 people a day. Keep in mind, however, that this number includes indirect contact, e.g., riding on the same bus as an infected person or touching an object that might have had a pathogen deposited on it by an infected person.

```

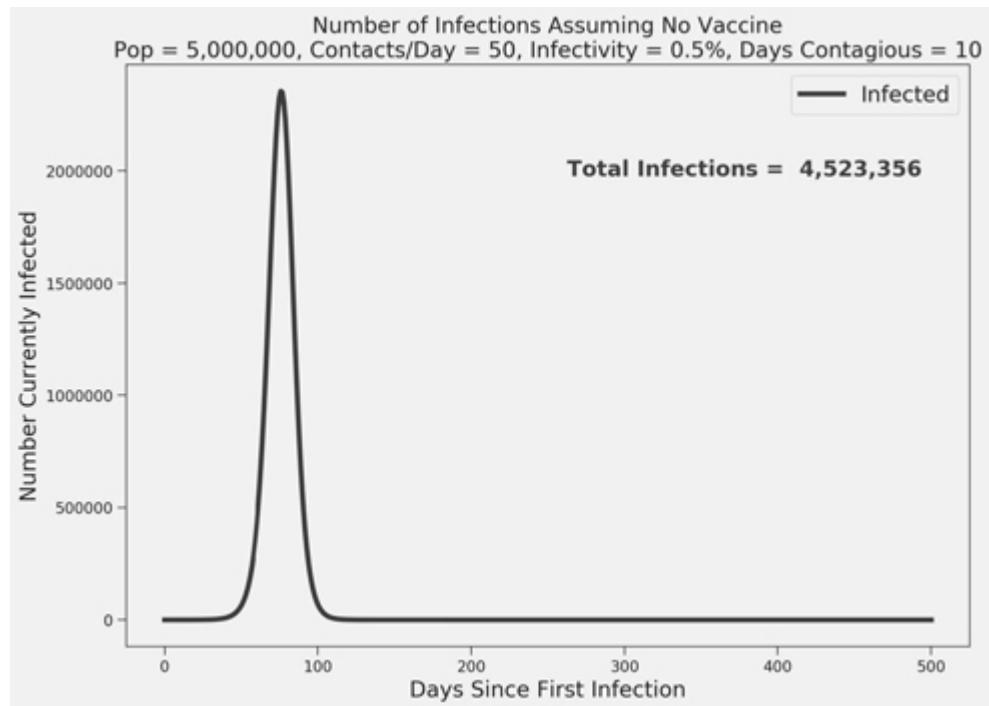
fixed = {
    'pop': 5000000, # population at risk
    'duration': 500, # number of days for simulation
    'initial_infections': 4, # initial number of cases
    'init_contacts': 50, #contacts without social distancing
    'contagiousness': 0.005, # prob. of getting disease if exposed
    'days_spreading': 10} # days contagious after infection

variable = {
    # 'red_daily_contacts': 4, # social distancing
    'red_daily_contacts': fixed['init_contacts'], # social distancing
    'red_start': 20, # start of social distancing
    'red_end': 200} # end of social distancing

infections, total_infections = simulation(fixed, variable)
fig = plt.figure(figsize=(12, 8.5))
plot_infections(infections, total_infections, fixed)

```

[Figure 13-16](#) Produce plot with a single set of parameters



[Figure 13-17](#) Static plot of number of infections

The plot shows a rapid rise in the number of current infections, followed by a rapid decline to a stable state of zero current infections. The rapid growth occurs because each infected person infects

multiple other people, so the number of people capable of spreading the infection grows exponentially. The steady state of no new infections occurs because the population has achieved **herd immunity**. When a sufficiently large fraction of a population is immune to a disease (and we are assuming that people who have recovered from this disease cannot get it again), there are long periods when nobody contracts the disease, which eventually leads to there being nobody left to spread it.<sup>84</sup> If we want to explore the impact of different parameter settings, we could change the values of some of the variables in `fixed`, and produce another plot. That, however, is a rather cumbersome way to explore “what if” scenarios. Instead, let’s produce a figure that contains **sliders**<sup>85</sup> that can be used to dynamically alter the key parameters related to social distancing: `reduced_contacts_per_day`, `red_start`, and `red_end`.

The figure will have four independent components: the main plot and one slider for each of the elements of the dictionary variable. We start by describing the layout of the figure by specifying its overall dimensions (12 inches wide and 8.5 inches high), the location (specified in the same way as for a text box), and dimensions (relative to the size of the entire figure) of each component. We also bind a name to each of these components, so that we can refer to them later.

```
fig = plt.figure(figsize=(12, 8.5))
infections_ax = plt.axes([0.12, 0.2, 0.8, 0.65])
contacts_ax = plt.axes([0.25, 0.09, 0.65, 0.03])
start_ax = plt.axes([0.25, 0.06, 0.65, 0.03])
end_ax = plt.axes([0.25, 0.03, 0.65, 0.03])
```

The next lines of code define three sliders, one for each value we want to vary. First, we import a module that contains a class `Slider`.

```
from Matplotlib.widgets import Slider
```

Next, we create three sliders, binding each to a variable.

```
contacts_slider = Slider(
    contacts_ax,    # axes object containing
the slider        'reduced\ncontacts/day',   # name of
slider           0,      # minimal value of the parameter
```

```

    50,    # maximal value of the parameter
    50)   # initial value of the parameter)
contacts_slider.label.set_fontsize(12)
start_day_slider = Slider(start_ax, 'start reduction', 1,
30, 20)
start_day_slider.label.set_fontsize(12)
end_day_slider = Slider(end_ax, 'end reduction', 30, 400,
200)
end_day_slider.label.set_fontsize(12)

```

Next, we provide a function, `update`, that updates the plot based upon the current values of the sliders.

```

def update(fixed, infection_plot, txt_box,
           contacts_slider, start_day_slider,
           end_day_slider):
    variable = {'red_daily_contacts': contacts_slider.val,
                'red_start': start_day_slider.val,
                'red_end': end_day_slider.val}
    I, total_infections = simulation(fixed, variable)
    infection_plot.set_ydata(I)    # new y-coordinates for
plot
    txt_box.set_text(f'Total Infections =
{total_infections:.0f}')

```

Next, we need to instruct Python to call `update` whenever the value of a slider is changed. This is a bit tricky. The `Slider` class contains a method, `on_changed`, which takes an argument of type `function` that is invoked whenever the slider is changed. This function always takes exactly one argument, a number representing the current value of the slider. In our case, however, each time a slider is changed we want to run the simulation using the values of all three sliders and the values the dictionary `fixed`.

We solve the problem by introducing a new function that is a suitable argument for `on_changed`. The function `slider_update` takes the mandated numeric argument, but it doesn't use it. Instead, the lambda expression defining `slider_update` captures the objects to which `fixed`, `infection_plot`, `txt_box`, and the three sliders are bound. It then calls `update` with these arguments.

```

slider_update = lambda _: update(fixed, infection_plot,
                                txt_box,
                                contacts_slider,
                                start_day_slider,

```

```

        end_day_slider)
contacts_slider.on_changed(slider_update)
start_day_slider.on_changed(slider_update)
end_day_slider.on_changed(slider_update)

```

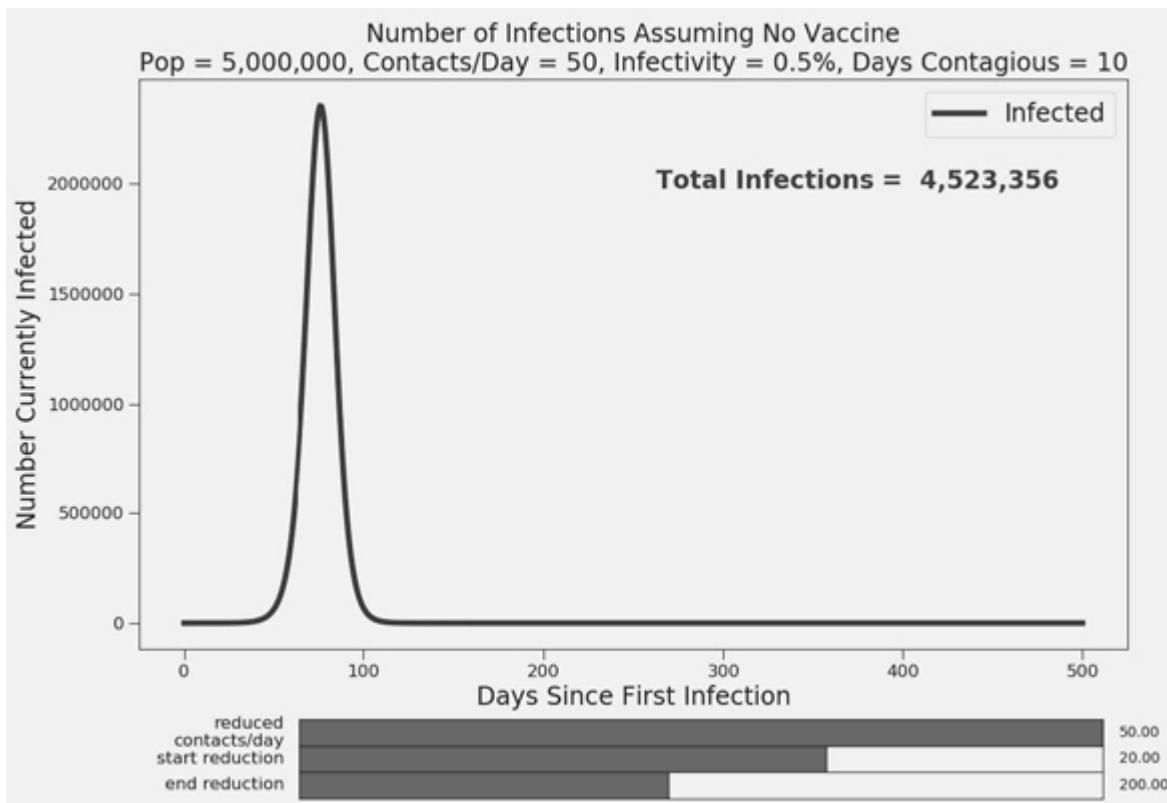
Finally, we plot the curve of infections and update the text box in the portion of the figure bound to `infections_ax`.

```

infections, total_infections = simulation(fixed, variable)
plt.axes(infections_ax)
infection_plot, txt_box = plot_infections(infections,
                                            total_infections,
                                            fixed)

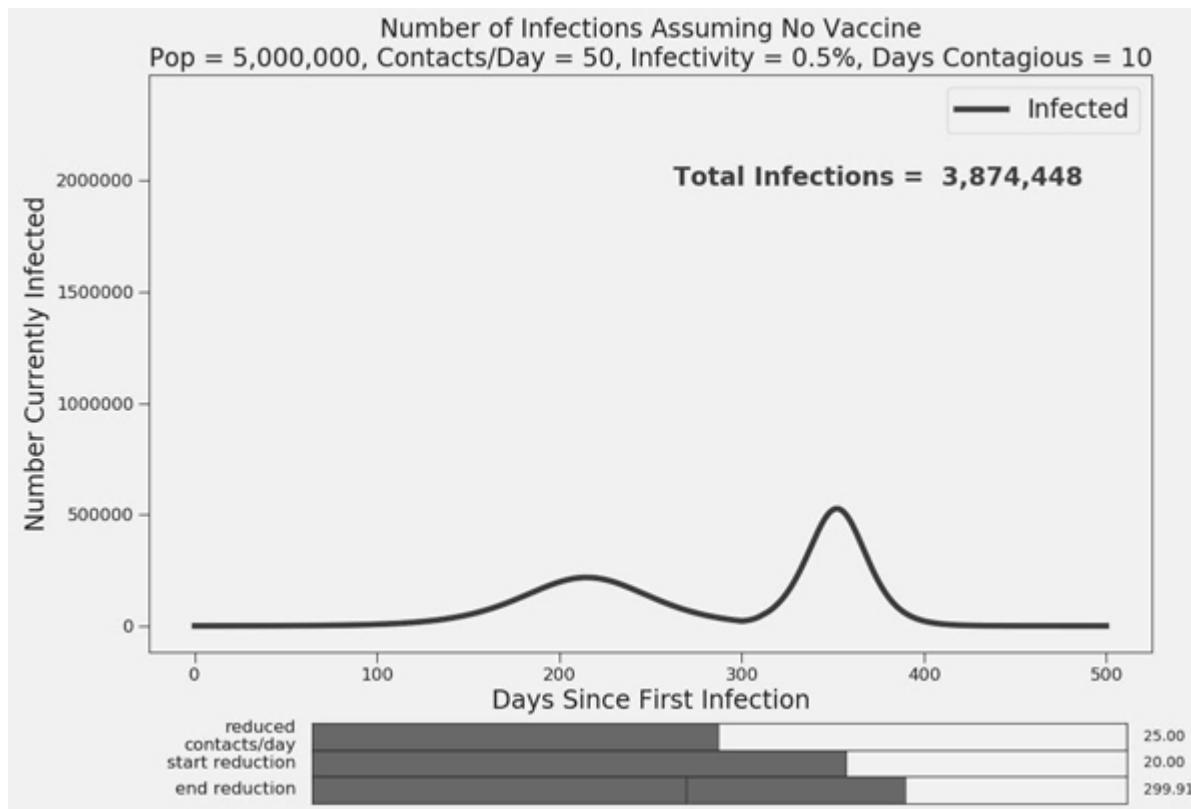
```

When this code is run, it produces the plot in [Figure 13-18](#).<sup>86</sup>



[Figure 13-18](#) Interactive plot with initial slider values

Now, we can easily experiment with many combinations of slider values, one of which is shown in [Figure 13-19](#).



[Figure 13-19](#) Interactive plot with changed slider values

[Figure 13-19](#) shows that if contacts are reduced to an average of 25 a day after 20 days and held at the level for 40 weeks, the total number of infections is reduced. More significantly, the peak number of infections (and therefore the maximum burden on the healthcare system) is dramatically reduced. This is frequently referred to as **flattening the curve**.

---

## 13.4 Terms Introduced in Chapter

plot

Matplotlib

figure

current figure

rcParams  
array  
numpy  
interactive plot  
text box  
herd immunity  
slider  
flattening the curve

---

80 [www.mathworks.com/products/matlab/description1.html?  
s\\_cid=ML\\_b1008\\_desintro](http://www.mathworks.com/products/matlab/description1.html? s_cid=ML_b1008_desintro)

- 81 In order to keep the price down, we chose to publish this book in black and white. That posed a dilemma: should we discuss how to use color in plots or not? We concluded that color is too important to ignore. However, we did try to ensure that the plots make sense in grayscale.
- 82 The point is the smallest unit measure used in typography. Over the years, it has varied in size from about 0.18 to 0.4 millimeters. The desktop publishing point (DTP) has become the *de facto* standard. It is equal to  $1/72$  of an inch, which is 0.3527mm.
- 83 It is an approximation because it does not perform a net present value calculation to take into account the time value of cash.
- 84 This kind of trajectory is typical of outbreaks of infectious diseases. To view a similarly shaped plot produced in 1852 describing the 1849 Cholera epidemic in England, take a look at [https://images.slideplayer.com/13/4177382/slides/slide\\_19.jpg](https://images.slideplayer.com/13/4177382/slides/slide_19.jpg).
- 85 In this context, we mean an interactive object that can be used to change the value of a variable—not a small hamburger or a breaking ball.

86 To use the interactive plot, you might need to change your Python preferences. If the plot shows up inline rather than in a separate window, go to Preferences->IPythonConsole->Graphics, set backend to automatic, and then restart the IPython console.

# 14

## KNAPSACK AND GRAPH OPTIMIZATION PROBLEMS

The notion of an optimization problem provides a structured way to think about solving lots of computational problems. Whenever you set about solving a problem that involves finding the biggest, the smallest, the most, the fewest, the fastest, the least expensive, etc., there is a good chance that you can map the problem onto a classic optimization problem for which there is a known computational solution.

In general, an **optimization problem** has two parts:

- An **objective function** to be maximized or minimized. For example, the airfare between Boston and Istanbul.
- A **set of constraints** (possibly empty) that must be honored. For example, an upper bound on the travel time.

In this chapter, we introduce the notion of an optimization problem and give a few examples. We also provide some simple algorithms that solve them. In Chapter 15, we discuss an efficient way of solving an important class of optimization problems.

The main things to take away from this chapter are:

- Many problems of real importance can be formulated in a simple way that leads naturally to a computational solution.
- Reducing a seemingly new problem to an instance of a well-known problem allows you to use preexisting solutions.

- Knapsack problems and graph problems are classes of problems to which other problems can often be reduced.
- Exhaustive enumeration algorithms provide a simple, but usually computationally intractable, way to search for optimal solutions.
- A greedy algorithm is often a practical approach to finding a pretty good, but not always optimal, solution to an optimization problem.

As usual, we will supplement the material on computational thinking with a few bits of Python and some tips about programming.

## 14.1 Knapsack Problems

It's not easy being a burglar. In addition to the obvious problems (making sure that a home is empty, picking locks, circumventing alarms, dealing with ethical quandaries, etc.), a burglar has to decide what to steal. The problem is that most homes contain more things of value than the average burglar can carry away. What's a poor burglar to do? He (or she) needs to find the set of things that provides the most value without exceeding his or her carrying capacity.

Suppose, for example, a burglar who has a knapsack<sup>[87](#)</sup> that can hold at most 20 pounds of loot breaks into a house and finds the items in [Figure 14-1](#). Clearly, he will not be able to fit them all in his knapsack, so he needs to decide what to take and what to leave behind.

Item	Value	Weight	Value/Weight
Clock	175	10	17.5
Painting	90	9	10
Radio	20	4	5
Vase	50	2	25
Book	10	1	10
Computer	200	20	10

[Figure 14-1](#) Table of items

### 14.1.1 Greedy Algorithms

The simplest way to find an approximate solution to this problem is to use a **greedy algorithm**. The thief would choose the best item first, then the next best, and continue until he reached his limit. Of course, before doing this, the thief would have to decide what “best” should mean. Is the best item the most valuable, the least heavy, or maybe the item with the highest value-to-weight ratio? If he chose highest value, he would leave with just the computer, which he could fence for \$200. If he chose lowest weight, he would take, in order, the book, the vase, the radio, and the painting—which would be worth a total of \$170. Finally, if he decided that best meant highest value-to-weight ratio, he would start by taking the vase and the clock. That would leave three items with a value-to-weight ratio of 10, but of those only the book would still fit in the knapsack. After taking the book, he would take the remaining item that still fit, the radio. The total value of his loot would be \$255.

Though greedy-by-density (value-to-weight ratio) happens to yield the best result for this data set, there is no guarantee that a greedy-by-density algorithm always finds a better solution than greedy by weight or value. More generally, there is no guarantee that any solution to this kind of knapsack problem found by a greedy algorithm will be optimal.<sup>88</sup> We will discuss this issue in more detail a bit later.

The code in the next three figures implements all three of these greedy algorithms. In [Figure 14-2](#) we define class `Item`. Each `Item` has a `name`, `value`, and `weight` attribute. We also define three functions that can be bound to the argument `key_function` of our implementation of `greedy`; see [Figure 14-3](#).

```

class Item(object):
    def __init__(self, n, v, w):
        self._name = n
        self._value = v
        self._weight = w
    def get_name(self):
        return self._name
    def get_value(self):
        return self._value
    def get_weight(self):
        return self._weight
    def __str__(self):
        return f'{self._name}, {self._value}, {self._weight}'

def value(item):
    return item.get_value()

def weight_inverse(item):
    return 1.0/item.get_weight()

def density(item):
    return item.get_value()/item.get_weight()

```

[Figure 14-2](#) Class `Item`

```

def greedy(items, max_weight, key_function):
    """Assumes items a list, max_weight >= 0,
       key_function maps elements of items to numbers"""
    items_copy = sorted(items, key=key_function, reverse = True)
    result = []
    total_value, total_weight = 0.0, 0.0
    for i in range(len(items_copy)):
        if (total_weight + items_copy[i].get_weight()) <= max_weight:
            result.append(items_copy[i])
            total_weight += items_copy[i].get_weight()
            total_value += items_copy[i].get_value()
    return (result, total_value)

```

[Figure 14-3](#) Implementation of a greedy algorithm

By introducing the parameter `key_function`, we make `greedy` independent of the order in which to consider the elements of the list. All that is required is that `key_function` defines an ordering on the elements in `items`. We then use this ordering to produce a sorted

list containing the same elements as `items`. We use the built-in Python function `sorted` to do this. (We use `sorted` rather than `sort` because we want to generate a new list rather than mutate the list passed to the function.) We use the `reverse` parameter to indicate that we want the list sorted from largest (with respect to `key_function`) to smallest.

What is the algorithmic efficiency of `greedy`? There are two things to consider: the time complexity of the built-in function `sorted`, and the number of times through the `for` loop in the body of `greedy`. The number of iterations of the loop is bounded by the number of elements in `items`, i.e., it is  $\theta(n)$ , where `n` is the length of `items`. However, the worst-case time for Python's built-in sorting function is roughly order  $\theta(n \log n)$ , where `n` is the length of the list to be sorted.<sup>89</sup> Therefore the running time of `greedy` is order  $\theta(n \log n)$ .

The code in [Figure 14-4](#) builds a list of items and then tests the function `greedy` using different ways of ordering the list.

```

def build_items():
    names = ['clock', 'painting', 'radio', 'vase', 'book', 'computer']
    values = [175, 90, 20, 50, 10, 200]
    weights = [10, 9, 4, 2, 1, 20]
    Items = []
    for i in range(len(values)):
        Items.append(Item(names[i], values[i], weights[i]))
    return Items

def test_greedy(items, max_weight, key_function):
    taken, val = greedy(items, max_weight, key_function)
    print('Total value of items taken is', val)
    for item in taken:
        print(' ', item)

def test_greedys(max_weight = 20):
    items = build_items()
    print('Use greedy by value to fill knapsack of size', max_weight)
    test_greedy(items, max_weight, value)
    print('\nUse greedy by weight to fill knapsack of size',
          max_weight)
    test_greedy(items, max_weight, weight_inverse)
    print('\nUse greedy by density to fill knapsack of size',
          max_weight)
    test_greedy(items, max_weight, density)

```

[Figure 14-4](#) Using a greedy algorithm to choose items

When `test_greedys()` is executed, it prints

```

Use greedy by value to fill knapsack of size 20
Total value of items taken is 200.0
    <computer, 200, 20>
Use greedy by weight to fill knapsack of size 20
Total value of items taken is 170.0
    <book, 10, 1>
    <vase, 50, 2>
    <radio, 20, 4>
    <painting, 90, 9>
Use greedy by density to fill knapsack of size 20
Total value of items taken is 255.0
    <vase, 50, 2>
    <clock, 175, 10>
    <book, 10, 1>
    <radio, 20, 4>

```

### 14.1.2 An Optimal Solution to the 0/1 Knapsack Problem

Suppose we decide that an approximation is not good enough, i.e., we want the best possible solution to this problem. Such a solution is called **optimal**, not surprising since we are solving an optimization problem. As it happens, the problem confronting our burglar is an instance of a classic optimization problem, called the **0/1 knapsack problem**. The 0/1 knapsack problem can be formalized as follows:

- Each item is represented by a pair,  $\langle \text{value}, \text{weight} \rangle$ .
- The knapsack can accommodate items with a total weight of no more than  $w$ .
- A vector,  $I$ , of length  $n$ , represents the set of available items. Each element of the vector is an item.
- A vector,  $V$ , of length  $n$ , is used to indicate whether or not each item is taken by the burglar. If  $V[i] = 1$ , item  $I[i]$  is taken. If  $V[i] = 0$ , item  $I[i]$  is not taken.
- Find a  $V$  that maximizes

$$\sum_{i=0}^{n-1} V[i] * I[i].\text{value}$$

subject to the constraint that

$$\sum_{i=0}^{n-1} V[i] * I[i].\text{weight} \leq w$$

Let's see what happens if we try to implement this formulation of the problem in a straightforward way:

1. Enumerate all possible combinations of items. That is to say, generate all subsets of the set of items.<sup>90</sup> This is called the power set, and was discussed in Chapter 11.

2. Remove all of the combinations whose weight exceeds the allowed weight.
3. From the remaining combinations, choose any one whose value is the largest.

This approach will certainly find an optimal answer. However, if the original set of items is large, it will take a very long time to run, because, as we saw in Section 11.3.6, the number of subsets grows exceedingly quickly with the number of items.

[Figure 14-5](#) contains a straightforward implementation of this brute-force approach to solving the 0/1 knapsack problem. It uses the classes and functions defined in [Figure 14-2](#), [Figure 14-3](#), [Figure 14-4](#), and the function `gen_powerset` defined in Figure 11-6.

```

def choose_best(pset, max_weight, get_val, get_weight):
    best_val = 0.0
    best_set = None
    for items in pset:
        items_val = 0.0
        items_weight = 0.0
        for item in items:
            items_val += get_val(item)
            items_weight += get_weight(item)
        if items_weight <= max_weight and items_val > best_val:
            best_val = items_val
            best_set = items
    return (best_set, best_val)

def test_best(max_weight = 20):
    items = build_items()
    pset = gen_powerset(items)
    taken, val = choose_best(pset, max_weight, Item.get_value,
                            Item.get_weight)
    print('Total value of items taken is', val)
    for item in taken:
        print(item)

```

[Figure 14-5](#) Brute-force optimal solution to the 0/1 knapsack problem

The complexity of this implementation is order  $\theta(n * 2^n)$ , where  $n$  is the length of `items`. The function `gen_powerset` returns a list of lists

of `items`. This list is of length  $2^n$ , and the longest list in it is of length  $n$ . Therefore the outer loop in `choose_best` will be executed order  $\Theta(2^n)$  times, and the number of times the inner loop will be executed is bounded by  $n$ .

Many small optimizations can be applied to speed up this program. For example, `gen_powerset` could have had the header

```
def gen_powerset(items, constraint, get_val, get_weight)
```

and returned only those combinations that meet the weight constraint. Alternatively, `choose_best` could exit the inner loop as soon as the weight constraint is exceeded. While these kinds of optimizations are often worth doing, they don't address the fundamental issue. The complexity of `choose_best` will still be order  $\Theta(n^*2^n)$ , where  $n$  is the length of `items`, and `choose_best` will therefore still take a very long time to run when `items` is large.

In a theoretical sense, the problem is hopeless. The 0/1 knapsack problem is inherently exponential in the number of items. In a practical sense, however, the problem is far from hopeless, as we will discuss in Section 15.2.

When `test_best` is run, it prints

```
Total value of items taken is 275.0
<clock, 175, 10>
<painting, 90, 9>
<book, 10, 1>
```

Notice that this solution finds a combination of items with a higher total value than any of the solutions found by the greedy algorithms. The essence of a greedy algorithm is making the best (as defined by some metric) local choice at each step. It makes a choice that is **locally optimal**. However, as this example illustrates, a series of locally optimal decisions does not always lead to a solution that is **globally optimal**.

Despite the fact that they do not always find the best solution, greedy algorithms are often used in practice. They are usually easier to implement and more efficient to run than algorithms guaranteed to find optimal solutions. As Ivan Boesky once said, “I think greed is healthy. You can be greedy and still feel good about yourself.” [91](#)

For a variant of the knapsack problem, called the **fractional** (or **continuous**) **knapsack problem**, a greedy algorithm is guaranteed to find an optimal solution. Since the items in this variant are infinitely divisible, it always makes sense to take as much as possible of the item with the highest remaining value-to-weight ratio. Suppose, for example, that our burglar found only three things of value in the house: a sack of gold dust, a sack of silver dust, and a sack of raisins. In this case, a greedy-by-density algorithm will always find the optimal solution.

---

## 14.2 Graph Optimization Problems

Let's think about another kind of optimization problem. Suppose you had a list of the prices of all of the airline flights between each pair of cities in the United States. Suppose also that for all cities,  $A$ ,  $B$ , and  $C$ , the cost of flying from  $A$  to  $C$  by way of  $B$  was the cost of flying from  $A$  to  $B$  plus the cost of flying from  $B$  to  $C$ . A few questions you might like to ask are:

- What is the smallest number of stops between some pair of cities?
- What is the least expensive airfare between some pair of cities?
- What is the least expensive airfare between some pair of cities involving no more than two stops?
- What is the least expensive way to visit some collection of cities?

All of these problems (and many others) can be easily formalized as graph problems.

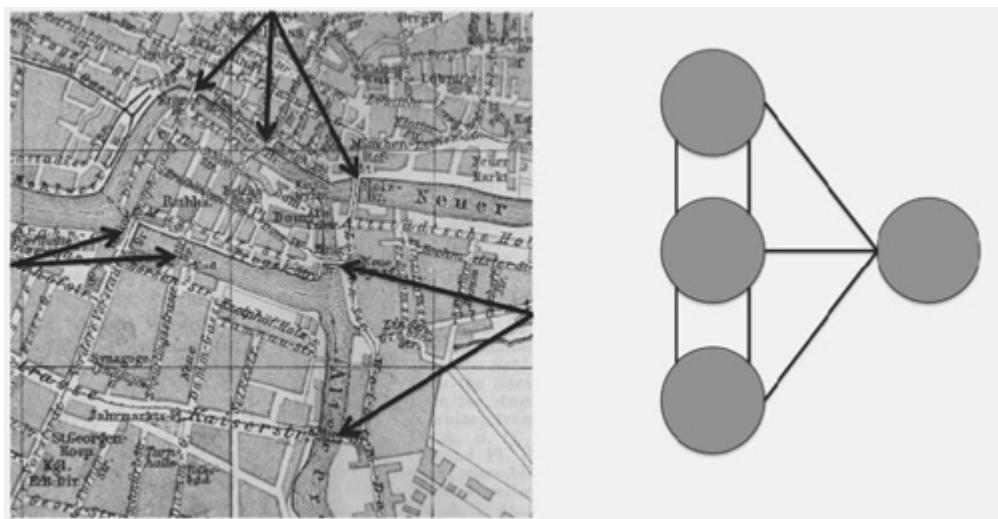
A **graph**<sup>92</sup> is a set of objects called **nodes** (or **vertices**) connected by a set of **edges** (or **arcs**). If the edges are unidirectional, the graph is called a **directed graph** or **digraph**. In a directed graph, if there is an edge from  $n_1$  to  $n_2$ , we refer to  $n_1$  as the **source** or **parent node** and  $n_2$  as the **destination** or **child node**.

A graph (or a digraph) is said to contain a **path** between two nodes,  $n_1$  and  $n_2$ , if there is a sequence of edges  $\langle e_0, \dots, e_n \rangle$  such that the source of  $e_0$  is  $n_1$ , the destination of  $e_n$  is  $n_2$ , and for all

edges  $e_1$  to  $e_n$  in the sequence, the source of  $e_i$  is the destination of  $e_{i-1}$ . A path from a node to itself is called a **cycle**. A graph containing a cycle is called **cyclic**, and a graph that contains no cycles is called **acyclic**.

Graphs are typically used to represent situations in which there are interesting relations among the parts. The first documented use of graphs in mathematics was in 1735 when the Swiss mathematician Leonhard Euler used what has come to be known as **graph theory** to formulate and solve the **Königsberg bridges problem**.

Königsberg, then the capital of East Prussia, was built at the intersection of two rivers that contained a number of islands. The islands were connected to each other and to the mainland by seven bridges, as shown on the map on the left side of [Figure 14-6](#). For some reason, the residents of the city were obsessed with the question of whether it was possible to take a walk that crossed each bridge exactly once.



[Figure 14-6](#) The bridges of Königsberg (left) and Euler's simplified map (right)

Euler's great insight was that the problem could be vastly simplified by viewing each separate landmass as a point (think "node") and each bridge as a line (think "edge") connecting two of these points. The map of the town could then be represented by the undirected graph to the right of the map in [Figure 14-6](#). Euler then

reasoned that if a walk were to traverse each edge exactly once, it must be the case that each node in the middle of the walk (i.e., any island that is both entered and exited during the walk) must be connected by an even number of edges. Since none of the nodes in this graph has an even number of edges, Euler concluded that it is impossible to traverse each bridge exactly once.

Of greater interest than the Königsberg bridges problem, or even Euler's theorem (which generalizes his solution to the Königsberg bridges problem), is the whole idea of using graph theory to help understand problems.

For example, only one small extension to the kind of graph used by Euler is needed to model a country's highway system. If a weight is associated with each edge in a graph (or digraph), it is called a **weighted graph**. Using weighted graphs, the highway system can be represented as a graph in which cities are represented by nodes and the highways connecting them as edges, where each edge is labeled with the distance between the two nodes. More generally, we can represent any road map (including those with one-way streets) by a weighted digraph.

Similarly, the structure of the World Wide Web can be represented as a digraph in which the nodes are webpages with an edge from node  $A$  to node  $B$  if and only if there is a link to page  $B$  on page  $A$ . Traffic patterns could be modeled by adding a weight to each edge indicating how often it is used.

There are also many less obvious uses of graphs. Biologists use graphs to model things ranging from the way proteins interact with each other to gene expression networks. Physicists use graphs to describe phase transitions. Epidemiologists use graphs to model disease trajectories. And so on.

[Figure 14-7](#) contains classes implementing abstract types corresponding to nodes, weighted edges, and edges.

```

class Node(object):
    def __init__(self, name):
        """Assumes name is a string"""
        self._name = name
    def get_name(self):
        return self._name
    def __str__(self):
        return self._name

class Edge(object):
    def __init__(self, src, dest):
        """Assumes src and dest are nodes"""
        self._src = src
        self._dest = dest
    def get_source(self):
        return self._src
    def get_destination(self):
        return self._dest
    def __str__(self):
        return self._src.get_name() + ' -> ' + self._dest.get_name()

class Weighted_edge(Edge):
    def __init__(self, src, dest, weight = 1.0):
        """Assumes src and dest are nodes, weight a number"""
        self._src = src
        self._dest = dest
        self._weight = weight
    def get_weight(self):
        return self._weight
    def __str__(self):
        return f'{self._src.get_name()} -> ({self._weight})' +
            f'{self._dest.get_name()}'
```

[Figure 14-7](#) Nodes and edges

Having a class for nodes may seem like overkill. After all, none of the methods in class `Node` perform any interesting computation. We introduced the class merely to give us the flexibility of deciding, perhaps at some later point, to introduce a subclass of `Node` with additional properties.

[Figure 14-8](#) contains implementations of the classes `Digraph` and `Graph`.

```

class Digraph(object):
    #nodes is a list of the nodes in the graph
    #edges is a dict mapping each node to a list of its children
    def __init__(self):
        self._nodes = []
        self._edges = {}
    def add_node(self, node):
        if node in self._nodes:
            raise ValueError('Duplicate node')
        else:
            self._nodes.append(node)
            self._edges[node] = []
    def add_edge(self, edge):
        src = edge.get_source()
        dest = edge.get_destination()
        if not (src in self._nodes and dest in self._nodes):
            raise ValueError('Node not in graph')
        self._edges[src].append(dest)
    def children_of(self, node):
        return self._edges[node]
    def has_node(self, node):
        return node in self._nodes
    def __str__(self):
        result = ''
        for src in self._nodes:
            for dest in self._edges[src]:
                result = (result + src.get_name() + '->'
                          + dest.get_name() + '\n')
        return result[:-1] #omit final newline

class Graph(Digraph):
    def add_edge(self, edge):
        Digraph.add_edge(self, edge)
        rev = Edge(edge.get_destination(), edge.get_source())
        Digraph.add_edge(self, rev)

```

[Figure 14-8](#) Classes `Graph` and `Digraph`

One important decision is the choice of data structure used to represent a `Digraph`. One common representation is an  $n \times n$  **adjacency matrix**, where  $n$  is the number of nodes in the graph. Each cell of the matrix contains information (e.g., weights) about the edges connecting the pair of nodes  $\langle i, j \rangle$ . If the edges are unweighted, each entry is `True` if and only if there is an edge from  $i$  to  $j$ .

Another common representation is an **adjacency list**, which we use here. Class `Digraph` has two instance variables. The variable `nodes` is a Python list containing the names of the nodes in the `Digraph`. The connectivity of the nodes is represented using an adjacency list implemented as a dictionary. The variable `edges` is a dictionary that maps each `Node` in the `Digraph` to a list of the children of that `Node`.

Class `Graph` is a subclass of `Digraph`. It inherits all of the methods of `Digraph` except `add_edge`, which it overrides. (This is not the most space-efficient way to implement `Graph`, since it stores each edge twice, once for each direction in the `Digraph`. But it has the virtue of simplicity.)

You might want to stop for a minute and think about why `Graph` is a subclass of `Digraph`, rather than the other way around. In many of the examples of subclassing we have looked at, the subclass adds attributes to the superclass. For example, class `Weighted_edge` added a `weight` attribute to class `Edge`.

Here, `Digraph` and `Graph` have the same attributes. The only difference is the implementation of the `add_edge` method. Either could have been easily implemented by inheriting methods from the other, but the choice of which to make the superclass was not arbitrary. In Chapter 10, we stressed the importance of obeying the substitution principle: 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.

And indeed if client code works correctly using an instance of `Digraph`, it will work correctly if an instance of `Graph` is substituted for the instance of `Digraph`. The converse is not true. There are many algorithms that work on graphs (by exploiting the symmetry of edges) that do not work on directed graphs.

### 14.2.1 Some Classic Graph-Theoretic Problems

One of the nice things about formulating a problem using graph theory is that there are well-known algorithms for solving many optimization problems on graphs. Some of the best-known graph optimization problems are:

- **Shortest path.** For some pair of nodes,  $n_1$  and  $n_2$ , find the shortest sequence of edges  $\langle s_n, d_n \rangle$  (source node and destination node), such that
  - The source node in the first edge is  $n_1$ .
  - The destination node of the last edge is  $n_2$ .
  - For all edges  $e_1$  and  $e_2$  in the sequence, if  $e_2$  follows  $e_1$  in the sequence, the source node of  $e_2$  is the destination node of  $e_1$ .
- **Shortest weighted path.** This is like the shortest path, except instead of choosing the shortest sequence of edges that connects two nodes, we define some function on the weights of the edges in the sequence (e.g., their sum) and minimize that value. This is the kind of problem solved by Google and Apple Maps when asked to compute driving directions between two points.
- **Min cut.** Given two sets of nodes in a graph, a **cut** is a set of edges whose removal eliminates all paths from each node in one set to each node in the other.
- **Maximum clique.** A **clique** is a set of nodes such that there is an edge between each pair of nodes in the set.<sup>93</sup> A maximum clique is a clique of the largest size in a graph. The minimum cut is the smallest set of edges whose removal accomplishes this.

#### 14.2.2 Shortest Path: Depth-First Search and Breadth-First Search

Social networks are made up of individuals and relationships between individuals. These are typically modeled as graphs in which the individuals are nodes and the edges relationships. If the relationships are symmetric, the edges are undirected; if the relationships are asymmetric, the edges are directed. Some social networks model multiple kinds of relationships, in which case labels on the edges indicate the kind of relationship.

In 1990 the playwright John Guare wrote *Six Degrees of Separation*. The dubious premise underlying the play is that “everybody on this planet is separated by only six other people.” By this he meant that if we built a social network including every person

on the Earth using the relation “knows,” the shortest path between any two individuals would pass through at most six other nodes.

A less hypothetical question is the distance using the “friend” relation between pairs of people on Facebook. For example, you might wonder if you have a friend who has a friend who has a friend who is a friend of Lady Gaga. Let’s think about designing a program to answer such questions.

The friend relation (at least on Facebook) is symmetric, e.g., if Sam is a friend of Andrea, Andrea is a friend of Sam. We will, therefore, implement the social network using type `Graph`. We can then define the problem of finding the shortest connection between you and Lady Gaga as:

- Let  $G$  be the graph representing the friend relation.
- For  $G$ , find the shortest sequence of nodes, [You, ..., Lady Gaga], such that
- If  $n_i$  and  $n_{i+1}$  are consecutive nodes in the sequence, there is an edge in  $G$  connecting  $n_i$  and  $n_{i+1}$ .

[Figure 14-9](#) contains a recursive function that finds the shortest path between two nodes, `start` and `end`, in a `Digraph`. Since `Graph` is a subclass of `Digraph`, it will work for our Facebook problem.

```

def print_path(path):
    """Assumes path is a list of nodes"""
    result = ''
    for i in range(len(path)):
        result = result + str(path[i])
        if i != len(path) - 1:
            result = result + '->'
    return result

def DFS(graph, start, end, path, shortest, to_print = False):
    """Assumes graph is a Digraph; start and end are nodes;
       path and shortest are lists of nodes
       Returns a shortest path from start to end in graph"""
    path = path + [start]
    if to_print:
        print('Current DFS path:', print_path(path))
    if start == end:
        return path
    for node in graph.children_of(start):
        if node not in path: #avoid cycles
            if shortest == None or len(path) < len(shortest):
                new_path = DFS(graph, node, end, path, shortest,
                               to_print)
                if new_path != None:
                    shortest = new_path
    return shortest

def shortest_path(graph, start, end, to_print = False):
    """Assumes graph is a Digraph; start and end are nodes
       Returns a shortest path from start to end in graph"""
    return DFS(graph, start, end, [], None, to_print)

```

[Figure 14-9](#) Depth-first-search shortest-path algorithm

The algorithm implemented by `DFS` is an example of a recursive **depth-first-search (DFS)** algorithm. In general, a depth-first-search algorithm begins by choosing one child of the start node. It then chooses one child of that node and so on, going deeper and deeper until it either reaches the goal node or a node with no children. The search then **backtracks**, returning to the most recent node with children that it has not yet visited. When all paths have been explored, it chooses the shortest path (assuming that there is one) from the start to the goal.

The code is more complicated than the algorithm we just described because it has to deal with the possibility of the graph

containing cycles. It also avoids exploring paths longer than the shortest path that it has already found.

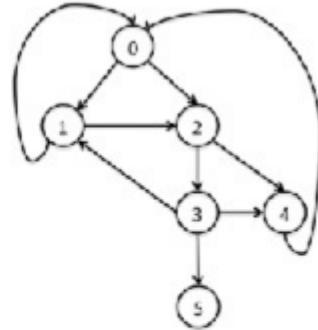
- The function `shortest_path` calls `DFS` with `path == []` (to indicate that the current path being explored is empty) and `shortest == None` (to indicate that no path from `start` to `end` has yet been found).
- `DFS` begins by choosing one child of `start`. It then chooses one child of that node and so on, until either it reaches the node `end` or a node with no unvisited children.
  - The check `if node not in path` prevents the program from getting caught in a cycle.
  - The check `if shortest == None or len(path) < len(shortest)` is used to decide if it is possible that continuing to search this path might yield a shorter path than the best path found so far.
  - If so, `DFS` is called recursively. If it finds a path to `end` that is no longer than the best found so far, `shortest` is updated.
  - When the last node on `path` has no children left to visit, the program backtracks to the previously visited node and visits the next child of that node.
- The function returns when all possible shortest paths from `start` to `end` have been explored.

[Figure 14-10](#) contains some code that runs the code in [Figure 14-9](#). The function `test_SP` in [Figure 14-10](#) first builds a directed graph like the one pictured in the figure, and then searches for a shortest path between node 0 and node 5.

```

def test_SP():
    nodes = []
    for name in range(6): #Create 6 nodes
        nodes.append(Node(str(name)))
    g = Digraph()
    for n in nodes:
        g.add_node(n)
    g.add_edge(Edge(nodes[0],nodes[1]))
    g.add_edge(Edge(nodes[1],nodes[2]))
    g.add_edge(Edge(nodes[2],nodes[3]))
    g.add_edge(Edge(nodes[2],nodes[4]))
    g.add_edge(Edge(nodes[3],nodes[4]))
    g.add_edge(Edge(nodes[3],nodes[5]))
    g.add_edge(Edge(nodes[0],nodes[2]))
    g.add_edge(Edge(nodes[1],nodes[0]))
    g.add_edge(Edge(nodes[3],nodes[1]))
    g.add_edge(Edge(nodes[4],nodes[0]))
    sp = shortest_path(g, nodes[0], nodes[5], to_print = True)
    print('Shortest path found by DFS:', print_path(sp))

```



[Figure 14-10](#) Test depth-first-search code

When executed, `test_SP` produces the output

```

Current DFS path: 0
Current DFS path: 0->1
Current DFS path: 0->1->2
Current DFS path: 0->1->2->3
Current DFS path: 0->1->2->3->4
Current DFS path: 0->1->2->3->5
Current DFS path: 0->1->2->4
Current DFS path: 0->2
Current DFS path: 0->2->3
Current DFS path: 0->2->3->4
Current DFS path: 0->2->3->5
Current DFS path: 0->2->3->1
Current DFS path: 0->2->4
Shortest path found by DFS: 0->2->3->5

```

Notice that after exploring the path  $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4$ , it backs up to node 3 and explores the path  $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 5$ . After saving that as the shortest successful path so far, it backs up to node 2 and explores the path  $0 \rightarrow 1 \rightarrow 2 \rightarrow 4$ . When it reaches the end of that path (node 4), it backs up all the way to node 0 and investigates the path starting with the edge from 0 to 2. And so on.

The DFS algorithm implemented in [Figure 14-9](#) finds the path with the minimum number of edges. If the edges have weights, it will not necessarily find the path that minimizes the sum of the weights of the edges. However, it is easily modified to do so.

**Finger exercise:** Modify the DFS algorithm to find a path that minimizes the sum of the weights. Assume that all weights are positive integers.

Of course, there are other ways to traverse a graph than depth-first. Another common approach is breadth-first search (BFS). A breadth-first traversal first visits all children of the start node. If none of those is the end node, it visits all children of each of those nodes. And so on. Unlike depth-first search, which is often implemented recursively, breadth-first search is usually implemented iteratively. BFS explores many paths simultaneously, adding one node to each path on each iteration. Since it generates the paths in ascending order of length, the first path found with the goal as its last node is guaranteed to have a minimum number of edges.

[Figure 14-11](#) contains code that uses a breadth-first search to find the shortest path in a directed graph. The variable `path_queue` is used to store all of the paths currently being explored. Each iteration starts by removing a path from `path_queue` and assigning that path to `tmp_path`. If the last node in `tmp_path` is `end`, `tmp_path` is a shortest path and is returned. Otherwise, a set of new paths is created, each of which extends `tmp_path` by adding one of its children. Each of these new paths is then added to `path_queue`.

```

def BFS(graph, start, end, to_print = False):
    """Assumes graph is a Digraph; start and end are nodes
       Returns a shortest path from start to end in graph"""
    init_path = [start]
    path_queue = [init_path]
    while len(path_queue) != 0:
        #Get and remove oldest element in path_queue
        tmp_path = path_queue.pop(0)
        if to_print:
            print('Current BFS path:', print_path(tmp_path))
        last_node = tmp_path[-1]
        if last_node == end:
            return tmp_path
        for next_node in graph.children_of(last_node):
            if next_node not in tmp_path:
                new_path = tmp_path + [next_node]
                path_queue.append(new_path)
    return None

```

[Figure 14-11](#) Breadth-first-search shortest path algorithm

## When the lines

```

sp = BFS(g, nodes[0], nodes[5])
print('Shortest path found by BFS:', print_path(sp))

```

are added at the end of `test_SP` and the function is executed, it prints the additional lines

```

Current BFS path: 0
Current BFS path: 0->1
Current BFS path: 0->2
Current BFS path: 0->1->2
Current BFS path: 0->2->3
Current BFS path: 0->2->4
Current BFS path: 0->1->2->3
Current BFS path: 0->1->2->4
Current BFS path: 0->2->3->4
Current BFS path: 0->2->3->5
Shortest path found by BFS: 0->2->3->5

```

Comfortingly, each algorithm found a path of the same length. In this case, they found the same path. However, if a graph contains more than one shortest path between a pair of nodes, DFS and BFS will not necessarily find the same shortest path.

As mentioned above, BFS is a convenient way to search for a path with the fewest edges because the first time a path is found, it is guaranteed to be such a path.

**Finger exercise:** Consider a digraph with weighted edges. Is the first path found by BFS guaranteed to minimize the sum of the weights of the edges?

---

### 14.3 Terms Introduced in Chapter

optimization problem

objective function

set of constraints

knapsack problem

greedy algorithm

optimal solution

0/1 knapsack problem

locally optimal

globally optimal

continuous knapsack problem

graph

node (vertex)

edge (arc)

directed graph (digraph)

source (parent) node

destination (child) node

path

cycle

cyclic graph

acyclic graph  
graph theory  
weighted graph  
adjacency matrix  
adjacency list  
shortest path  
shortest weighted path  
min cut  
maximum clique  
depth-first search (DFS)  
backtracking  
breadth-first search (BFS)

---

- 87 For those of you too young to remember, a “knapsack” is a simple bag that people used to carry on their back—long before “backpacks” became fashionable. If you happen to have been in scouting you might remember the words of the “Happy Wanderer,” “I love to go a-wandering, Along the mountain track, And as I go, I love to sing, My knapsack on my back.”
- 88 There is probably some deep moral lesson to be extracted from this fact, and it is probably not “greed is good.”
- 89 As we discussed in Chapter 10, the time complexity of the sorting algorithm, timsort, used in most Python implementations is  $O(n \log n)$ .
- 90 Recall that every set is a subset of itself and the empty set is a subset of every set.
- 91 He said this, to enthusiastic applause, in a 1986 commencement address at the University of California at Berkeley Business

School. A few months later he was indicted for insider trading, a charge that led to two years in prison and a \$100,000,000 fine.

- 92** Computer scientists and mathematicians use the word “graph” in the sense used in this book. They typically use the word “plot” or “chart” to denote pictorial representations of information.
- 93** This notion is quite similar to the notion of a social clique, i.e., a group of people who feel closely connected to each other and are inclined to exclude those not in the clique.

# 15

## DYNAMIC PROGRAMMING

**Dynamic programming** was invented by Richard Bellman in the 1950s. Don't try to infer anything about the technique from its name. As Bellman described it, the name "dynamic programming" was chosen to hide from governmental sponsors "the fact that I was really doing mathematics... [the phrase dynamic programming] was something not even a Congressman could object to."<sup>94</sup>

Dynamic programming is a method for efficiently solving problems that exhibit the characteristics of overlapping subproblems and optimal substructure. Fortunately, many optimization problems exhibit these characteristics.

A problem has **optimal substructure** if a globally optimal solution can be found by combining optimal solutions to local subproblems. We've already looked at a number of such problems. Merge sort, for example, exploits the fact that a list can be sorted by first sorting sublists and then merging the solutions.

A problem has **overlapping subproblems** if an optimal solution involves solving the same problem multiple times. Merge sort does not exhibit this property. Even though we are performing a merge many times, we are merging different lists each time.

It's not immediately obvious, but the 0/1 knapsack problem exhibits both of these properties. First, however, we digress to look at a problem where the optimal substructure and overlapping subproblems are more obvious.

---

### 15.1 Fibonacci Sequences, Revisited

In Chapter 4, we looked at a straightforward recursive implementation of the Fibonacci function:

```

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

```

While this implementation of the recurrence is obviously correct, it is terribly inefficient. Try, for example, running `fib(120)`, but don't wait for it to complete. The complexity of the implementation is a bit hard to derive, but it is roughly  $O(fib(n))$ . That is, its growth is proportional to the growth in the value of the result, and the growth rate of the Fibonacci sequence is substantial. For example, `fib(120)` is 8,670,007,398,507,948,658,051,921. If each recursive call took a nanosecond, `fib(120)` would take about 250,000 years to finish.

Let's try and figure out why this implementation takes so long. Given the tiny amount of code in the body of `fib`, it's clear that the problem must be the number of times that `fib` calls itself. As an example, look at the tree of calls associated with the invocation `fib(6)`.

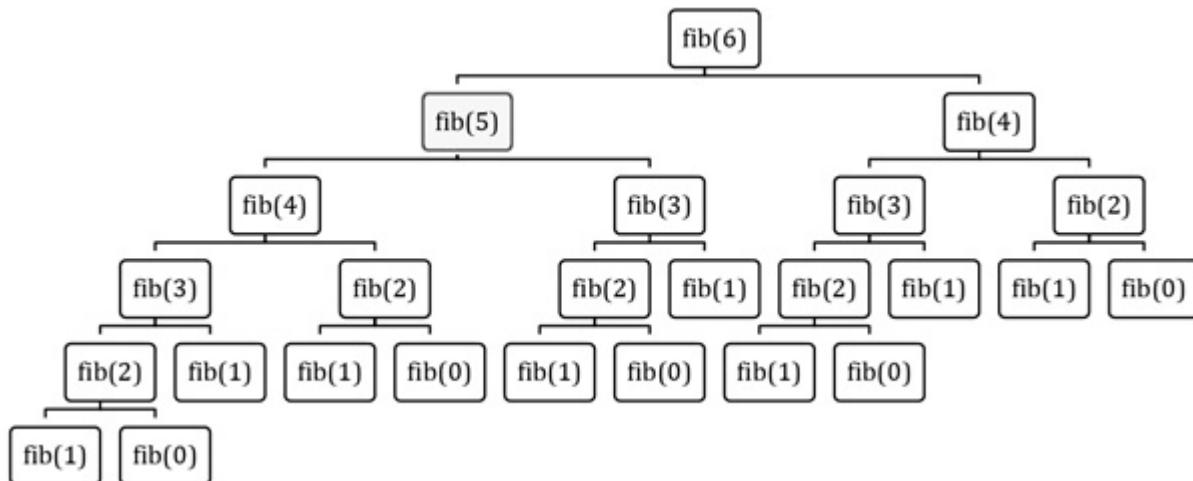


Figure 15-1 Tree of calls for recursive Fibonacci

Notice that we are computing the same values over and over again. For example, `fib` gets called with 3 three times, and each of these calls provokes four additional calls of `fib`. It doesn't require a

genius to think that it might be a good idea to record the value returned by the first call, and then look it up rather than compute it each time it is needed. This is the key idea behind dynamic programming.

There are two approaches to dynamic programming

- **Memorization** solves the original problem top-down. It starts from the original problem, breaks it into subproblems, breaks the subproblems into subproblems, etc. Each time it solves a subproblem, it stores the answer in a table. Each time it needs to solve a subproblem, it first tries to look up the answer in the table.
- **Tabular** is a bottom-up method. It starts from the smallest problems, and stores the answers to those in a table. It then combines the solutions to these problems to solve the next smallest problems, and stores those answers in the table.

[Figure 15-2](#) contains implementations of Fibonacci using each approach to dynamic programming. The function `fib_memo` has a parameter, `memo`, that it uses to keep track of the numbers it has already evaluated. The parameter has a default value, the empty dictionary, so that clients of `fib_memo` don't have to worry about supplying an initial value for `memo`. When `fib_memo` is called with an  $n > 1$ , it attempts to look up  $n$  in `memo`. If it is not there (because this is the first time `fib_memo` has been called with that value), an exception is raised. When this happens, `fib_memo` uses the normal Fibonacci recurrence, and then stores the result in `memo`.

The function `fib_tab` is quite simple. It exploits the fact that all of the subproblems for Fibonacci are known in advance and easy to enumerate in a useful order.

```

def fib_memo(n, memo = None):
    """Assumes n is an int >= 0, memo used only by recursive calls
       Returns Fibonacci of n"""
    if memo == None:
        memo = {}
    if n == 0 or n == 1:
        return 1
    try:
        return memo[n]
    except KeyError:
        result = fib_memo(n-1, memo) + fib_memo(n-2, memo)
        memo[n] = result
        return result

def fib_tab(n):
    """Assumes n is an int >= 0
       Returns Fibonacci of n"""
    tab = [1]*(n+1) #only first two values matter
    for i in range(2, n + 1):
        tab[i] = tab[i-1] + tab[i-2]
    return tab[n]

```

[Figure 15-2](#) Implementing Fibonacci using a memo

If you try running `fib_memo` and `fib_tab`, you will see that they are indeed quite fast: `fib(120)` returns almost instantly. What is the complexity of these functions? `fib_memo` calls `fib` exactly once for each value from 0 to `n`. Therefore, under the assumption that dictionary lookup can be done in constant time, the time complexity of `fib_memo(n)` is in  $O(n)$ . `fib_tab` is even more obviously in  $O(n)$ .

If solving the original problem requires solving all subproblems, it is usually better to use the tabular approach. It is simpler to program, and faster because it doesn't have the overhead associated with recursive calls and can pre-allocate a table of the appropriate size rather than growing a memo. If only some of the subproblems need to be solved (which is often the case), memoization is typically more efficient.

**Finger exercise:** Use the tabular method to implement a dynamic programming solution that meets the specification

```

def make_change(coin_vals, change):
    """coin_vals is a list of positive ints and coin_vals[0]

```

```

= 1
    change is a positive int,
    return the minimum number of coins needed to have a
set of
    coins the values of which sum to change. Coins may
be used
    more than once. For example, make_change([1, 5,
8], 11)
    should return 3.""""

```

---

## 15.2 Dynamic Programming and the 0/1 Knapsack Problem

One of the optimization problems we looked at in Chapter 14 was the 0/1 knapsack problem. Recall that we looked at a greedy algorithm that ran in  $n \log n$  time, but was not guaranteed to find an optimal solution. We also looked at a brute-force algorithm that was guaranteed to find an optimal solution, but ran in exponential time. Finally, we discussed the fact that the problem is inherently exponential in the size of the input. In the worst case, one cannot find an optimal solution without looking at all possible answers.

Fortunately, the situation is not as bad as it seems. Dynamic programming provides a practical method for solving most 0/1 knapsack problems in a reasonable amount of time. As a first step in deriving such a solution, we begin with an exponential solution based on exhaustive enumeration. The key idea is to think about exploring the space of possible solutions by constructing a rooted binary tree that enumerates all states satisfying the weight constraint.

A **rooted binary tree** is an acyclic directed graph in which

- There is exactly one node with no parents. This is called the **root**.
- Each non-root node has exactly one parent.
- Each node has at most two children. A childless node is called a **leaf**.

Each node in the search tree for the 0/1 knapsack problem is labeled with a quadruple that denotes a partial solution to the knapsack problem. The elements of the quadruple are:

- A set of items to be taken
- The list of items for which a decision has not been made
- The total value of the items in the set of items to be taken (this is merely an optimization, since the value could be computed from the set)
- The remaining space in the knapsack. (Again, this is an optimization, since it is merely the difference between the weight allowed and the weight of all the items taken so far.)

The tree is built top-down starting with the root.<sup>95</sup> One element is selected from the still-to-be-considered items. If there is room for that item in the knapsack, a node is constructed that reflects the consequence of choosing to take that item. By convention, we draw that node as the left child. The right child shows the consequences of choosing not to take that item. The process is then applied recursively until either the knapsack is full or there are no more items to consider. Because each edge represents a decision (to take or not to take an item), such trees are called **decision trees**.<sup>96</sup>

[Figure 15-3](#) is a table describing a set of items.

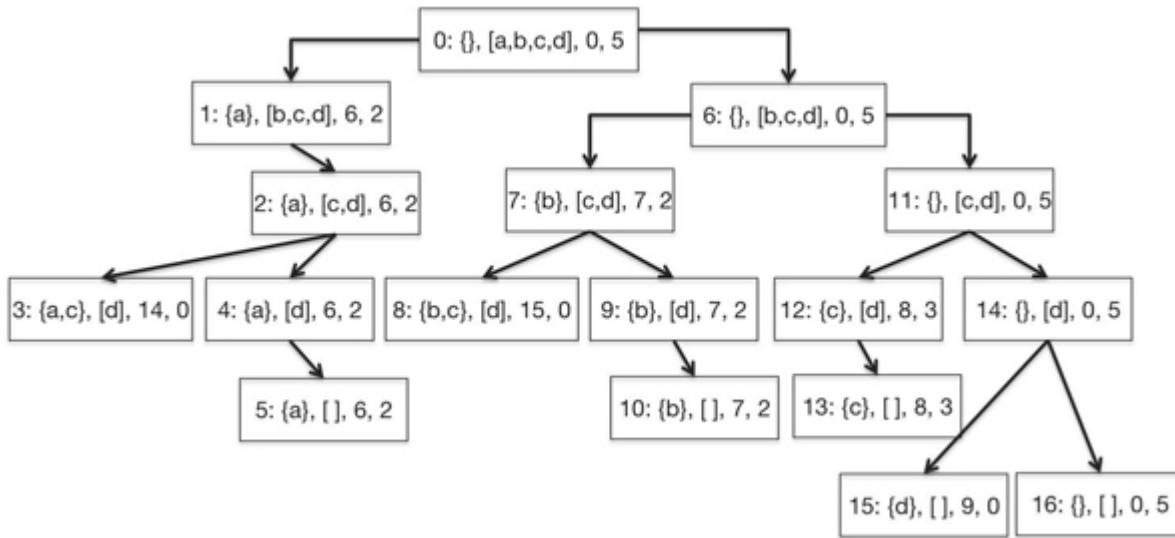
Name	Value	Weight
a	6	3
b	7	3
c	8	2
d	9	5

[Figure 15-3](#) Table of items with values and weights

[Figure 15-4](#) is a decision tree for deciding which of those items to take under the assumption that the knapsack has a maximum weight of 5. The root of the tree (node 0) has a label  $\langle \{ \}, [a, b, c, d], 0, 5 \rangle$ , indicating that no items have been taken, all items remain to be considered, the value of the items taken is 0, and a weight of 5 is still available. Node 1 indicates that item  $a$  has been taken,  $[b, c, d]$  remain to be considered, the value of the items taken is 6, and the

knapsack can hold another 2 pounds. Node 1 has no left child since item b, which weighs 3 pounds, would not fit in the knapsack.

In [Figure 15-4](#), the numbers that precede the colon in each node indicate one order in which the nodes could be generated. This particular ordering is called left-first depth-first. At each node, we attempt to generate a left node. If that is impossible, we attempt to generate a right node. If that too is impossible, we back up one node (to the parent) and repeat the process. Eventually, we find ourselves having generated all descendants of the root, and the process halts. When it does, each combination of items that could fit in the knapsack has been generated, and any leaf node with the greatest value represents an optimal solution. Notice that for each leaf node, either the second element is the empty list (indicating that there are no more items to consider taking) or the fourth element is 0 (indicating that there is no room left in the knapsack).



[Figure 15-4](#). Decision tree for knapsack problem

Unsurprisingly (especially if you read Chapter 14), the natural implementation of a depth-first tree search is recursive. [Figure 15-5](#) contains such an implementation. It uses class `Item` and the functions defined in [Figure 14-2](#).

The function `max_val` returns two values, the set of items chosen and the total value of those items. It is called with two arguments,

corresponding to the second and fourth elements of the labels of the nodes in the tree:

- `to_consider`. Those items that nodes higher up in the tree (corresponding to earlier calls in the recursive call stack) have not yet considered.
- `avail`. The amount of space still available.

Notice that the implementation of `max_val` does not build the decision tree and then look for an optimal node. Instead, it uses the local variable `result` to record the best solution found so far. The code in [Figure 15-6](#) can be used to test `max_val`.

When `small_test` (which uses the values in [Figure 15-3](#)) is run it prints a result indicating that node 8 in [Figure 15-4](#) is an optimal solution:

```
<c, 8, 2>
<b, 7, 3>
Total value of items taken = 15
```

```

def max_val(to_consider, avail):
    """Assumes to_consider a list of items, avail a weight
       Returns a tuple of the total value of a solution to the
       0/1 knapsack problem and the items of that solution"""
    if to_consider == [] or avail == 0:
        result = (0, ())
    elif to_consider[0].get_weight() > avail:
        #Explore right branch only
        result = max_val(to_consider[1:], avail)
    else:
        next_item = to_consider[0]
        #Explore left branch
        with_val, with_to_take = max_val(to_consider[1:],
                                         avail - next_item.get_weight())
        with_val += next_item.get_value()
        #Explore right branch
        without_val, without_to_take = max_val(to_consider[1:], avail)
        #Choose better branch
        if with_val > without_val:
            result = (with_val, with_to_take + (next_item,))
        else:
            result = (without_val, without_to_take)
    return result

```

[Figure 15-5](#) Using a decision tree to solve a knapsack problem

The functions `build_many_items` and `big_test` can be used to test `max_val` on randomly generated sets of items. Try `big_test(10, 40)`. That didn't take long. Now try `big_test(40, 100)`. After you get tired of waiting for it to return, stop the computation and ask yourself what is going on.

Let's think about the size of the tree we are exploring. Since at each level of the tree we are deciding to keep or not keep one item, the maximum depth of the tree is `len(items)`. At level 0 we have only one node, at level 1 up to two nodes, at level 2 up to four nodes, and at level 3 up to eight nodes. At level 39 we have up to  $2^{39}$  nodes. No wonder it takes a long time to run!

Let's see if dynamic programming can help.

Optimal substructure is visible both in [Figure 15-4](#) and in [Figure 15-5](#). Each parent node combines the solutions reached by its children to derive an optimal solution for the subtree rooted at that

parent. This is reflected in [Figure 15-5](#) by the code following the comment `#Choose better branch.`

```
def small_test():
    names = ['a', 'b', 'c', 'd']
    vals = [6, 7, 8, 9]
    weights = [3, 3, 2, 5]
    Items = []
    for i in range(len(vals)):
        Items.append(Item(names[i], vals[i], weights[i]))
    val, taken = max_val(Items, 5)
    for item in taken:
        print(item)
    print('Total value of items taken =', val)

def build_many_items(num_items, max_val, max_weight):
    items = []
    for i in range(num_items):
        items.append(Item(str(i),
                          random.randint(1, max_val),
                          random.randint(1, max_weight)))
    return items

def big_test(num_items, avail_weight):
    items = build_many_items(num_items, 10, 10)
    val, taken = max_val(items, avail_weight)
    print('Items Taken')
    for item in taken:
        print(item)
    print('Total value of items taken =', val)
```

[Figure 15-6](#) Testing the decision tree-based implementation

Are there also overlapping subproblems? At first glance, the answer seems to be “no.” At each level of the tree we have a different set of available items to consider. This implies that if common subproblems do exist, they must be at the same level of the tree. And indeed, at each level of the tree, each node has the same set of items to consider taking. However, we can see by looking at the labels in [Figure 15-4](#) that each node at a level represents a different set of choices about the items considered higher in the tree.

Think about what problem is being solved at each node: finding the optimal items to take from those left to consider, given the

remaining available weight. The available weight depends upon the total weight of the items taken so far, but not on which items are taken or the total value of the items taken. So, for example, in [Figure 15-4](#), nodes 2 and 7 are actually solving the same problem: deciding which elements of [c,d] should be taken, given that the available weight is 2.

The code in [Figure 15-7](#) exploits the optimal substructure and overlapping subproblems to provide a memorization-based dynamic programming solution to the 0/1 knapsack problem.

```
def fast_max_val(to_consider, avail, memo = {}):
    """Assumes to_consider a list of items, avail a weight
       memo supplied by recursive calls
       Returns a tuple of the total value of a solution to the
       0/1 knapsack problem and the items of that solution"""
    if (len(to_consider), avail) in memo:
        result = memo[(len(to_consider), avail)]
    elif to_consider == [] or avail == 0:
        result = (0, ())
    elif to_consider[0].get_weight() > avail:
        #Explore right branch only
        result = fast_max_val(to_consider[1:], avail, memo)
    else:
        next_item = to_consider[0]
        #Explore left branch
        with_val, with_to_take = \
            fast_max_val(to_consider[1:],
                         avail - next_item.get_weight(), memo)
        with_val += next_item.get_value()
        #Explore right branch
        without_val, without_to_take = fast_max_val(to_consider[1:],
                                                      avail, memo)
        #Choose better branch
        if with_val > without_val:
            result = (with_val, with_to_take + (next_item,))
        else:
            result = (without_val, without_to_take)
    memo[(len(to_consider), avail)] = result
    return result
```

[Figure 15-7](#) Dynamic programming solution to knapsack problem

An extra parameter, `memo`, has been added to keep track of solutions to subproblems that have already been solved. It is

implemented using a dictionary with a key constructed from the length of `to_consider` and the available weight. The expression `len(to_consider)` is a compact way of representing the items still to be considered. This works because items are always removed from the same end (the front) of the list `to_consider`.

Now, replace the call to `max_val` by a call to `fast_max_val` in `big_test`, and try running `big_test(40, 100)`. It returns almost instantly with an optimal solution to the problem.

[Figure 15-8](#) shows the number of calls made when we ran the code on problems with a varying number of items and a maximum weight of 100. The growth is hard to quantify, but it is clearly far less than exponential.<sup>97</sup> But how can this be, since we know that the 0/1 knapsack problem is inherently exponential in the number of items? Have we found a way to overturn fundamental laws of the universe? No, but we have discovered that computational complexity can be a subtle notion.<sup>98</sup>

<code>len(items)</code>	Number of items selected	$2^{**(\text{len(items)})}$	Number of calls
4	4	16	31
8	8	256	171
16	16	65,536	1,085
32	23	4,294,967,296	2,935
64	29	18,446,744,073,709,551,616	6,258
128	42	340,282,366,920,938,463,463,374,607, 431,768,211,456	12,055
256	54	115,792,089,237,316,195,423,570,985, 008,687,907,853,269,984,665,640,564, 039,457,584,007,913,129,639,936	25,474
512	69	A really really big number	50,3056
1024	83	A humongous number	100,231

[Figure 15-8](#) Performance of dynamic programming solution

The running time of `fast_max_val` is governed by the number of distinct pairs, `<to_consider, avail>`, generated. This is because the decision about what to do next depends only upon the items still available and the total weight of the items already taken.

The number of possible values of `to_consider` is bounded by `len(items)`. The number of possible values of `avail` is more difficult to characterize. It is bounded from above by the maximum number of distinct totals of weights of the items that the knapsack can hold. If the knapsack can hold at most  $n$  items (based on the capacity of the knapsack and the weights of the available items), `avail` can take on at most  $2^n$  different values. In principle, this could be a rather large number. However, in practice, it is not usually so large. Even if the knapsack has a large capacity, if the weights of the items are chosen from a reasonably small set of possible weights, many sets of items will have the same total weight, greatly reducing the running time.

This algorithm falls into a complexity class called **pseudo-polynomial**. A careful explanation of this concept is beyond the scope of this book. Roughly speaking, `fast_max_val` is exponential in the number of bits needed to represent the possible values of `avail`.

To see what happens when the values of `avail` are chosen from a considerably larger space, change the call to `max_val` in the function `big_test` in [Figure 15-6](#) to

```
val, taken = fast_max_val(items, 1000)
```

Finding a solution now takes 1,028,403 calls of `fast_max_val` when the number of items is 1024.

To see what happens when the weights are chosen from an enormous space, we can choose the possible weights from the positive reals rather than the positive integers. To do this, replace the line,

```
items.append(Item(str(i),
                  random.randint(1, max_val),
                  random.randint(1, max_weight)))
```

in `build_many_items` by the line

```
items.append(Item(str(i),
                  random.randint(1, max_val),
                  random.randint(1,
                  max_weight)*random.random()))
```

Each time it is called, `random.random()` returns a random floating-point number between 0.0 and 1.0, so there are, for all intents and

purposes, an infinite number of possible weights. Don't hold your breath waiting for this last test to finish. Dynamic programming may be a miraculous technique in the common sense of the word,<sup>99</sup> but it is not capable of performing miracles in the liturgical sense.

---

### 15.3 Dynamic Programming and Divide-and-Conquer

Like divide-and-conquer algorithms, dynamic programming is based upon solving independent subproblems and then combining those solutions. There are, however, some important differences.

Divide-and-conquer algorithms are based upon finding subproblems that are substantially smaller than the original problem. For example, merge sort works by dividing the problem size in half at each step. In contrast, dynamic programming involves solving problems that are only slightly smaller than the original problem. For example, computing the nineteenth Fibonacci number is not a substantially smaller problem than computing the twentieth Fibonacci number.

Another important distinction is that the efficiency of divide-and-conquer algorithms does not depend upon structuring the algorithm so that identical problems are solved repeatedly. In contrast, dynamic programming is efficient only when the number of distinct subproblems is significantly smaller than the total number of subproblems.

---

### 15.4 Terms Introduced in Chapter

- dynamic programming
- optimal substructure
- overlapping subproblems
- memoization
- tabular method
- rooted binary tree

root

leaf

decision tree

pseudo-polynomial complexity

---

**94.** As quoted in Stuart Dreyfus “Richard Bellman on the Birth of Dynamic Programming,” *Operations Research*, vol. 50, no. 1 (2002).

**95.** It may seem odd to put the root of a tree at the top, but that is the way that mathematicians and computer scientists usually draw them. Perhaps it is evidence that those folks do not spend enough time contemplating nature.

**96.** Decision trees, which need not be binary, provide a structured way to explore the consequences of making a series of sequential decisions. They are used extensively in many fields.

**97.** Since  $2^{138} =$   
340,282,366,920,938,463,463,374,607,431,768,211,456

**98.** OK, “discovered” may be too strong a word. People have known this for a long time. You probably figured it out around Chapter 9.

**99.** Extraordinary and bringing welcome consequences.

# 16

## RANDOM WALKS AND MORE ABOUT DATA VISUALIZATION

This book is about using computation to solve problems. Thus far, we have focused our attention on problems that can be solved by a **deterministic program**. A program is deterministic if whenever it is run on the same input, it produces the same output. Such computations are highly useful, but clearly not sufficient to tackle some kinds of problems. Many aspects of the world in which we live can be accurately modeled only as **stochastic processes**.<sup>100</sup> A process is stochastic if its next state can depend upon some random element. The outcome of a stochastic process is usually uncertain. Therefore, we can rarely make definitive statements about what a stochastic process will do. Instead, we make probabilistic statements about what they might do. Much of the rest of this book deals with building programs that help to understand uncertain situations. Many of these programs will be simulation models.

A simulation mimics the activity of a real system. For example, the code in Figure 10-11 simulates a person making a series of mortgage payments. Think of that code as an experimental device, called a **simulation model**, that provides useful information about the possible behaviors of the system being modeled. Among other things, simulations are widely used to predict a future state of a physical system (e.g., the temperature of the planet 50 years from now), and in lieu of real-world experiments that would be too expensive, time consuming, or dangerous to perform (e.g., the impact of a change in the tax code).

It is important to always remember that simulation models, like all models, are only an approximation of reality. We can never be

sure that the actual system will behave in the way predicted by the model. In fact, we can usually be pretty confident that the actual system will not behave exactly as predicted by the model. For example, not every borrower will make all mortgage payments on time. It is a commonly quoted truism that “all models are wrong, but some are useful.”<sup>[101](#)</sup>

---

## 16.1 Random Walks

In 1827, the Scottish botanist Robert Brown observed that pollen particles suspended in water seemed to float around at random. He had no plausible explanation for what came to be known as Brownian motion, and made no attempt to model it mathematically.<sup>[102](#)</sup> A clear mathematical model of the phenomenon was first presented in 1900 in Louis Bachelier's doctoral thesis, *The Theory of Speculation*. However, since this thesis dealt with the then disreputable problem of understanding financial markets, it was largely ignored by respectable academics. Five years later, a young Albert Einstein brought this kind of stochastic thinking to the world of physics with a mathematical model almost the same as Bachelier's and a description of how it could be used to confirm the existence of atoms.<sup>[103](#)</sup> For some reason, people seemed to think that understanding physics was more important than making money, and the world started paying attention. Times were certainly different.

Brownian motion is an example of a **random walk**. Random walks are widely used to model physical processes (e.g., diffusion), biological processes (e.g., the kinetics of displacement of RNA from heteroduplexes by DNA), and social processes (e.g., movements of the stock market).

In this chapter we look at random walks for three reasons:

- Random walks are intrinsically interesting and widely used.
- They provide us with a good example of how to use abstract data types and inheritance to structure programs in general and simulation models in particular.
- They provide an opportunity to introduce a few more features of Python and to demonstrate some additional techniques for producing plots.

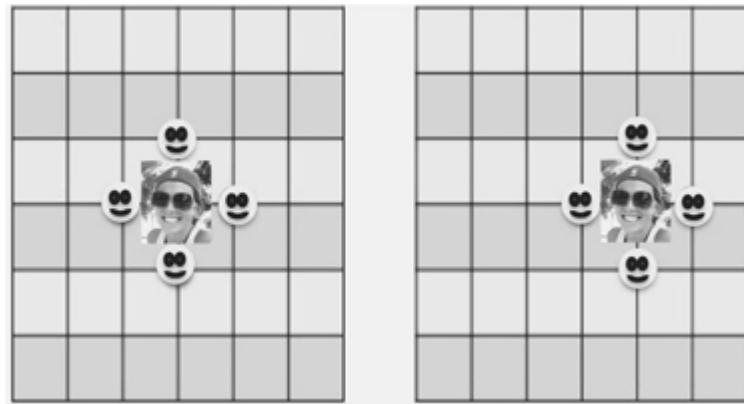
---

## 16.2 The Drunkard's Walk

Let's look at a random walk that actually involves walking. A drunken farmer is standing in the middle of a field, and every second the farmer takes one step in a random direction. What is her (or his) expected distance from the origin in 1000 seconds? If she takes many steps, is she likely to move ever farther from the origin, or is she more likely to wander back to the origin over and over, and end up not far from where she started? Let's write a simulation to find out.

Before starting to design a program, it is always a good idea to try to develop some intuition about the situation the program is intended to model. Let's start by sketching a simple model of the situation using Cartesian coordinates. Assume that the farmer is standing in a field where the grass has, mysteriously, been cut to resemble a piece of graph paper. Assume further that each step the farmer takes is of length one and is parallel to either the x-axis or y-axis.

The picture on the left of [Figure 16-1](#) depicts a farmer<sup>104</sup> standing in the middle of the field. The smiley faces indicate all the places the farmer might be after one step. Notice that after one step she is always exactly one unit away from where she started. Let's assume that she wanders eastward from her initial location on her first step. How far away might she be from her initial location after her second step?



[Figure 16-1](#) An unusual farmer

Looking at the smiley faces in the picture on the right, we see that with a probability of 0.25 she will be 0 units away, with a probability of 0.25 she will be 2 units away, and with a probability of 0.5 she will be  $\sqrt{2}$  units away.<sup>105</sup> So, on average she will be farther away after two steps than after one step. What about the third step? If the second step is to the top or bottom smiley face, the third step will bring the farmer closer to the origin half the time and farther half the time. If the second step is to the left smiley face (the origin), the third step will be away from the origin. If the second step is to the right smiley face, the third step will be closer to the origin a quarter of the time, and farther away three quarters of the time.

It seems as if the more steps the drunk takes, the greater the expected distance from the origin. We could continue this exhaustive enumeration of possibilities and perhaps develop a pretty good intuition about how this distance grows with respect to the number of steps. However, it is getting tedious, so it seems like a better idea to write a program to do it for us.

Let's begin the design process by thinking about some data abstractions that are likely to be useful in building this simulation and perhaps simulations of other kinds of random walks. As usual, we should try to invent types that correspond to the kinds of things that appear in the situation we are attempting to model. Three obvious types are `Location`, `Field`, and `Drunk`. As we look at the classes providing these types, it is worthwhile to think about what each might imply about the kinds of simulation models they will allow us to build.

Let's start with `Location`, [Figure 16-2](#). This is a simple class, but it does embody two important decisions. It tells us that the simulation will involve at most two dimensions. This is consistent with the pictures above. Also, since the values supplied for `delta_x` and `delta_y` could be floats rather than integers, there is no built-in assumption in this class about the set of directions in which a drunk might move. This is a generalization of the informal model in which each step was of length one and was parallel to the x-axis or y-axis.

Class `Field`, [Figure 16-2](#), is also quite simple, but it too embodies notable decisions. It simply maintains a mapping of drunks to locations. It places no constraints on locations, so presumably a `Field` is of unbounded size. It allows multiple drunks to be added into a `Field` at random locations. It says nothing about the patterns in which drunks move, nor does it prohibit multiple drunks from occupying the same location or moving through spaces occupied by other drunks.

```

class Location(object):
    def __init__(self, x, y):
        """x and y are numbers"""
        self._x, self._y = x, y
    def move(self, delta_x, delta_y):
        """delta_x and delta_y are numbers"""
        return Location(self._x + delta_x, self._y + delta_y)

    def get_x(self):
        return self._x

    def get_y(self):
        return self._y

    def dist_from(self, other):
        ox, oy = other._x, other._y
        x_dist, y_dist = self._x - ox, self._y - oy
        return (x_dist**2 + y_dist**2)**0.5

    def __str__(self):
        return f'<{self._x}, {self._y}>'

class Field(object):
    def __init__(self):
        self._drunks = {}

    def add_drunk(self, drunk, loc):
        if drunk in self._drunks:
            raise ValueError('Duplicate drunk')
        else:
            self._drunks[drunk] = loc

    def move_drunk(self, drunk):
        if drunk not in self._drunks:
            raise ValueError('Drunk not in field')
        x_dist, y_dist = drunk.take_step()
        current_location = self._drunks[drunk]
        #use move method of Location to get new location
        self._drunks[drunk] = current_location.move(x_dist, y_dist)

    def get_loc(self, drunk):
        if drunk not in self._drunks:
            raise ValueError('Drunk not in field')
        return self._drunks[drunk]

```

[Figure 16-2](#) Location and Field classes

The classes Drunk and Usual\_drunk in [Figure 16-3](#) define ways in which a drunk might wander through the field. In particular, the value of step\_choices in Usual\_drunk introduces the restriction that

each step is of length one and is parallel to either the x-axis or y-axis. Since the function `random.choice` returns a randomly chosen member of the sequence that it is passed, each kind of step is equally likely and not influenced by previous steps. Later we will look at subclasses of `Drunk` with different kinds of behaviors.

```
class Drunk(object):
    def __init__(self, name = None):
        """Assumes name is a str"""
        self._name = name

    def __str__(self):
        if self != None:
            return self._name
        return 'Anonymous'

class Usual_drunk(Drunk):
    def take_step(self):
        step_choices = [(0,1), (0,-1), (1, 0), (-1, 0)]
        return random.choice(step_choices)
    return random.choice(step_choices)
```

[Figure 16-3](#) Classes defining Drunks

The next step is to use these classes to build a simulation that answers the original question. [Figure 16-4](#) contains three functions used in this simulation.

```

def walk(f, d, num_steps):
    """Assumes: f a Field, d a Drunk in f, and num_steps an int >= 0.
    Moves d num_steps times; returns the distance between the
    final location and the location at the start of the walk."""
    start = f.get_loc(d)
    for s in range(num_steps):
        f.move_drunk(d)
    return start.dist_from(f.get_loc(d))

def sim_walks(num_steps, num_trials, d_class):
    """Assumes num_steps an int >= 0, num_trials an int > 0,
       d_class a subclass of Drunk
    Simulates num_trials walks of num_steps steps each.
    Returns a list of the final distances for each trial"""
    Homer = d_class()
    origin = Location(0, 0)
    distances = []
    for t in range(num_trials):
        f = Field()
        f.add_drunk(Homer, origin)
        distances.append(round(walk(f, Homer, num_steps), 1))
    return distances

def drunk_test(walk_lengths, num_trials, d_class):
    """Assumes walk_lengths a sequence of ints >= 0
       num_trials an int > 0, d_class a subclass of Drunk
       For each number of steps in walk_lengths, runs sim_walks with
       num_trials walks and prints results"""
    for num_steps in walk_lengths:
        distances = sim_walks(num_steps, num_trials, d_class)
        print(d_class.__name__, 'walk of', num_steps, 'steps: Mean =',
              f'{sum(distances)/len(distances):.3f}, Max =',
              f'{max(distances)}, Min = {min(distances)}')

```

[Figure 16-4](#) The drunkard's walk (with a bug)

The function `walk` simulates one walk of `num_steps` steps. The function `sim_walks` calls `walk` to simulate `num_trials` walks of `num_steps` steps each. The function `drunk_test` calls `sim_walks` to simulate walks of varying lengths.

The parameter `d_class` of `sim_walks` is of type `class`, and is used in the first line of code to create a `Drunk` of the appropriate subclass. Later, when `drunk.take_step` is invoked from `Field.move_drunk`, the method from the appropriate subclass is automatically selected.

The function `drunk_test` also has a parameter, `d_class`, of type `class`. It is used twice, once in the call to `sim_walks` and once in the first `print` statement. In the `print` statement, the built-in `class` attribute `__name__` is used to get a string with the name of the class.

When we executed `drunk_test((10, 100, 1000, 10000), 100, Usual_drunk)`, it printed

```
Usual_drunk walk of 10 steps: Mean = 8.634, Max = 21.6, Min = 1.4
Usual_drunk walk of 100 steps: Mean = 8.57, Max = 22.0, Min = 0.0
Usual_drunk walk of 1000 steps: Mean = 9.206, Max = 21.6, Min = 1.4
Usual_drunk walk of 10000 steps: Mean = 8.727, Max = 23.5, Min = 1.4
```

This is surprising, given the intuition we developed earlier that the mean distance should grow with the number of steps. It could mean that our intuition is wrong, or it could mean that our simulation is buggy, or both.

The first thing to do at this point is to run the simulation on values for which we already think we know the answer, and make sure that what the simulation produces matches the expected result. Let's try walks of zero steps (for which the mean, minimum and maximum distances from the origin should all be 0) and one step (for which the mean, minimum and maximum distances from the origin should all be 1).

When we ran `drunk_test((0,1), 100, Usual_drunk)`, we got the highly suspect result

```
Usual_drunk walk of 0 steps: Mean = 8.634, Max = 21.6, Min = 1.4
Usual_drunk walk of 1 steps: Mean = 8.57, Max = 22.0, Min = 0.0
```

How on earth can the mean distance of a walk of zero steps be over 8? We must have at least one bug in our simulation. After some investigation, the problem is clear. In `sim_walks`, the function call `walk(f, Homer, num_trials)` should have been `walk(f, Homer, num_steps)`.

The moral here is an important one: Always bring some skepticism to bear when looking at the results of a simulation. First ask if the results pass the smell test (i.e., are plausible). And always **smoke test**<sup>106</sup> the simulation on parameters for which you have a strong intuition about what the results should be.

When the corrected version of the simulation is run on our two simple cases, it yields exactly the expected answers:

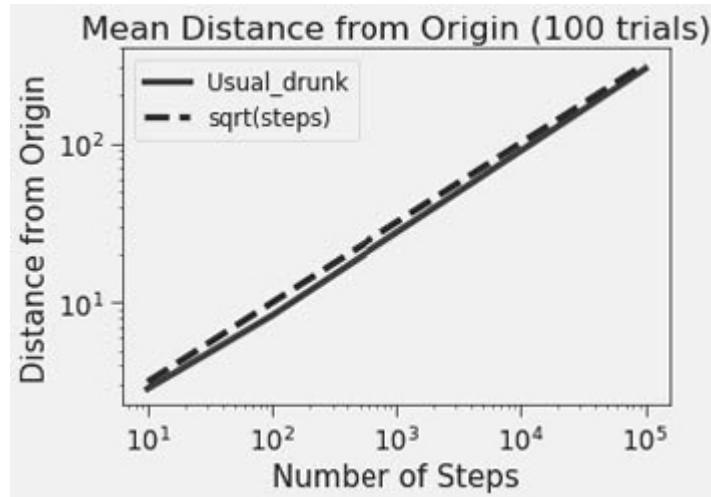
```
Usual_drunk walk of 0 steps: Mean = 0.0, Max = 0.0, Min =  
0.0  
Usual_drunk walk of 1 steps: Mean = 1.0, Max = 1.0, Min =  
1.0
```

When run on longer walks it printed

```
Usual_drunk walk of 10 steps: Mean = 2.863, Max = 7.2, Min =  
0.0  
Usual_drunk walk of 100 steps: Mean = 8.296, Max = 21.6, Min  
= 1.4  
Usual_drunk walk of 1000 steps: Mean = 27.297, Max = 66.3,  
Min = 4.2  
Usual_drunk walk of 10000 steps: Mean = 89.241, Max = 226.5,  
Min = 10.0
```

As anticipated, the mean distance from the origin grows with the number of steps.

Now let's look at a plot of the mean distances from the origin, [Figure 16-5](#). To give a sense of how fast the distance is growing, we have placed on the plot a line showing the square root of the number of steps (and increased the number of steps to 100,000).



[Figure 16-5](#) Distance from starting point versus steps taken

**Finger exercise:** Write code to produce the plot in [Figure 16-5](#).

Does this plot provide any information about the expected final location of a drunk? It does tell us that on average the drunk will be somewhere on a circle with its center at the origin and with a radius equal to the expected distance from the origin. However, it tells us little about where we might actually find the drunk at the end of any particular walk. We return to this topic in the next section.

### 16.3 Biased Random Walks

Now that we have a working simulation, we can start modifying it to investigate other kinds of random walks. Suppose, for example, that we want to consider the behavior of a drunken farmer in the northern hemisphere who hates the cold, and even in his drunken stupor is able to move twice as fast when his random movements take him in a southward direction. Or maybe a phototropic drunk who always moves towards the sun (east in the morning and west in the afternoon). These are examples of **biased random walks**. The walk is still stochastic, but there is a bias in the outcome.

[Figure 16-6](#) defines two additional subclasses of `Drunk`. In each case the specialization involves choosing an appropriate value for `step_choices`. The function `sim_all` iterates over a sequence of

subclasses of `Drunk` to generate information about how each kind behaves.

```
class Cold_drunk(Drunk):
    def take_step(self):
        stepChoices = [(0.0,1.0), (0.0,-2.0), (1.0, 0.0), (-1.0, 0.0)]
        return random.choice(stepChoices)

class EW_drunk(Drunk):
    def take_step(self):
        stepChoices = [(1.0, 0.0), (-1.0, 0.0)]
        return random.choice(stepChoices)

def sim_all(drunk_kinds, walk_lengths, num_trials):
    for d_class in drunk_kinds:
        drunk_test(walk_lengths, num_trials, d_class)
```

[Figure 16-6](#) Subclasses of `Drunk` base class

When we ran

```
sim_all((Usual_drunk, Cold_drunk, EW_drunk), (100, 1000),
10)
```

it printed

```
Usual_drunk walk of 100 steps: Mean = 9.64, Max = 17.2, Min
= 4.2
Usual_drunk walk of 1000 steps: Mean = 22.37, Max = 45.5,
Min = 4.5
Cold_drunk walk of 100 steps: Mean = 27.96, Max = 51.2, Min
= 4.1
Cold_drunk walk of 1000 steps: Mean = 259.49, Max = 320.7,
Min = 215.1
EW_drunk walk of 100 steps: Mean = 7.8, Max = 16.0, Min =
0.0
EW_drunk walk of 1000 steps: Mean = 20.2, Max = 48.0, Min =
4.0
```

It appears that our heat-seeking drunk moves away from the origin faster than the other two kinds of drunk. However, it is not easy to digest all of the information in this output. It is once again time to move away from textual output and start using plots.

Since we are showing different kinds of drunks on the same plot, we will associate a distinct style with each type of drunk so that it is easy to differentiate among them. The style will have three aspects:

- The color of the line and marker
- The shape of the marker
- The kind of the line, e.g., solid or dotted.

The class `style_iterator`, [Figure 16-7](#), rotates through a sequence of styles defined by the argument to `style_iterator.__init__`.

```
class style_iterator(object):
    def __init__(self, styles):
        self.index = 0
        self.styles = styles

    def next_style(self):
        result = self.styles[self.index]
        if self.index == len(self.styles) - 1:
            self.index = 0
        else:
            self.index += 1
        return result
```

[Figure 16-7](#) Iterating over styles

The code in [Figure 16-8](#) is similar in structure to that in [Figure 16-4](#).

```

def sim_drunk(num_trials, d_class, walk_lengths):
    meanDistances = []
    for num_steps in walk_lengths:
        print('Starting simulation of', num_steps, 'steps')
        trials = sim_walks(num_steps, num_trials, d_class)
        mean = sum(trials)/len(trials)
        meanDistances.append(mean)
    return meanDistances

def sim_all_plot(drunk_kinds, walk_lengths, num_trials):
    style_choice = style_iterator((m-, r:, k-.))
    for d_class in drunk_kinds:
        cur_style = style_choice.next_style()
        print('Starting simulation of', d_class.__name__)
        means = sim_drunk(num_trials, d_class, walk_lengths)
        plt.plot(walk_lengths, means, cur_style,
                  label = d_class.__name__)
    plt.title(f'Mean Distance from Origin ({num_trials} trials')
    plt.xlabel('Number of Steps')
    plt.ylabel('Distance from Origin')
    plt.legend(loc = 'best')
    plt.semilogx()
    plt.semilogy()

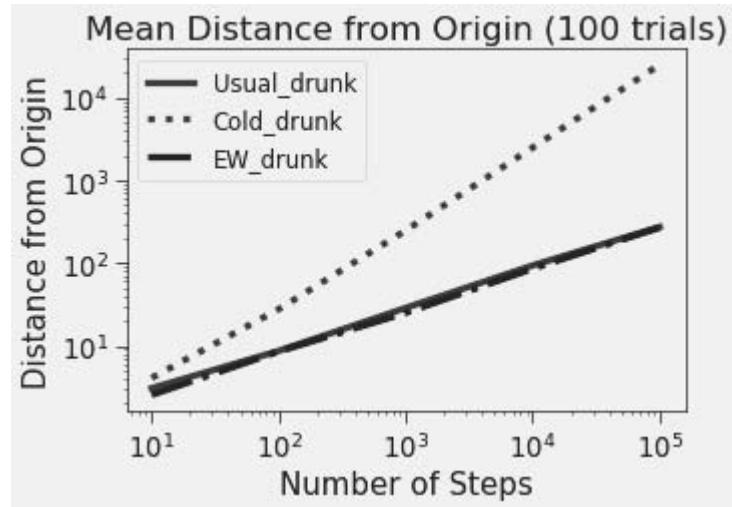
```

[Figure 16-8](#) Plotting the walks of different drunks

The `print` statements in `sim_drunk` and `sim_all_plot` contribute nothing to the result of the simulation. They are there because this simulation can take a rather long time to complete, and printing an occasional message indicating that progress is being made can be reassuring to a user who might be wondering if the program is actually making progress.

The plot in [Figure 16-9](#) was produced by executing

```
sim_all_plot((Usual_drunk, Cold_drunk, EW_drunk),
             (10, 100, 1000, 10000, 100000), 100)
```



[Figure 16-9](#) Mean distance for different kinds of drunks

The usual drunk and the phototropic drunk (`EW_drunk`) seem to be moving away from the origin at approximately the same pace, but the heat-seeking drunk (`Cold_drunk`) seems to be moving away orders of magnitude faster. This is interesting, since on average he is only moving 25% faster (he takes, on average, five steps for every four taken by the others).

Let's construct a different plot to help us get more insight into the behavior of these three classes. Instead of plotting the change in distance over time for an increasing number of steps, the code in [Figure 16-10](#) plots the distribution of final locations for a single number of steps.

```

def get_final_locs(num_steps, num_trials, d_class):
    locs = []
    d = d_class()
    for t in range(num_trials):
        f = Field()
        f.add_drunk(d, Location(0, 0))
        for s in range(num_steps):
            f.move_drunk(d)
        locs.append(f.get_loc(d))
    return locs

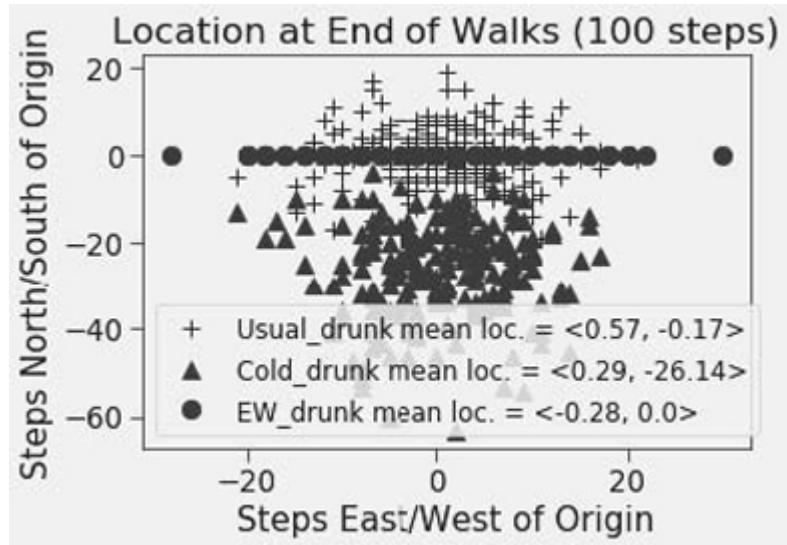
def plot_locs(drunk_kinds, num_steps, num_trials):
    style_choice = style_iterator(('k+', 'r^', 'mo'))
    for d_class in drunk_kinds:
        locs = get_final_locs(num_steps, num_trials, d_class)
        x_vals, y_vals = [], []
        for loc in locs:
            x_vals.append(loc.get_x())
            y_vals.append(loc.get_y())
        meanX = sum(x_vals)/len(x_vals)
        meanY = sum(y_vals)/len(y_vals)
        cur_style = style_choice.next_style()
        plt.plot(x_vals, y_vals, cur_style,
                  label = (f'{d_class.__name__} mean loc. = <' +
                            f'{meanX}, {meanY}>'))
    plt.title(f'Location at End of Walks ({num_steps} steps)')
    plt.xlabel('Steps East/West of Origin')
    plt.ylabel('Steps North/South of Origin')
    plt.legend(loc = 'best')

```

[Figure 16-10](#) Plotting final locations

The first thing `plot_locs` does is create an instance of `style_iterator` with three styles of markers. It then uses `plt.plot` to place a marker at a location corresponding to the end of each trial. The call to `plt.plot` sets the color and shape of the marker to be plotted using the values returned by the iterator `style_iterator`.

The call `plot_locs((Usual_drunk, Cold_drunk, EW_drunk), 100, 200)` produces the plot in [Figure 16-11](#).



[Figure 16-11](#) Where the drunk stops

The first thing to say is that our drunks seem to be behaving as advertised. The `EW_drunk` ends up on the x-axis, the `Cold_drunk` seem to have made progress southwards, and the `Usual_drunk` seem to have wandered aimlessly.

But why do there appear to be far fewer circle markers than triangle or + markers? Because many of the `EW_drunk`'s walks ended up at the same place. This is not surprising, given the small number of possible endpoints (200) for the `EW_drunk`. Also the circle markers seem to be fairly uniformly spaced across the x-axis.

It is still not immediately obvious, at least to us, why the `Cold_drunk` manages, on average, to get so much farther from the origin than the other kinds of drunks. Perhaps it's time to look not at the endpoints of many walks, but at the path followed by a single walk. The code in [Figure 16-12](#) produces the plot in [Figure 16-13](#).

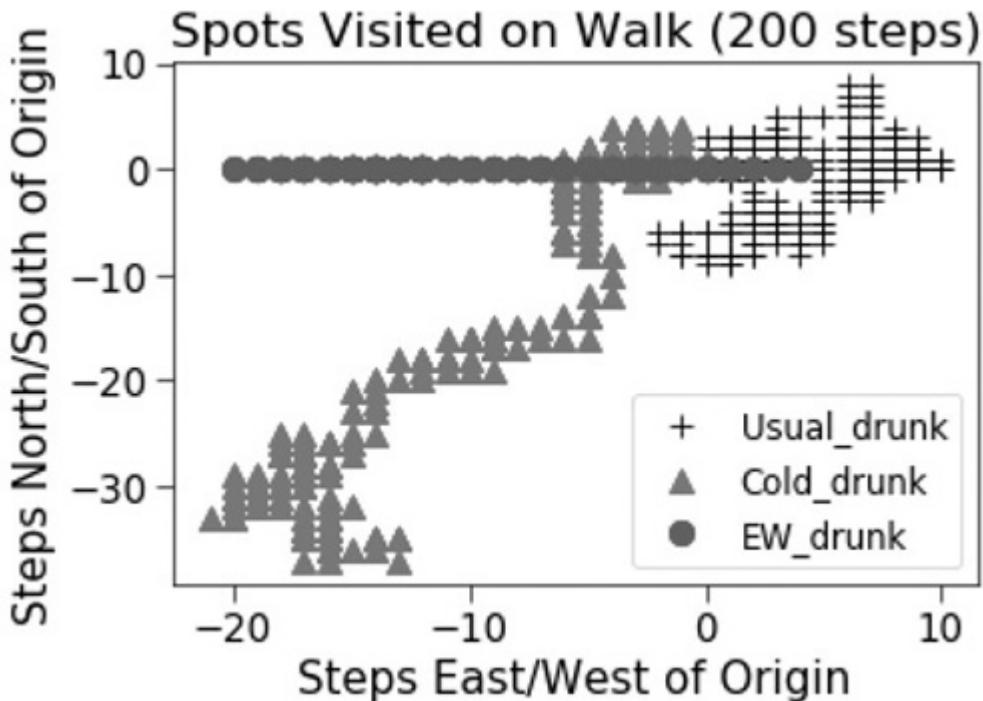
```

def trace_walk(drunk_kinds, num_steps):
    style_choice = style_iterator(('k+', 'r^', 'mo'))
    f = Field()
    for d_class in drunk_kinds:
        d = d_class()
        f.add_drunk(d, Location(0, 0))
        locs = []
        for s in range(num_steps):
            f.move_drunk(d)
            locs.append(f.get_loc(d))
    x_vals, y_vals = [], []
    for loc in locs:
        x_vals.append(loc.get_x())
        y_vals.append(loc.get_y())
    cur_style = style_choice.next_style()
    plt.plot(x_vals, y_vals, cur_style,
              label = d_class.__name__)
    plt.title('Spots Visited on Walk (' +
              + str(num_steps) + ' steps)')
    plt.xlabel('Steps East/West of Origin')
    plt.ylabel('Steps North/South of Origin')
    plt.legend(loc = 'best')

trace_walk((Usual_drunk, Cold_drunk, EW_drunk), 200)

```

[Figure 16-12](#) Tracing walks



[Figure 16-13](#) Trajectory of walks

Since the walk is 200 steps long and the `EW_drunk`'s walk visits fewer than 30 different locations, it's clear that he is spending a lot of time retracing his steps. The same kind of observation holds for the `Usual_drunk`. In contrast, while the `Cold_drunk` is not exactly making a beeline for Florida, he is managing to spend relatively less time visiting places he has already been.

None of these simulations is interesting in its own right. (In Chapter 18, we will look at more intrinsically interesting simulations.) But there are some points worth taking away:

- Initially we divided our simulation code into four separate chunks. Three of them were classes (`Location`, `Field`, and `Drunk`) corresponding to abstract data types that appeared in the informal description of the problem. The fourth chunk was a group of functions that used these classes to perform a simple simulation.
- We then elaborated `Drunk` into a hierarchy of classes so that we could observe different kinds of biased random walks. The code

for `Location` and `Field` remained untouched, but the simulation code was changed to iterate through the different subclasses of `Drunk`. In doing this, we took advantage of the fact that a class is itself an object, and therefore can be passed as an argument.

- Finally, we made a series of incremental changes to the simulation that did not involve any changes to the classes representing the abstract types. These changes mostly involved introducing plots designed to provide insight into the different walks. This is typical of the way in which simulations are developed. Get the basic simulation working first, and then start adding features.

---

## 16.4 Treacherous Fields

Did you ever play the board game known as *Chutes and Ladders* in the U.S. and *Snakes and Ladders* in the UK? This children's game originated in India (perhaps in the second century BCE), where it was called *Moksha-patamu*. Landing on a square representing virtue (e.g., generosity) sent a player up a ladder to a higher tier of life. Landing on a square representing evil (e.g., lust), sent a player back to a lower tier of life.

We can easily add this kind of feature to our random walks by creating a `Field` with wormholes,<sup>107</sup> as shown in [Figure 16-14](#), and replacing the second line of code in the function `trace_walk` by the line of code

```
f = Odd_field(1000, 100, 200)
```

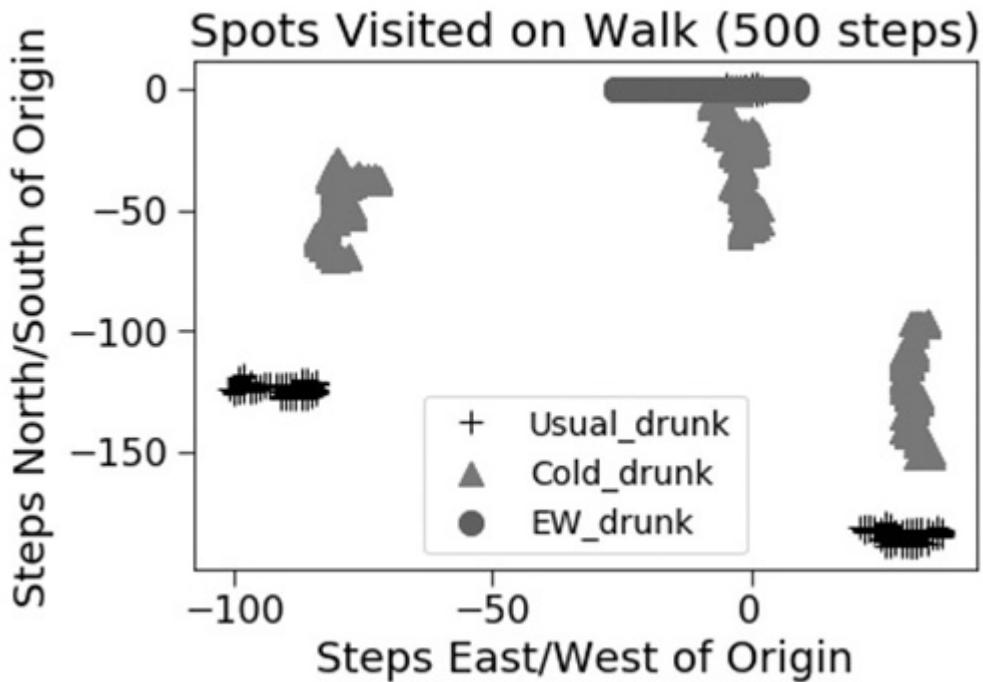
In an `Odd_field`, a drunk who steps into a wormhole location is transported to the location at the other end of the wormhole.

```
class Odd_field(Field):
    def __init__(self, numHoles, x_range, y_range):
        Field.__init__(self)
        self.wormholes = {}
        for w in range(numHoles):
            x = random.randint(-x_range, x_range)
            y = random.randint(-y_range, y_range)
            newX = random.randint(-x_range, x_range)
            newY = random.randint(-y_range, y_range)
            newLoc = Location(newX, newY)
            self.wormholes[(x, y)] = newLoc

    def move_drunk(self, drunk):
        Field.move_drunk(self, drunk)
        x = self.drunks[drunk].get_x()
        y = self.drunks[drunk].get_y()
        if (x, y) in self.wormholes:
            self.drunks[drunk] = self.wormholes[(x, y)]
```

[Figure 16-14](#) Fields with strange properties

When we ran `trace_walk((Usual_drunk, Cold_drunk, EW_drunk), 500)`, we got the rather odd-looking plot in [Figure 16-15](#).



[Figure 16-15](#) A strange walk

Clearly changing the properties of the field has had a dramatic effect. However, that is not the point of this example. The main points are:

- Because of the way we structured our code, it was easy to accommodate a significant change to the situation being modeled. Just as we could add different kinds of drunks without touching `Field`, we can add a new kind of `Field` without touching `Drunk` or any of its subclasses. (Had we been sufficiently prescient to make the field a parameter of `trace_walk`, we wouldn't have had to change `trace_walk` either.)
- While it would have been feasible to analytically derive different kinds of information about the expected behavior of the simple random walk and even the biased random walks, it would have been challenging to do so once the wormholes were introduced. Yet it was exceedingly simple to change the simulation to model the new situation. Simulation models often enjoy this advantage relative to analytic models.

---

## 16.5 Terms Introduced in Chapterdeterministic program

stochastic process

simulation model

random walk

smoke test

biased random walks

logarithmic scale

---

100 The word stems from the Greek word *stokhastikos*, which means something like “capable of divining.” A stochastic program, as we shall see, is aimed at getting a good result, but the exact results are not guaranteed.

101 Usually attributed to the statistician George E.P. Box.

102 Nor was he the first to observe it. As early as 60 BCE, the Roman Titus Lucretius, in his poem “On the Nature of Things,” described a similar phenomenon, and even implied that it was caused by the random movement of atoms.

103 “On the movement of small particles suspended in a stationary liquid demanded by the molecular-kinetic theory of heat,” *Annalen der Physik*, May 1905. Einstein would come to describe 1905 as his “*annus mirabilis*.” That year, in addition to his paper on Brownian motion, he published papers on the production and transformation of light (pivotal to the development of quantum theory), on the electrodynamics of moving bodies (special relativity), and on the equivalence of matter and energy ( $E = mc^2$ ). Not a bad year for a newly minted PhD.

104 To be honest, the person pictured here is not really a farmer, or even a serious gardener.

105 Why  $\sqrt{2}$ ? We are using the Pythagorean theorem.

106 In the nineteenth century, it became standard practice for plumbers to test closed systems of pipes for leaks by filling the system with smoke. Later, electronic engineers adopted the term to cover the very first test of a piece of electronics—turning on the power and looking for smoke. Still later, software developers starting using the term for a quick test to see if a program did anything useful.

107 This kind of wormhole is a hypothetical concept invented by theoretical physicists (or maybe science fiction writers). It provides shortcuts through the time/space continuum.

# 17

## STOCHASTIC PROGRAMS, PROBABILITY, AND DISTRIBUTIONS

There is something comforting about Newtonian mechanics. You push down on one end of a lever, and the other end goes up. You throw a ball up in the air; it travels a parabolic path, and eventually comes down.  $\ddot{F} = m\ddot{a}$ . In short, everything happens for a reason. The physical world is a completely predictable place—all future states of a physical system can be derived from knowledge about its current state.

For centuries, this was the prevailing scientific wisdom; then along came quantum mechanics and the Copenhagen Doctrine. The doctrine's proponents, led by Bohr and Heisenberg, argued that at its most fundamental level, the behavior of the physical world cannot be predicted. One can make probabilistic statements of the form “x is highly likely to occur,” but not statements of the form “x is certain to occur.” Other distinguished physicists, most notably Einstein and Schrödinger, vehemently disagreed.

This debate roiled the worlds of physics, philosophy, and even religion. The heart of the debate was the validity of **causal nondeterminism**, i.e., the belief that not every event is caused by previous events. Einstein and Schrödinger found this view philosophically unacceptable, as exemplified by Einstein's often-repeated comment, “God does not play dice.” What they could accept was **predictive nondeterminism**, i.e., the concept that our inability to make accurate measurements about the physical world makes it impossible to make precise predictions about future states. This distinction was nicely summed up by Einstein, who said, “The essentially statistical character of contemporary theory is solely to be

ascribed to the fact that this theory operates with an incomplete description of physical systems.”

The question of causal nondeterminism is still unsettled. However, whether the reason we cannot predict events is because they are truly unpredictable or because we simply don't have enough information to predict them is of no practical importance.

While the Bohr/Einstein debate was about how to understand the lowest levels of the physical world, the same issues arise at the macroscopic level. Perhaps the outcomes of horse races, spins of roulette wheels, and stock market investments are causally deterministic. However, there is ample evidence that it is perilous to treat them as predictably deterministic.[108](#)

---

## 17.1 Stochastic Programs

A program is **deterministic** if whenever it is run on the same input, it produces the same output. Notice that this is not the same as saying that the output is completely defined by the specification of the problem. Consider, for example, the specification of `square_root`:

```
def square_root(x, epsilon):
    """Assumes x and epsilon are of type float; x >= 0 and
    epsilon > 0
    Returns float y such that x-epsilon <= y*y <=
    x+epsilon"""

```

This specification admits many possible return values for the function call `square_root(2, 0.001)`. However, the successive approximation algorithm that we looked at in Chapter 3 will always return the same value. The specification doesn't require that the implementation be deterministic, but it does allow deterministic implementations.

Not all interesting specifications can be met by deterministic implementations. Consider, for example, implementing a program to play a dice game, say backgammon or craps. Somewhere in the program will be a function that simulates a fair roll of a single six-sided die.[109](#) Suppose it had a specification something like

```
def roll_die():
    """Returns an int between 1 and 6"""

```

This would be problematic, since it allows the implementation to return the same number each time it is called, which would make for a pretty boring game. It would be better to specify that `roll_die` “returns a randomly chosen int between 1 and 6,” thus requiring a stochastic implementation.

Most programming languages, including Python, include simple ways to write stochastic programs, i.e., programs that exploit randomness. The tiny program in [Figure 17-1](#) is a simulation model. Rather than asking some person to roll a die multiple times, we wrote a program to simulate that activity. The code uses one of several useful functions found in the imported Python standard library module `random`. (The code in this chapter assumes that the imports

```
import random
import numpy as np
import matplotlib.pyplot as plt
import scipy.integrate
```

have all been executed.)

The function `random.choice` takes a non-empty sequence as its argument and returns a randomly chosen member of that sequence. Almost all of the functions in `random` are built using the function `random.random`, which generates a random floating-point number between 0.0 and 1.0. [110](#)

```
def roll_die():
    """Returns a random int between 1 and 6"""
    return random.choice([1,2,3,4,5,6])

def roll_n(n):
    result = ''
    for i in range(n):
        result = result + str(roll_die())
    print(result)
```

[Figure 17-1](#) Roll die

Now, imagine running `roll_n(10)`. Would you be more surprised to see it print 1111111111 or 5442462412? Or, to put it another way,

which of these two sequences is more random? It's a trick question. Each of these sequences is equally likely, because the value of each roll is independent of the values of earlier rolls. In a stochastic process, two events are **independent** if the outcome of one event has no influence on the outcome of the other.

This is easier to see if we simplify the situation by thinking about a two-sided die (also known as a coin) with the values 0 and 1. This allows us to think of the output of a call of `roll_n` as a binary number. When we use a binary die, there are  $2^n$  possible sequences that `roll_n` might return. Each of these sequences is equally likely; therefore each has a probability of occurring of  $(1/2)^n$ .

Let's go back to our six-sided die. How many different sequences are there of length 10?  $6^{10}$ . So, the probability of rolling 10 consecutive 1's is  $1/6^{10}$ . Less than one out of 60 million. Pretty low, but no lower than the probability of any other sequence, e.g., 5442462412.

---

## 17.2 Calculating Simple Probabilities

In general, when we talk about the **probability** of a result having some property (e.g., all 1's), we are asking what fraction of all possible results has that property. This is why probabilities range from 0 to 1. Suppose we want to know the probability of getting any sequence other than all 1's when rolling the die. It is simply  $1 - (1/6^{10})$ , because the probability of something happening and the probability of the same thing not happening must add up to 1.

Suppose we want to know the probability of rolling the die 10 times without getting a single 1. One way to answer this question is to transform it into the question of how many of the  $6^{10}$  possible sequences don't contain a 1. This can be computed as follows:

1. The probability of not rolling a 1 on any single roll is  $5/6$ .
2. The probability of not rolling a 1 on either the first or the second roll is  $(5/6)^2$ , or  $(5/6)^2$ .

3. So, the probability of not rolling a 1 10 times in a row is  $(5/6)^{10}$ , slightly more than 0.16.

Step 2 is an application of the **multiplicative law** for independent probabilities. Consider, for example, two independent events A and B. If A occurs one  $1/3$  of the time and B occurs  $1/4$  of the time, the probability that both A and B occur is  $1/4$  of  $1/3$ , i.e.,  $(1/3)/4$  or  $(1/3) * (1/4)$ .

What about the probability of rolling at least one 1? It is simply 1 minus the probability of not rolling at least one 1, i.e.,  $1 - (5/6)^{10}$ . Notice that this cannot be correctly computed by saying that the probability of rolling a 1 on any roll is  $1/6$ , so the probability of rolling at least one 1 is  $10 * (1/6)$ , i.e.,  $10/6$ . This is obviously incorrect since a probability cannot be greater than 1.

How about the probability of rolling exactly two 1's in 10 rolls? This is equivalent to asking what fraction of the first  $6^{10}$  integers has exactly two 1's in its base 6 representation. We could easily write a program to generate all of these sequences and count the number that contained exactly one 1. Deriving the probability analytically is a bit tricky, and we defer it to Section 17.4.4.

---

## 17.3 Inferential Statistics

As you just saw, we can use a systematic process to derive the precise probability of some complex event based upon knowing the probability of one or more simpler events. For example, we can easily compute the probability of flipping a coin and getting 10 consecutive heads based on the assumption that flips are independent and we know the probability of each flip coming up heads. Suppose, however, that we don't actually know the probability of the relevant simpler event. Suppose, for example, that we don't know whether the coin is fair (i.e., a coin where heads and tails are equally likely).

All is not lost. If we have some data about the behavior of the coin, we can combine that data with our knowledge of probability to derive an estimate of the true probability. We can use **inferential statistics** to estimate the probability of a single flip coming up

heads, and then conventional probability to compute the probability of a coin with that behavior coming up heads 10 times in a row.

In brief (since this is not a book about statistics), the guiding principle of inferential statistics is that a random sample tends to exhibit the same properties as the population from which it is drawn.

Suppose Harvey Dent (also known as Two-Face) flipped a coin, and it came up heads. You would not infer from this that the next flip would also come up heads. Suppose he flipped it twice, and it came up heads both times. You might reason that the probability of this happening for a fair coin was  $0.25$ , so there was still no reason to assume the next flip would be heads. Suppose, however, 100 out of 100 flips came up heads. The number  $(1/2)^{100}$  (the probability of this event, assuming a fair coin) is pretty small, so you might feel safe in inferring that the coin has a head on both sides.

Your belief in whether the coin is fair is based on the intuition that the behavior of a single sample of 100 flips is similar to the behavior of the population of all samples of 100 flips. This belief seems appropriate when all 100 flips are heads. Suppose that 52 flips came up heads and 48 tails. Would you feel comfortable in predicting that the next 100 flips would have the same ratio of heads to tails? For that matter, how comfortable would you feel about even predicting that there would be more heads than tails in the next 100 flips? Take a few minutes to think about this, and then try the experiment. Or, if you don't happen to have a coin handy, simulate the flips using the code in [Figure 17-2](#).

The function `flip` in [Figure 17-2](#) simulates flipping a fair coin `num_flips` times and returns the fraction of those flips that came up heads. For each flip, the call `random.choice(['H', 'T'])` randomly returns either 'H' or 'T'.

```

def flip(num_flips):
    """Assumes num_flips a positive int"""
    heads = 0
    for i in range(num_flips):
        if random.choice(('H', 'T')) == 'H':
            heads += 1
    return heads/num_flips

def flip_sim(num_flips_per_trial, num_trials):
    """Assumes num_flips_per_trial and num_trials positive ints"""
    frac_heads = []
    for i in range(num_trials):
        frac_heads.append(flip(num_flips_per_trial))
    mean = sum(frac_heads)/len(frac_heads)
    return mean

```

[Figure 17-2](#) Flipping a coin

Try executing the function `flip_sim(10, 1)` a couple of times. Here's what we saw the first two times we tried `print('Mean =', flip_sim(10, 1))`:

```

Mean = 0.2
Mean = 0.6

```

It seems that it would be inappropriate to assume much (other than that the coin has both heads and tails) from any one trial of 10 flips. That's why we typically structure our simulations to include multiple trials and compare the results. Let's try `flip_sim(10, 100)` a couple of times:

```

Mean = 0.5029999999999999
Mean = 0.496

```

Do you feel better about these results? When we tried `flip_sim(100, 100000)`, we got

```

Mean = 0.5005000000000038
Mean = 0.5003139999999954

```

This looks really good (especially since we know that the answer should be 0.5—but that's cheating). Now it seems we can safely

conclude something about the next flip, i.e., that heads and tails are about equally likely. But why do we think that we can conclude that?

What we are depending upon is the **law of large numbers** (also known as **Bernoulli's theorem**<sup>111</sup>). This law states that in repeated independent tests (flips in this case) with the same actual probability  $p$  of a particular outcome in each test (e.g., an actual probability of 0.5 of getting a head for each flip), the chance that the fraction of times that outcome occurs differs from  $p$  converges to zero as the number of trials goes to infinity.

It is worth noting that the law of large numbers does not imply, as too many seem to think, that if deviations from expected behavior occur, these deviations are likely to be “evened out” by opposite deviations in the future. This misapplication of the law of large numbers is known as the **gambler's fallacy**.<sup>112</sup>

People often confuse the gambler's fallacy with regression to the mean. **Regression to the mean**<sup>113</sup> states that following an extreme random event, the next random event is likely to be less extreme. If you were to flip a fair coin six times and get six heads, regression to the mean implies that the next sequence of six flips is likely to have closer to the expected value of three heads. It does not imply, as the gambler's fallacy suggests, that the next sequence of flips is likely to have fewer heads than tails.

Success in most endeavors requires a combination of skill and luck. The skill component determines the mean and the luck component accounts for the variability. The randomness of luck leads to regression to the mean.

The code in [Figure 17-3](#) produces a plot, [Figure 17-4](#), illustrating regression to the mean. The function `regress_to_mean` first generates `num_trials` trials of `num_flips` coin flips each. It then identifies all trials where the fraction of heads was either less than  $1/3$  or more than  $2/3$  and plots these extremal values as circles. Then, for each of these points, it plots the value of the subsequent trial as a triangle in the same column as the circle.

The horizontal line at 0.5, the expected mean, is created using the `axhline` function. The function `plt.xlim` controls the extent of the x-axis. The function call `plt.xlim(xmin, xmax)` sets the minimum and maximum values of the x-axis of the current figure. The function call `plt.xlim()` returns a tuple composed of the minimum and maximum

values of the x-axis of the current figure. The function `plt.ylim` works the same way.

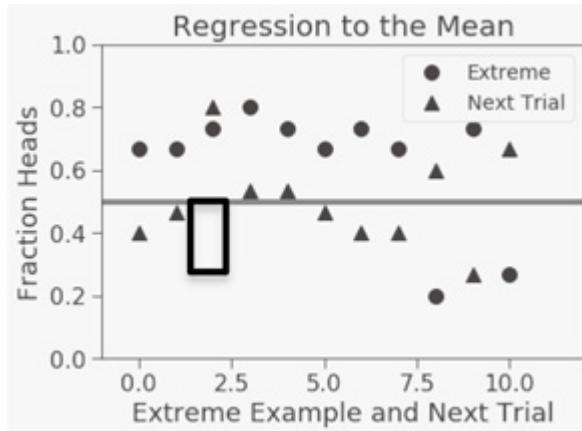
```
def regress_to_mean(num_flips, num_trials):
    #Get fraction of heads for each trial of num_flips
    frac_heads = []
    for t in range(num_trials):
        frac_heads.append(flip(num_flips))
    #Find trials with extreme results and for each the next trial
    extremes, next_trials = [], []
    for i in range(len(frac_heads) - 1):
        if frac_heads[i] < 0.33 or frac_heads[i] > 0.66:
            extremes.append(frac_heads[i])
            next_trials.append(frac_heads[i+1])
    #Plot results
    plt.plot(range(len(extremes)), extremes, 'ko',
             label = 'Extreme')
    plt.plot(range(len(next_trials)), next_trials, 'k^',
             label = 'Next Trial')
    plt.axhline(0.5)
    plt.ylim(0, 1)
    plt.xlim(-1, len(extremes) + 1)
    plt.xlabel('Extreme Example and Next Trial')
    plt.ylabel('Fraction Heads')
    plt.title('Regression to the Mean')
    plt.legend(loc = 'best')

regress_to_mean(15, 50)
```

[Figure 17-3](#) Regression to the mean

Notice that while the trial following an extreme result is typically followed by a trial closer to the mean than the extreme result, that doesn't always occur—as shown by the boxed pair.

**Finger exercise:** Andrea averages 5 strokes a hole when she plays golf. One day, she took 40 strokes to complete the first nine holes. Her partner conjectured that she would probably regress to the mean and take 50 strokes to complete the next nine holes. Do you agree with her partner?



[Figure 17-4](#). Illustration of regression to mean

[Figure 17-5](#) contains a function, `flip_plot`, that produces two plots, [Figure 17-6](#), intended to show the law of large numbers at work. The first plot shows how the absolute value of the difference between the number of heads and number of tails changes with the number of flips. The second plot compares the ratio of heads to tails versus the number of flips. Since the numbers on the x-axis are large, we use `plt.xticks(rotation = 'vertical')` to rotate them.

```

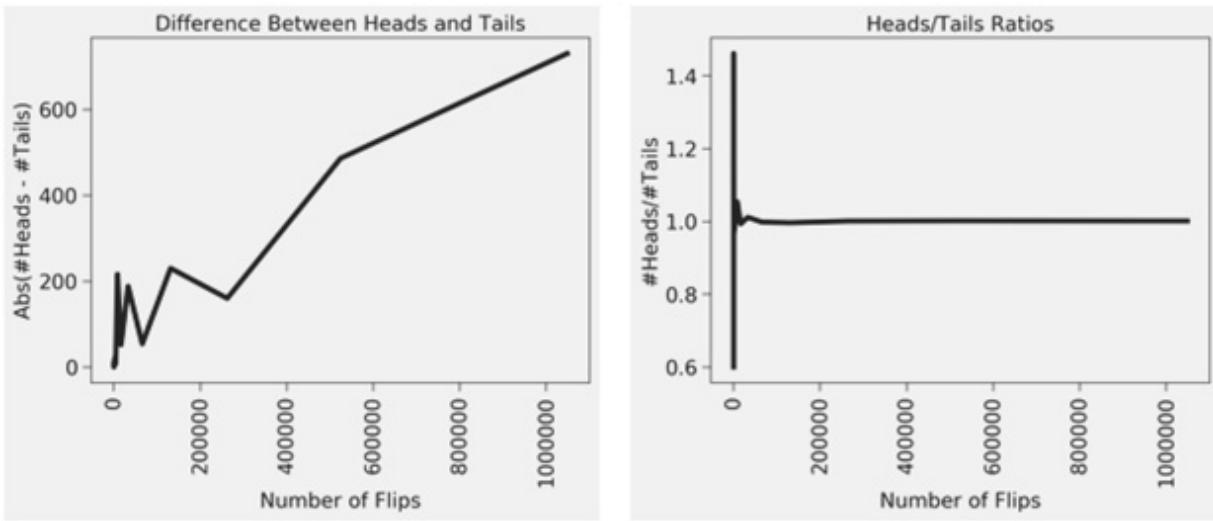
def flip_plot(min_exp, max_exp):
    """Assumes min_exp and max_exp positive ints; min_exp < max_exp
       Plots results of 2**min_exp to 2**max_exp coin flips"""
    ratios, diffs, xAxis = [], [], []
    for exp in range(min_exp, max_exp + 1):
        xAxis.append(2**exp)
        for num_flips in xAxis:
            num_heads = 0
            for n in range(num_flips):
                if random.choice(('H', 'T')) == 'H':
                    num_heads += 1
            num_tails = num_flips - num_heads
            try:
                ratios.append(num_heads/num_tails)
                diffs.append(abs(num_heads - num_tails))
            except ZeroDivisionError:
                continue
    plt.title('Difference Between Heads and Tails')
    plt.xlabel('Number of Flips')
    plt.ylabel('Abs(#Heads - #Tails)')
    plt.xticks(rotation = 'vertical')
    plt.plot(xAxis, diffs, 'k')
    plt.figure()
    plt.title('Heads/Tails Ratios')
    plt.xlabel('Number of Flips')
    plt.ylabel('#Heads/#Tails')
    plt.xticks(rotation = 'vertical')
    plt.plot(xAxis, ratios, 'k')

random.seed(0)
flip_plot(4, 20)

```

[Figure 17-5](#) Plotting the results of coin flips

The line `random.seed(0)` near the bottom of the figure ensures that the pseudorandom number generator used by `random.random` will generate the same sequence of pseudorandom numbers each time this code is executed.<sup>114</sup> This is convenient for debugging. The function `random.seed` can be called with any number. If it is called with no argument, the seed is chosen at random.



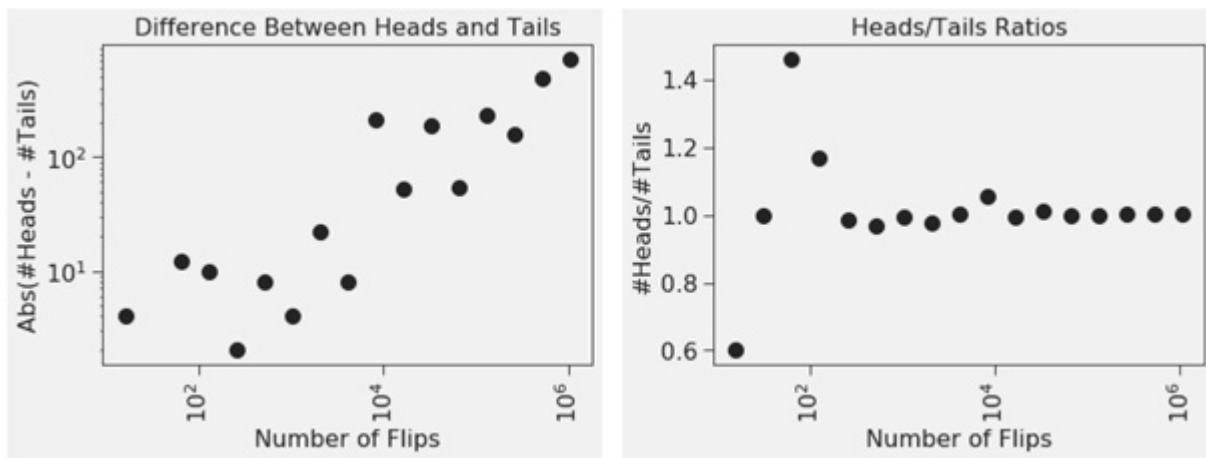
[Figure 17-6](#) The law of large numbers at work

The plot on the left seems to suggest that the absolute difference between the number of heads and the number of tails fluctuates in the beginning, crashes downwards, and then moves rapidly upwards. However, we need to keep in mind that we have only two data points to the right of  $x = 300,000$ . The fact that `plt.plot` connected these points with lines may mislead us into seeing trends when all we have are isolated points. This is not an uncommon phenomenon, so you should always ask how many points a plot actually contains before jumping to any conclusion about what it means.

It's hard to see much of anything in the plot on the right, which is mostly a flat line. This too is deceptive. Even though there are 16 data points, most of them are crowded into a small amount of real estate on the left side of the plot, so that the detail is impossible to see. This occurs because the plotted points have  $x$  values of  $2^4$ ,  $2^5$ ,  $2^6$ , ...,  $2^{20}$ , so the values on the  $x$ -axis range from  $16$  to over a million, and unless instructed otherwise `plot` will place these points based on their relative distance from the origin. This is called **linear scaling**. Because most of the points have  $x$  values that are small relative to  $2^{20}$ , they will appear relatively close to the origin.

Fortunately, these visualization problems are easy to address in Python. As we saw in Chapter 13 and earlier in this chapter, we can easily instruct our program to plot unconnected points, e.g., by writing `plt.plot(xAxis, diffs, 'ko')`.

Both plots in [Figure 17-7](#) use a logarithmic scale on the x-axis. Since the x values generated by `flip_plot` are  $2^{\text{minExp}}$ ,  $2^{\text{minExp}+1}$ , ...,  $2^{\text{maxExp}}$ , using a logarithmic x-axis causes the points to be evenly spaced along the x-axis—providing maximum separation between points. The left plot in [Figure 17-7](#) uses a logarithmic scale on the y-axis as well as on the x-axis. The y values on this plot range from nearly 0 to around 550. If the y-axis were linearly scaled, it would be difficult to see the relatively small differences in y values on the left end of the plot. On the other hand, on the plot on the right, the y values are fairly tightly grouped, so we use a linear y-axis.



[Figure 17-7](#) The law of large numbers at work

**Finger exercise:** Modify the code in [Figure 17-5](#) so that it produces plots like those shown in [Figure 17-7](#).

The plots in [Figure 17-7](#) are easier to interpret than the earlier plots. The plot on the right suggests strongly that the ratio of heads to tails converges to 1.0 as the number of flips gets large. The meaning of the plot on the left is less clear. It appears that the absolute difference grows with the number of flips, but it is not completely convincing.

It is never possible to achieve perfect accuracy through sampling without sampling the entire population. No matter how many samples we examine, we can never be sure that the sample set is typical until we examine every element of the population (and since

we are often dealing with infinite populations, e.g., all possible sequences of coin flips, this is often impossible). Of course, this is not to say that an estimate cannot be precisely correct. We might flip a coin twice, get one heads and one tails, and conclude that the true probability of each is 0.5. We would have reached the right conclusion, but our reasoning would have been faulty.

How many samples do we need to look at before we can have justified confidence in our answer? This depends on the **variance** in the underlying distribution. Roughly speaking, variance is a measure of how much spread there is in the possible different outcomes. More formally, the variance of a collection of values,  $X$ , is defined as

$$\text{variance}(X) = \frac{\sum_{x \in X} (x - \mu)^2}{|X|}$$

where  $|X|$  is the size of the collection and  $\mu$  (mu) its mean. Informally, the variance describes what fraction of the values are close to the mean. If many values are relatively close to the mean, the variance is relatively small. If many values are relatively far from the mean, the variance is relatively large. If all values are the same, the variance is zero.

The **standard deviation** of a collection of values is the square root of the variance. While it contains exactly the same information as the variance, the standard deviation is easier to interpret because it is in the same units as the original data. For example, it is easier to understand the statement “the mean height of a population is 70 inches with a standard deviation of 4 inches,” than the sentence “the mean of height of a population is 70 inches with a variance of 16 inches squared.”

[Figure 17-8](#) contains implementations of variance and standard deviation.<sup>115</sup>

```

def variance(X):
    """Assumes that X is a list of numbers.
       Returns the variance of X"""
    mean = sum(X)/len(X)
    tot = 0.0
    for x in X:
        tot += (x - mean)**2
    return tot/len(X)

def std_dev(X):
    """Assumes that X is a list of numbers.
       Returns the standard deviation of X"""
    return variance(X)**0.5

```

[Figure 17-8](#) Variance and standard deviation

We can use the notion of standard deviation to think about the relationship between the number of samples we have looked at and how much confidence we should have in the answer we have computed. [Figure 17-10](#) contains a modified version of `flip_plot`. It uses the helper function `run_trial` to run multiple trials of each number of coin flips, and then plots the means and standard deviations for the `heads/tails` ratio. The helper function `make_plot`, [Figure 17-9](#), contains the code used to produce the plots.

```

def make_plot(x_vals, y_vals, title, x_label, y_label, style,
              log_x = False, log_y = False):
    plt.figure()
    plt.title(title)
    plt.xlabel(x_label)
    plt.ylabel(y_label)
    plt.plot(x_vals, y_vals, style)
    if log_x:
        plt.semilogx()
    if log_y:
        plt.semilogy()

```

[Figure 17-9](#) Helper function for coin-flipping simulation

```

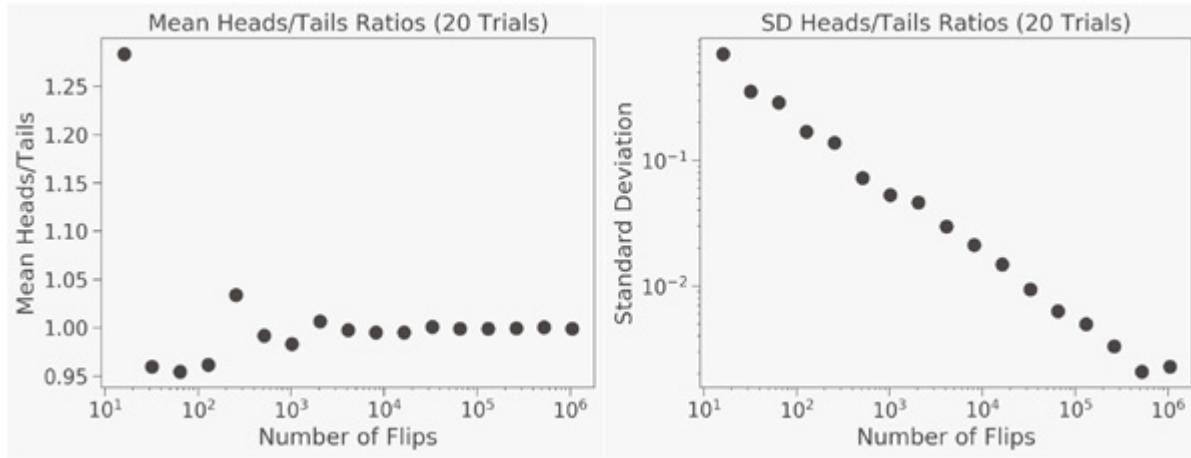
def run_trial(num_flips):
    num_heads = 0
    for n in range(num_flips):
        if random.choice(('H', 'T')) == 'H':
            num_heads += 1
    num_tails = num_flips - num_heads
    return (num_heads, num_tails)

def flip_plot1(min_exp, max_exp, num_trials):
    """Assumes min_exp, max_exp, num_trials ints >0; min_exp < max_exp
       Plots summaries of results of num_trials trials of
       2**min_exp to 2**max_exp coin flips"""
    ratios_means, diffs_means, ratios_SDs, diffs_SDs = [], [], [], []
    x_axis = []
    for exp in range(min_exp, max_exp + 1):
        x_axis.append(2**exp)
    for num_flips in x_axis:
        ratios, diffs = [], []
        for t in range(num_trials):
            num_heads, num_tails = run_trial(num_flips)
            ratios.append(num_heads/num_tails)
            diffs.append(abs(num_heads - num_tails))
        ratios_means.append(sum(ratios)/num_trials)
        diffs_means.append(sum(diffs)/num_trials)
        ratios_SDs.append(std_dev(ratios))
        diffs_SDs.append(std_dev(diffs))
    title = f'Mean Heads/Tails Ratios ({num_trials} Trials)'
    make_plot(x_axis, ratios_means, title, 'Number of Flips',
              'Mean Heads/Tails', 'ko', log_x = True)
    title = f'SD Heads/Tails Ratios ({num_trials} Trials)'
    make_plot(x_axis, ratios_SDs, title, 'Number of Flips',
              'Standard Deviation', 'ko', log_x = True, log_y = True)

```

[Figure 17-10](#) Coin-flipping simulation

Let's try `flip_plot1(4, 20, 20)`. It generates the plots in [Figure 17-11](#).



[Figure 17-11](#) Convergence of heads/tails ratios

This is encouraging. The mean heads/tails ratio is converging towards 1 and the log of the standard deviation is falling linearly with the log of the number of flips per trial. By the time we get to about  $10^6$  coin flips per trial, the standard deviation (about  $10^{-3}$ ) is roughly three decimal orders of magnitude smaller than the mean (about 1), indicating that the variance across the trials was small. We can, therefore, have considerable confidence that the expected heads/tails ratio is quite close to 1.0. As we flip more coins, not only do we have a more precise answer, but more important, we also have reason to be more confident that it is close to the right answer.

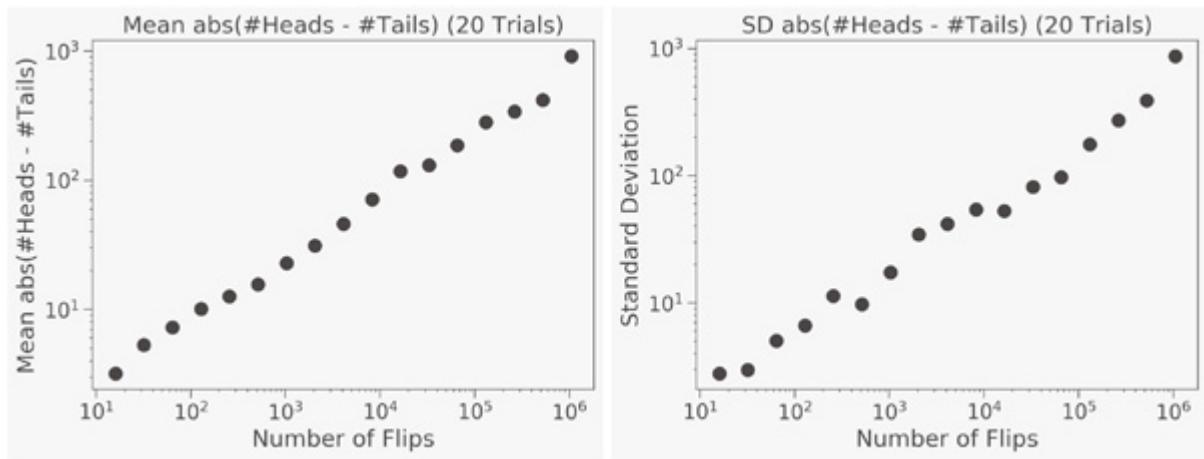
What about the absolute difference between the number of heads and the number of tails? We can take a look at that by adding to the end of `flip_plot1` the code in [Figure 17-12](#). This produces the plots in [Figure 17-13](#).

```

title = f'Mean abs(#Heads - #Tails) ({num_trials} Trials)'
make_plot(x_axis, diff_means, title,
          'Number of Flips', 'Mean abs(#Heads - #Tails)', 'ko',
          log_x = True, log_y = True)
title = 'SD abs(#Heads - #Tails) ({num_trials} Trials)'
make_plot(x_axis, diff_SDs, title,
          'Number of Flips', 'Standard Deviation', 'ko',
          log_x = True, log_y = True)

```

[Figure 17-12](#) Absolute differences



[Figure 17-13](#) Mean and standard deviation of heads - tails

As expected, the absolute difference between the numbers of heads and tails grows with the number of flips. Furthermore, since we are averaging the results over 20 trials, the plot is considerably smoother than when we plotted the results of a single trial in [Figure 17-7](#). But what's up with the plot on the right of [Figure 17-13](#)? The standard deviation is growing with the number of flips. Does this mean that as the number of flips increases we should have less rather than more confidence in the estimate of the expected value of the difference between heads and tails?

No, it does not. The standard deviation should always be viewed in the context of the mean. If the mean were a billion and the standard deviation 100, we would view the dispersion of the data as small. But if the mean were 100 and the standard deviation 100, we would view the dispersion as large.

The **coefficient of variation** is the standard deviation divided by the mean. When comparing data sets with different means (as here), the coefficient of variation is often more informative than the standard deviation. As you can see from its implementation in [Figure 17-14](#), the coefficient of variation is not defined when the mean is 0.

```
def CV(X):
    mean = sum(X)/len(X)
    try:
        return std_dev(X)/mean
    except ZeroDivisionError:
        return float('nan')
```

[Figure 17-14.](#) Coefficient of variation

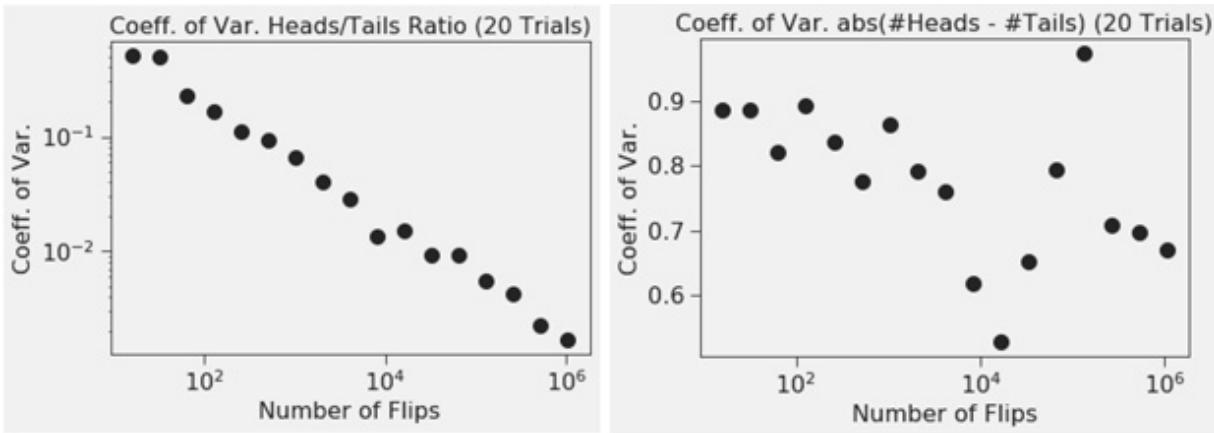
[Figure 17-15](#) contains a function that plots coefficients of variation. In addition to the plots produced by `flip_plot1`, it produces the plots in [Figure 17-16](#).

```

def flip_plot2(min_exp, max_exp, num_trials):
    """Assumes min_exp and max_exp positive ints; min_exp < max_exp
       num_trials a positive integer
       Plots summaries of results of num_trials trials of
       2**min_exp to 2**max_exp coin flips"""
    ratios_means, diffs_means, ratios_SDs, diffs_SDs = [], [], [], []
    ratios_CVs, diffs_CVs, x_axis = [], [], []
    for exp in range(min_exp, max_exp + 1):
        x_axis.append(2**exp)
    for num_flips in x_axis:
        ratios, diffs = [], []
        for t in range(num_trials):
            num_heads, num_tails = run_trial(num_flips)
            ratios.append(num_heads/float(num_tails))
            diffs.append(abs(num_heads - num_tails))
        ratios_means.append(sum(ratios)/num_trials)
        diffs_means.append(sum(diffs)/num_trials)
        ratios_SDs.append(std_dev(ratios))
        diffs_SDs.append(std_dev(diffs))
        ratios_CVs.append(CV(ratios))
        diffs_CVs.append(CV(diffs))
    num_trials_str = ' (' + str(num_trials) + ' Trials)'
    title = f'Mean Heads/Tails Ratios (' + str(num_trials) + ' Trials)'
    make_plot(x_axis, ratios_means, title, 'Number of flips',
              'Mean Heads/Tails', 'ko', log_x = True)
    title = 'SD Heads/Tails Ratios' + num_trials_str
    make_plot(x_axis, ratios_SDs, title, 'Number of flips',
              'Standard Deviation', 'ko', log_x = True, log_y = True)
    title = 'Mean abs(#Heads - #Tails)' + num_trials_str
    make_plot(x_axis, diffs_means, title, 'Number of Flips',
              'Mean abs(#Heads - #Tails)', 'ko',
              log_x = True, log_y = True)
    title = 'SD abs(#Heads - #Tails)' + num_trials_str
    make_plot(x_axis, diffs_SDs, title, 'Number of Flips',
              'Standard Deviation', 'ko', log_x = True, log_y = True)
    title = 'Coeff. of Var. abs(#Heads - #Tails)' + num_trials_str
    make_plot(x_axis, diffs_CVs, title, 'Number of Flips',
              'Coeff. of Var.', 'ko', log_x = True)
    title = 'Coeff. of Var. Heads/Tails Ratio' + num_trials_str
    make_plot(x_axis, ratios_CVs, title, 'Number of Flips',
              'Coeff. of Var.', 'ko', log_x = True, log_y = True)

```

[Figure 17-15](#) Final version of `flip_plot`

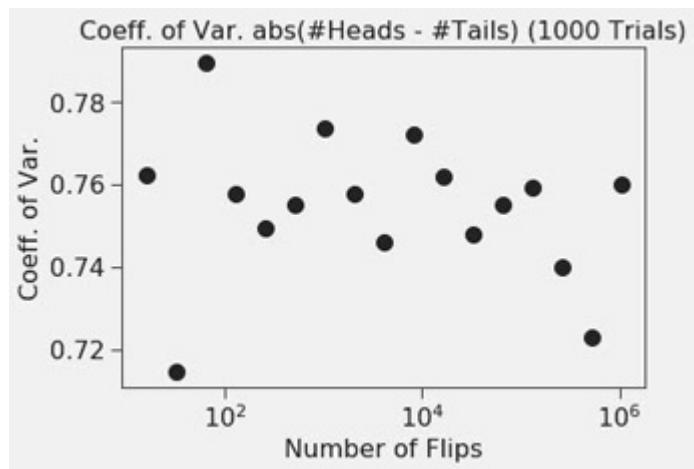


[Figure 17-16](#) Coefficient of variation of heads/tails and  $\text{abs}(\text{heads} - \text{tails})$

In this case we see that the plot of coefficient of variation for the heads/tails ratio is not much different from the plot of the standard deviation in [Figure 17-11](#). This is not surprising, since the only difference between the two is the division by the mean, and since the mean is close to 1, that makes little difference.

The plot of the coefficient of variation for the absolute difference between heads and tails is a different story. While the standard deviation exhibited a clear trend in [Figure 17-13](#), it would take a brave person to argue that the coefficient of variation is trending in any direction. It seems to be fluctuating wildly. This suggests that dispersion in the values of  $\text{abs}(\text{heads} - \text{tails})$  is independent of the number of flips. It's not growing, as the standard deviation might have misled us to believe, but it's not shrinking either. Perhaps a trend would appear if we tried 1000 trials instead of 20. Let's see.

In [Figure 17-17](#), it looks as if the coefficient of variation settles in somewhere in the neighborhood of 0.73-0.76. In general, distributions with a coefficient of variation of less than 1 are considered low-variance.



[Figure 17-17](#) A large number of trials

The main advantage of the coefficient of variation over the standard deviation is that it allows us to compare the dispersion of sets with different means. Consider, for example, the distribution of weekly income in different regions of Australia, as depicted in [Figure 17-18](#).



[Figure 17-18](#) Income distribution in Australia

If we use standard deviation as a measure of income inequality, it appears that there is considerably less income inequality in Tasmania than in the ACT (Australian Capital Territory). However, if we look at the coefficients of variation (about 0.32 for ACT and 0.42 for Tasmania), we reach a rather different conclusion.

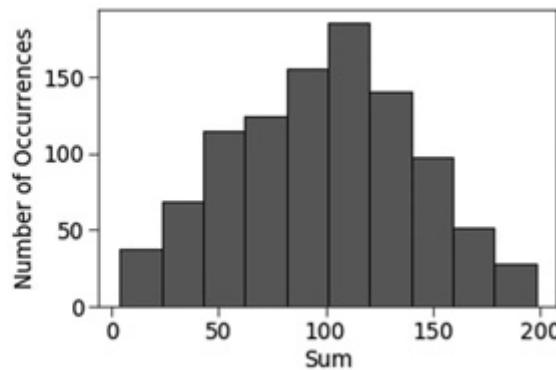
That isn't to say that the coefficient of variation is always more useful than the standard deviation. If the mean is near 0, small changes in the mean lead to large (but not necessarily meaningful) changes in the coefficient of variation, and when the mean is 0, the coefficient of variation is undefined. Also, as we shall see in Section 17.4.2, the standard deviation can be used to construct a confidence interval, but the coefficient of variation cannot.

---

## 17.4 Distributions

A **histogram** is a plot designed to show the distribution of values in a set of data. The values are first sorted, and then divided into a fixed number of equal-width bins. A plot is then drawn that shows the number of elements in each bin. The code on the left of [Figure 17-19](#). produces the plot on the right of that figure.

```
random.seed(0)
vals = []
for i in range(1000):
    num1 = random.choice(range(0, 101))
    num2 = random.choice(range(0, 101))
    vals.append(num1 + num2)
plt.hist(vals, bins = 10, ec = 'k')
plt.xlabel('Sum')
plt.ylabel('Number of Occurrences')
```



[Figure 17-19](#) Code and the histogram it generates

The function call `plt.hist(vals, bins = 10, ec = 'k')` produces a histogram with 10 bins and a black line separating the bars. Matplotlib has automatically chosen the width of each bin based on the number of bins and the range of values. Looking at the code, we know that the smallest number that might appear in `vals` is 0 and the largest number 200. Therefore, the possible values on the x-axis range from 0 to 200. Each bin represents an equal fraction of the values on the x-axis, so the first bin will contain the elements 0-19, the next bin the elements 20-39, etc.

**Finger exercise:** In [Figure 17-19](#), why are the bins near the middle of the histogram taller than the bins near the sides? Hint: think about why 7 is the most common outcome of rolling a pair of dice.

By now you must be getting awfully bored with flipping coins. Nevertheless, we are going to ask you to look at yet one more coin-flipping simulation. The simulation in [Figure 17-20](#) illustrates more of Matplotlib's plotting capabilities and gives us an opportunity to get a visual notion of what standard deviation means. It produces two histograms. The first shows the result of a simulation of 100,000

trials of 100 flips of a fair coin. The second shows the result of a simulation of 100,000 trials of 1,000 flips of a fair coin.

The method `plt.annotate` is used to place some statistics on the figure showing the histogram. The first argument is the string to be displayed on the figure. The next two arguments control where the string is placed. The argument `xycoords = 'axes fraction'` indicates the placement of the text will be expressed as a fraction of the width and height of the figure. The argument `xy = (0.67, 0.5)` indicates that the text should begin two-thirds of the way from the left edge of the figure and halfway from the bottom edge of the figure.

```

def flip(num_flips):
    """Assumes num_flips a positive int"""
    heads = 0
    for i in range(num_flips):
        if random.choice(('H', 'T')) == 'H':
            heads += 1
    return heads/float(num_flips)

def flip_sim(num_flips_per_trial, num_trials):
    frac_heads = []
    for i in range(num_trials):
        frac_heads.append(flip(num_flips_per_trial))
    mean = sum(frac_heads)/len(frac_heads)
    sd = std_dev(frac_heads)
    return (frac_heads, mean, sd)

def label_plot(num_flips, num_trials, mean, sd):
    plt.title(str(num_trials) + ' trials of '
              + str(num_flips) + ' flips each')
    plt.xlabel('Fraction of Heads')
    plt.ylabel('Number of Trials')
    plt.annotate('Mean = ' + str(round(mean, 4))
                 + '\nSD = ' + str(round(sd, 4)), size='x-large',
                 xycoords = 'axes fraction', xy = (0.67, 0.5))

def make_plots(num_flips1, num_flips2, num_trials):
    val1, mean1, sd1 = flip_sim(num_flips1, num_trials)
    plt.hist(val1, bins = 20)
    x_min, x_max = plt.xlim()
    label_plot(num_flips1, num_trials, mean1, sd1)
    plt.figure()
    val2, mean2, sd2 = flip_sim(num_flips2, num_trials)
    plt.hist(val2, bins = 20, ec = 'k')
    plt.xlim(x_min, x_max)
    label_plot(num_flips2, num_trials, mean2, sd2)

make_plots(100, 1000, 100000)

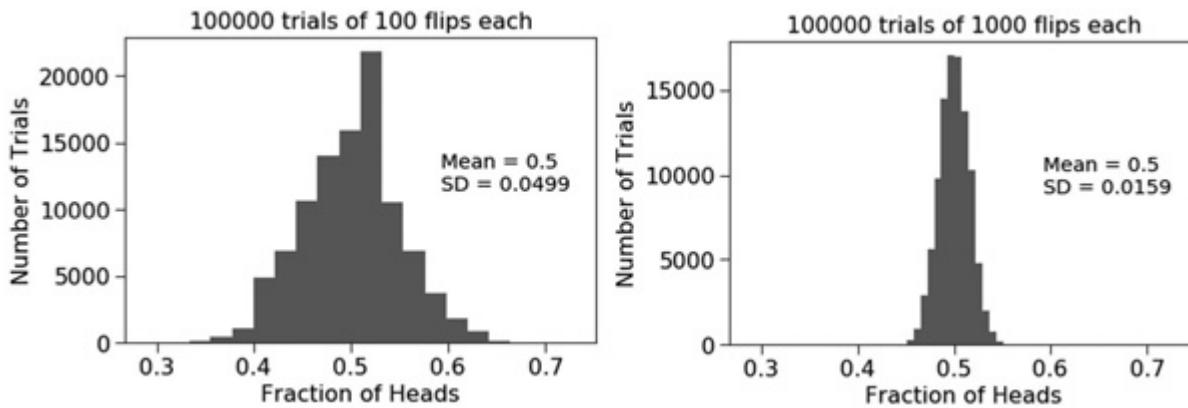
```

[Figure 17-20](#) Plot histograms of coin flips

To facilitate comparing the two figures, we have used `plt.xlim` to force the bounds of the x-axis in the second plot to match those in the first plot, rather than letting Matplotlib choose the bounds.

When the code in [Figure 17-20](#) is run, it produces the plots in [Figure 17-21](#). Notice that while the means in both plots are about the same, the standard deviations are quite different. The spread of

outcomes is much tighter when we flip the coin 1000 times per trial than when we flip the coin 100 times per trial.



[Figure 17-21](#) Histograms of coin flips

#### 17.4.1 Probability Distributions

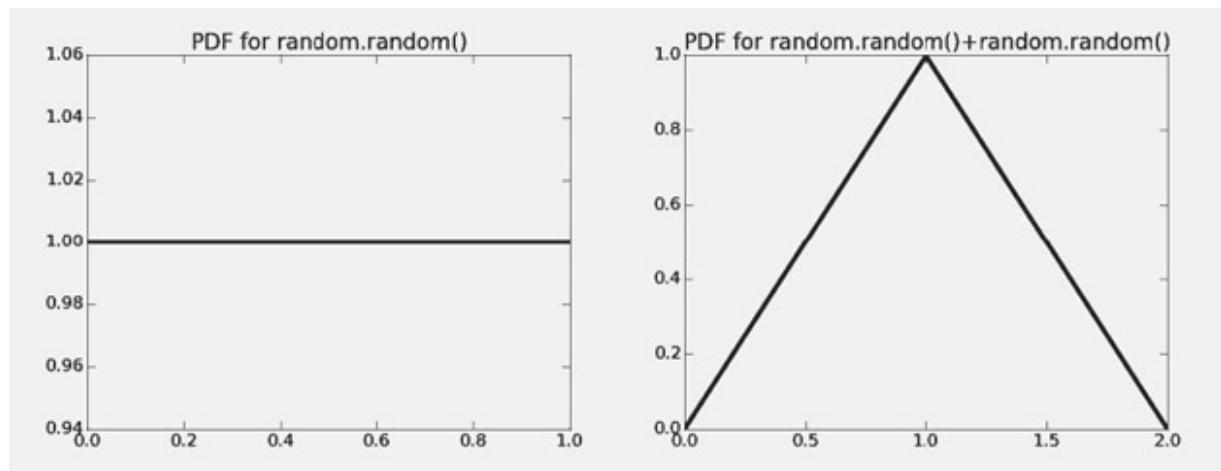
A histogram is a depiction of a **frequency distribution**. It tells us how often a random variable has taken on a value in some range, e.g., how often the fraction of times a coin came up heads was between 0.4 and 0.5. It also provides information about the relative frequency of various ranges. For example, we can easily see that the fraction of heads falls between 0.4 and 0.5 far more frequently than it falls between 0.3 and 0.4. A **probability distribution** captures the notion of relative frequency by giving the probability of a random value taking on a value within a range.

Probability distributions fall into two groups: discrete probability distributions and continuous probability distributions, depending upon whether they define the probability distribution for a discrete or a continuous random variable. A **discrete random variable** can take on one of a finite set of values, e.g., the values associated with a roll of a die. A **continuous random variable** can take on any of the infinite real values between two real numbers, e.g., the speed of a car traveling between 0 miles per hour and the car's maximum speed.

**Discrete probability distributions** are easy to describe. Since there are a finite number of values that the variable can take

on, the distribution can be described by simply listing the probability of each value.

**Continuous probability distributions** are trickier. Since there are an infinite number of possible values, the probability that a continuous random variable will take on a specific value is usually close to 0. For example, the probability that a car is travelling at exactly 81.3457283 miles per hour is almost 0. Mathematicians like to describe continuous probability distributions using a **probability density function**, often abbreviated as **PDF**. A PDF describes the probability of a random variable lying between two values. Think of the PDF as defining a curve where the values on the x-axis lie between the minimum and maximum value of the random variable. (In some cases the x-axis is infinitely long.) Under the assumption that  $x_1$  and  $x_2$  lie in the domain of the random variable, the probability of the variable having a value between  $x_1$  and  $x_2$  is the area under the curve between  $x_1$  and  $x_2$ . [Figure 17-22](#) shows the probability density functions for the expressions `random.random()` and `random.random() + random.random()`.



[Figure 17-22](#) PDF for `random.random()`

For `random.random()`, the area under the curve from 0 to 1 is 1. This makes sense because we know that the probability of `random.random()` returning a value between 0 and 1 is 1. If we consider the area under the part of the curve for `random.random()`

between 0.2 and 0.4, it is  $1.0 * 0.2 = 0.2$ —indicating that the probability of `random.random()` returning a value between 0.2 and 0.4 is 0.2. Similarly, the area under the curve for `random.random() + random.random()` from 0 to 2 is 1, and the area under the curve from 0 to 1 is 0.5. Notice, by the way, that the PDF for `random.random()` indicates that every possible interval of the same length has the same probability, whereas the PDF for `random.random() + random.random()` indicates that some ranges of values are more probable than others.

#### 17.4.2 Normal Distributions

A **normal** (or **Gaussian**) **distribution** is defined by the probability density function in [Figure 17-23](#).

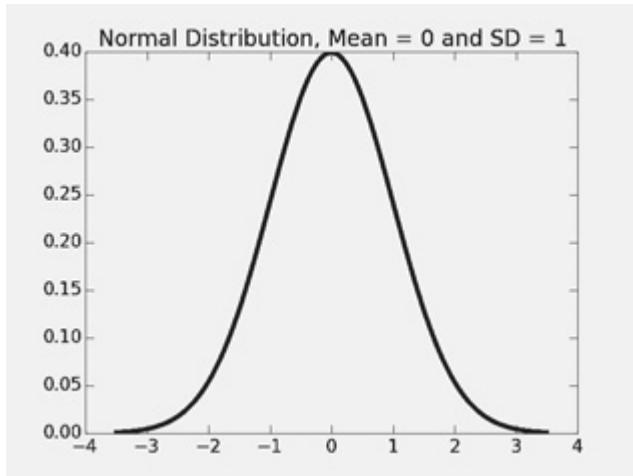
$$P(x) = \frac{1}{\sigma\sqrt{2\pi}} * e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

[Figure 17-23](#) PDF for Gaussian distribution

where  $\mu$  is the mean,  $\sigma$  the standard deviation, and  $e$  is Euler's number (roughly 2.718). [116](#)

If you don't feel like studying this equation, that's fine. Just remember that normal distributions peak at the mean, fall off symmetrically above and below the mean, and asymptotically approach 0. They have the nice mathematical property of being completely specified by two parameters: the mean and the standard deviation (the only two parameters in the equation). Knowing these is equivalent to knowing the entire distribution. The shape of the normal distribution resembles (in the eyes of some) that of a bell, so it sometimes is referred to as a **bell curve**.

[Figure 17-24](#) shows part of the PDF for a normal distribution with a mean of 0 and a standard deviation of 1. We can only show a portion of the PDF because the tails of a normal distribution converge towards 0, but don't reach it. In principle, no value has a zero probability of occurring.



[Figure 17-24](#). A normal distribution

Normal distributions can be easily generated in Python programs by calling `random.gauss(mu, sigma)`, which returns a randomly chosen floating-point number from a normal distribution with mean and standard deviation `sigma`.

Normal distributions are frequently used in constructing probabilistic models because they have nice mathematical properties. Of course, finding a mathematically nice model is of no use if it provides a bad model of the actual data. Fortunately, many random variables have an approximately normal distribution. For example, physical properties of populations of plants and animals (e.g., height, weight, body temperature) typically have approximately normal distributions. Importantly, many experiments have normally distributed measurement errors. This assumption was used in the early 1800s by the German mathematician and physicist Karl Gauss, who assumed a normal distribution of measurement errors in his analysis of astronomical data (which led to the normal distribution becoming known as the Gaussian distribution in much of the scientific community).

One of the nice properties of normal distributions is that independent of the mean and standard deviation, the number of standard deviations from the mean needed to encompass a fixed fraction of the data is a constant. For example,  $\sim 68.27\%$  of the data will always lie within one standard deviation of the mean,  $\sim 95.45\%$  within two standard deviations of the mean, and  $\sim 99.73\%$  within

three standard deviations of the mean. This is sometimes called the **68-95-99.7 rule**, but is more often called the **empirical rule**.

The rule can be derived by integrating the formula defining a normal distribution to get the area under the curve. Looking at [Figure 17-24](#), it is easy to believe that roughly two-thirds of the total area under the curve lies between  $-1$  and  $1$ , roughly 95% between  $-2$  and  $2$ , and almost all of it between  $-3$  and  $3$ . But that's only one example, and it is always dangerous to generalize from a single example. We could accept the empirical rule on the unimpeachable authority of Wikipedia. However, just to be sure, and as an excuse to introduce a Python library worth knowing about, let's check it ourselves.

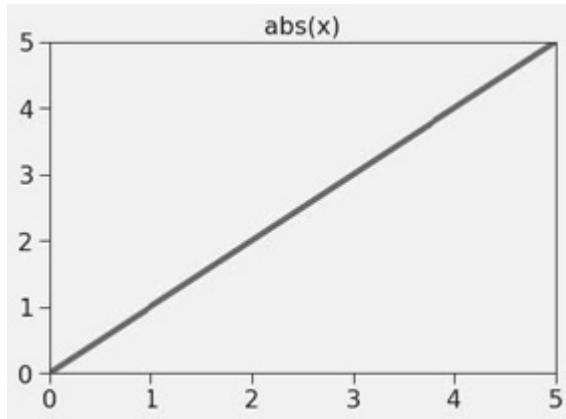
The **SciPy** library contains many mathematical functions commonly used by scientists and engineers. SciPy is organized into modules covering different scientific computing domains, such as signal processing and image processing. We will use a number of functions from SciPy later in this book. Here we use the function `scipy.integrate.quad`, which finds an approximation to the value of integrating a function between two points.

The function `scipy.integrate.quad` has three required parameters and one optional parameter

- A function or method to be integrated (if the function takes more than one argument, it is integrated along the axis corresponding to the first argument)
- A number representing the lower limit of the integration
- A number representing the upper limit of the integration
- An optional tuple supplying values for all arguments, except the first, of the function to be integrated

The `quad` function returns a tuple of two floating-point numbers. The first approximates the value of the integral, and the second estimates the absolute error in the result.

Consider, for example, evaluating the integral of the unary function `abs` in the interval  $0$  to  $5$ , as pictured in [Figure 17-25](#)



[Figure 17-25](#) Plot of absolute value of x

We don't need any fancy math to compute the area under this curve: it's simply the area of a right triangle with base and altitude of length 5, i.e., 12.5. So, it shouldn't be a surprise that

```
print(scipy.integrate.quad(abs, 0, 5)[0])
```

prints 12.5. (The second value in the tuple returned by `quad` is roughly  $10^{-13}$ , indicating that the approximation is quite good.)

The code in [Figure 17-26](#) computes the area under portions of normal distributions for some randomly chosen means and standard deviations. The ternary function `gaussian` evaluates the formula in [Figure 17-23](#) for a Gaussian distribution with mean `mu` and standard deviation `sigma` at the point `x`. For example,

```
for x in range(-2, 3):
    print(gaussian(x, 0, 1))
prints

0.05399096651318806
0.24197072451914337
0.3989422804014327
0.24197072451914337
0.05399096651318806
```

So, to find the area between `min_val` and `max_val` under a Gaussian with mean `mu` and standard deviation `sigma`, we need only evaluate

```
scipy.integrate.quad(gaussian, min_val, max_val, (mu,  
sigma))[0])
```

**Finger exercise:** Find the area between -1 and 1 for a standard normal distribution.

```
import scipy.integrate  
  
def gaussian(x, mu, sigma):  
    """assumes x, mu, sigma numbers  
    returns the value of P(x) for a Gaussian  
    with mean mu and sd sigma"""  
    factor1 = (1.0/(sigma*((2*np.pi)**0.5)))  
    factor2 = np.e**-(((x-mu)**2)/(2*sigma**2))  
    return factor1*factor2  
  
def check_empirical(mu_max, sigma_max, num_trials):  
    """assumes mu_max, sigma_max, num_trials positive ints  
    prints fraction of values of a Gaussians (with randomly  
    chosen mu and sigma) falling within 1, 2, 3 standard  
    deviations"""  
    for t in range(num_trials):  
        mu = random.randint(-mu_max, mu_max + 1)  
        sigma = random.randint(1, sigma_max)  
        print('For mu =', mu, 'and sigma =', sigma)  
        for num_std in (1, 2, 3):  
            area = scipy.integrate.quad(gaussian, mu-num_std*sigma,  
                                         mu+num_std*sigma,  
                                         (mu, sigma))[0]  
            print(' Fraction within', num_std, 'std =',  
                  round(area, 4))  
  
check_empirical(10, 10, 3)
```

[Figure 17-26](#) Checking the empirical rule

When we ran the code in [Figure 17-26](#), it printed what the empirical rule predicts:

```
For mu = 2 and sigma = 7  
Fraction within 1 std = 0.6827  
Fraction within 2 std = 0.9545  
Fraction within 3 std = 0.9973  
For mu = -9 and sigma = 5  
Fraction within 1 std = 0.6827
```

```

Fraction within 2 std = 0.9545
Fraction within 3 std = 0.9973
For mu = 6 and sigma = 8
Fraction within 1 std = 0.6827
Fraction within 2 std = 0.9545
Fraction within 3 std = 0.9973

```

People frequently use the empirical rule to derive confidence intervals. Instead of estimating an unknown value (e.g., the expected number of heads) by a single value, a **confidence interval** provides a range that is likely to contain the unknown value and a degree of confidence that the unknown value lies within that range. For example, a political poll might indicate that a candidate is likely to get 52% of the vote  $\pm 4\%$  (i.e., the confidence interval is of size 8) with a **confidence level** of 95%. What this means is that the pollster believes that 95% of the time the candidate will receive between 48% and 56% of the vote.<sup>117</sup> Together the confidence interval and the confidence level are intended to indicate the reliability of the estimate.<sup>118</sup> Almost always, increasing the confidence level will require widening the confidence interval.

Suppose that we run 100 trials of 100 coin flips each. Suppose further that the mean fraction of heads is 0.4999 and the standard deviation 0.0497. For reasons we will discuss in Section 19.2, we can assume that the distribution of the means of the trials was normal. Therefore, we can conclude that if we conducted more trials of 100 flips each,

- ~95% of the time the fraction of heads will be  $0.4999 \pm 0.0994$  and
- >99% of the time the fraction of heads will be  $0.4999 \pm 0.1491$ .

It is often useful to visualize confidence intervals using **error bars**. The function `show_error_bars` in [Figure 17-27](#) calls the version of `flip_sim` in [Figure 17-20](#) and then uses

```
plt.errorbar(xVals, means, yerr = 1.96*plt.array(sds))
```

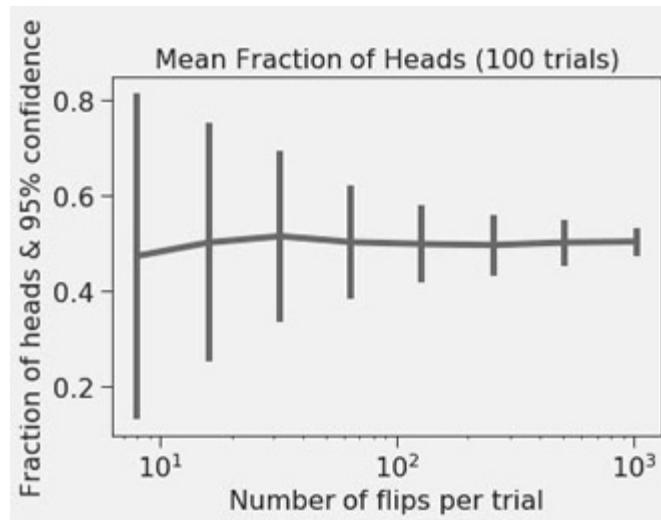
to produce a plot. The first two arguments give the x and y values to be plotted. The third argument says that the values in `sds` should be multiplied by 1.96 and used to create vertical error bars. We multiply

by 1.96 because 95% of the data in a normal distribution falls within 1.96 standard deviations of the mean.

```
def show_error_bars(min_exp, max_exp, num_trials):
    """Assumes min_exp and max_exp positive ints; min_exp < max_exp
       num_trials a positive integer
       Plots mean fraction of heads with error bars"""
    means, sds, x_vals = [], [], []
    for exp in range(min_exp, max_exp + 1):
        x_vals.append(2**exp)
        frac_heads, mean, sd = flip_sim(2**exp, num_trials)
        means.append(mean)
        sds.append(sd)
    plt.errorbar(x_vals, means, yerr=1.96*np.array(sds))
    plt.semilogx()
    plt.title('Mean Fraction of Heads (' +
              + str(num_trials) + ' trials)')
    plt.xlabel('Number of flips per trial')
    plt.ylabel('Fraction of heads & 95% confidence')
```

[Figure 17-27](#) Produce plot with error bars

The call `show_error_bars(3, 10, 100)` produces the plot in [Figure 17-28](#). Unsurprisingly, the error bars shrink (the standard deviation gets smaller) as the number of flips per trial grows.



[Figure 17-28](#) Estimates with error bars

Error bars provide a lot of useful information. When practical, they should always be provided on plots. Experienced users of statistical data are justifiably suspicious when they are not.

### 17.4.3 Continuous and Discrete Uniform Distributions

Imagine that you take a bus that arrives at your stop every 15 minutes. If you make no effort to time your arrival at the stop to correspond to the bus schedule, your expected waiting time is uniformly distributed between 0 and 15 minutes.

A uniform distribution can be either discrete or continuous. A **continuous uniform distribution**, also called a **rectangular distribution**, has the property that all intervals of the same length have the same probability. Consider the function `random.random`. As we saw in Section 17.4.1, the area under the PDF for any interval of a given length is the same. For example, the area under the curve between 0.23 and 0.33 is the same as the area under the curve between 0.53 and 0.63.

A continuous uniform distribution is fully characterized by a single parameter, its range (i.e., minimum and maximum values). If the range of possible values is from *min* to *max*, the probability of a value falling in the range *x* to *y* is given by

$$P(x, y) = \begin{cases} \frac{y - x}{\max - \min} & \text{if } x \geq \min \text{ and } y \leq \max \\ 0 & \text{otherwise} \end{cases}$$

Elements drawn from a continuous uniform distribution can be generated by calling `random.uniform(min, max)`, which returns a randomly chosen floating-point number between *min* and *max*.

In **discrete uniform distributions** each possible value is equally likely to occur, but the space of possible values is not continuous. For example, when a fair die is rolled, each of the six possible values is equally probable, but the outcomes are not uniformly distributed over the real numbers between 1 and 6—most values, e.g., 2.5, have a probability of 0 and a few values, e.g. 3, have

a probability of  $\frac{1}{6}$ . One can fully characterize a discrete uniform distribution by

$$P(x) = \begin{cases} \frac{1}{|S|} & \text{if } x \in S \\ 0 & \text{otherwise} \end{cases}$$

where  $S$  is the set of possible values and  $|S|$  the number of elements in  $S$ .

#### 17.4.4 Binomial and Multinomial Distributions

Random variables that can take on only a discrete set of values are called **categorical** (also **nominal** or **discrete**) **variables**.

When a categorical variable has only two possible values (e.g., success or failure), the probability distribution is called a **binomial distribution**. One way to think about a binomial distribution is as the probability of a test succeeding exactly  $k$  times in  $n$  independent trials. If the probability of a success in a single trial is  $p$ , the probability of exactly  $k$  successes in  $n$  independent trials is given by the formula

$$\binom{n}{k} * p^k * (1 - p)^{n-k}$$

where

$$\binom{n}{k} = \frac{n!}{k! * (n - k)!}$$

The formula  $\binom{n}{k}$  is known as the **binomial coefficient**. One way to read it is as “ $n$  choose  $k$ ,” since it is equivalent to the number of

subsets of size  $k$  that can be constructed from a set of size  $n$ . For example, there are

$$\binom{4}{2} = \frac{4!}{2! * 2!} = \frac{24}{4} = 6$$

subsets of size two that can be constructed from the set  $\{1,2,3,4\}$ .

In Section 17.2, we asked about the probability of rolling exactly two 1's in 10 rolls of a die. We now have the tools in hand to calculate this probability. Think of the 10 rolls as 10 independent trials, where the trial is a success if a 1 is rolled and a failure otherwise. A trial in which there are exactly two possible outcomes for each repetition of the experiment (success and failure in this case) is called a **Bernoulli trial**.

The binomial distribution tells us that the probability of having exactly two successful trials out of ten is

Total # pairs \* prob that two trials are successful \* probability that 8 trials fail

$$\binom{10}{2} * \left(\frac{1}{6}\right)^2 * \left(\frac{5}{6}\right)^8 = 45 * \frac{1}{36} * \frac{390625}{1679616} \approx 0.291$$

That is to say, the total number of trials multiplied by the product of the fraction of trials that succeed twice and fail eight times.

**Finger exercise:** Use the above formula to implement a function that calculates the probability of rolling exactly two 3's in  $k$  rolls of a fair die. Use this function to plot the probability as  $k$  varies from 2 to 100.

The **multinomial distribution** is a generalization of the binomial distribution to categorical data with more than two possible values. It applies when there are  $n$  independent trials each of which has  $m$  mutually exclusive outcomes, with each outcome having a fixed probability of occurring. The multinomial distribution gives the

probability of any given combination of numbers of occurrences of the various categories.

### 17.4.5 Exponential and Geometric Distributions

**Exponential distributions** occur quite commonly. They are often used to model inter-arrival times, e.g., of cars entering a highway or requests for a webpage.

Consider, for example, the concentration of a drug in the human body. Assume that at each time step each molecule has a constant probability  $p$  of being cleared (i.e., of no longer being in the body). The system is **memoryless** in the sense that at each time step, the probability of a molecule being cleared is independent of what happened at previous times. At time  $t = 0$ , the probability of an individual molecule still being in the body is 1. At time  $t = 1$ , the probability of that molecule still being in the body is  $1 - p$ . At time  $t = 2$ , the probability of that molecule still being in the body is  $(1 - p)^2$ . More generally, at time  $t$ , the probability of an individual molecule having survived is  $(1 - p)^t$ , i.e., it is exponential in  $t$ .

Suppose that at time  $t_0$  there are  $M_0$  molecules of the drug. In general, at time  $t$ , the number of molecules will be  $M_0$  multiplied by the probability that an individual module has survived to time  $t$ . The function `clear` implemented in [Figure 17-29](#) plots the expected number of remaining molecules versus time.

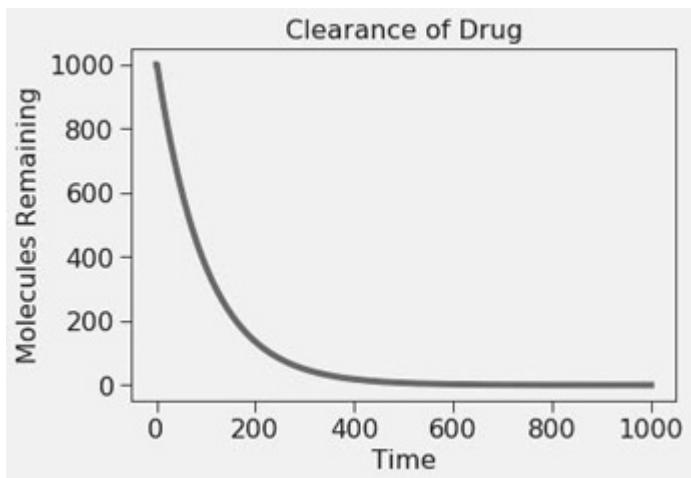
```

def flip_plot2(min_exp, max_exp, num_trials):
    """Assumes min_exp and max_exp positive ints; min_exp < max_exp
       num_trials a positive integer
       Plots summaries of results of num_trials trials of
       2**min_exp to 2**max_exp coin flips"""
    ratios_means, diffs_means, ratios_SDs, diffs_SDs = [], [], [], []
    ratios_CVs, diffs_CVs, x_axis = [], [], []
    for exp in range(min_exp, max_exp + 1):
        x_axis.append(2**exp)
    for num_flips in x_axis:
        ratios, diffs = [], []
        for t in range(num_trials):
            num_heads, num_tails = run_trial(num_flips)
            ratios.append(num_heads/float(num_tails))
            diffs.append(abs(num_heads - num_tails))
        ratios_means.append(sum(ratios)/num_trials)
        diffs_means.append(sum(diffs)/num_trials)
        ratios_SDs.append(std_dev(ratios))
        diffs_SDs.append(std_dev(diffs))
        ratios_CVs.append(CV(ratios))
        diffs_CVs.append(CV(diffs))
    num_trials_str = ' (' + str(num_trials) + ' Trials)'
    title = f'Mean Heads/Tails Ratios (' + str(num_trials) + ' Trials)'
    make_plot(x_axis, ratios_means, title, 'Number of flips',
              'Mean Heads/Tails', 'ko', log_x = True)
    title = 'SD Heads/Tails Ratios' + num_trials_str
    make_plot(x_axis, ratios_SDs, title, 'Number of flips',
              'Standard Deviation', 'ko', log_x = True, log_y = True)
    title = 'Mean abs(#Heads - #Tails)' + num_trials_str
    make_plot(x_axis, diffs_means, title, 'Number of Flips',
              'Mean abs(#Heads - #Tails)', 'ko',
              log_x = True, log_y = True)
    title = 'SD abs(#Heads - #Tails)' + num_trials_str
    make_plot(x_axis, diffs_SDs, title, 'Number of Flips',
              'Standard Deviation', 'ko', log_x = True, log_y = True)
    title = 'Coeff. of Var. abs(#Heads - #Tails)' + num_trials_str
    make_plot(x_axis, diffs_CVs, title, 'Number of Flips',
              'Coeff. of Var.', 'ko', log_x = True)
    title = 'Coeff. of Var. Heads/Tails Ratio' + num_trials_str
    make_plot(x_axis, ratios_CVs, title, 'Number of Flips',
              'Coeff. of Var.', 'ko', log_x = True, log_y = True)

```

[Figure 17-29](#) Exponential clearance of molecules

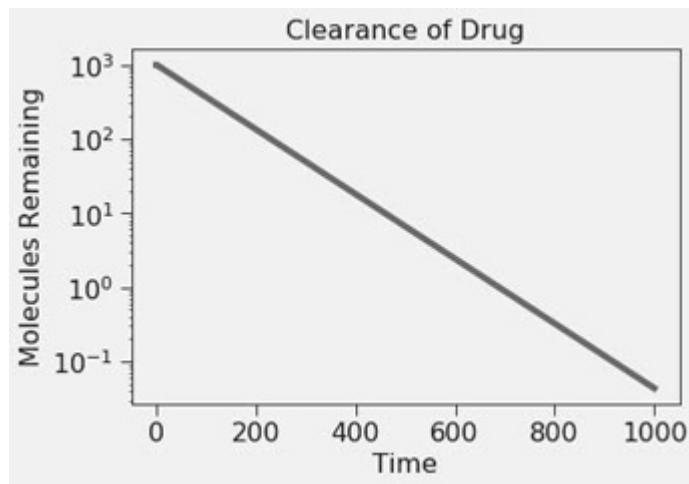
The call `clear(1000, 0.01, 1000)` produces the plot in [Figure 17-30](#).



[Figure 17-30](#) Exponential decay

This is an example of **exponential decay**. In practice, exponential decay is often talked about in terms of **half-life**, i.e., the expected time required for the initial value to decay by 50%. We can also talk about the half-life of a single item. For example, the half-life of a single molecule is the time at which the probability of that molecule having been cleared is 0.5. Notice that as time increases, the number of remaining molecules approaches 0. But it will never quite get there. This should not be interpreted as suggesting that a fraction of a molecule remains. Rather it should be interpreted as saying that since the system is probabilistic, we can never guarantee that all of the molecules have been cleared.

What happens if we make the y-axis logarithmic (by using `plt.semilogy`)? We get the plot in [Figure 17-31](#). In the plot in [Figure 17-30](#), the values on the y-axis are changing exponentially quickly relative to the values on the x-axis. If we make the y-axis itself change exponentially quickly, we get a straight line. The slope of that line is the **rate of decay**.



[Figure 17-31](#) Plotting exponential decay with a logarithmic axis

**Exponential growth** is the inverse of exponential decay. It too is commonly seen in nature. Compound interest, the growth of algae in a swimming pool, and the chain reaction in an atomic bomb are all examples of exponential growth.

Exponential distributions can easily be generated in Python by calling the function `random.expovariate(lambd)`,<sup>119</sup> where `lambd` is `1.0` divided by the desired mean. The function returns a value between `0` and positive infinity if `lambd` is positive, and between negative infinity and `0` if `lambd` is negative.

The **geometric distribution** is the discrete analog of the exponential distribution.<sup>120</sup> It is usually thought of as describing the number of independent attempts required to achieve a first success (or a first failure). Imagine, for example, that you have a balky car that starts only half of the time you turn the key (or push the starter button). A geometric distribution could be used to characterize the expected number of times you would have to attempt to start the car before being successful. This is illustrated by the histogram in [Figure 17-33](#), which was produced by the code in [Figure 17-32](#).

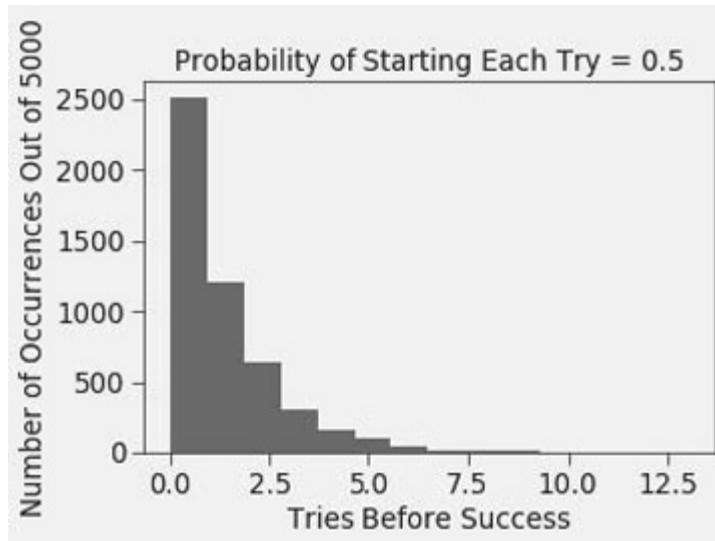
```

def successful_starts(success_prob, num_trials):
    """Assumes success_prob is a float representing probability of a
       single attempt being successful. num_trials a positive int
       Returns a list of the number of attempts needed before a
       success for each trial."""
    tries_before_success = []
    for t in range(num_trials):
        consec_failures = 0
        while random.random() > success_prob:
            consec_failures += 1
        tries_before_success.append(consec_failures)
    return tries_before_success

prob_of_success = 0.5
num_trials = 5000
distribution = successful_starts(prob_of_success, num_trials)
plt.hist(distribution, bins = 14)
plt.xlabel('Tries Before Success')
plt.ylabel('Number of Occurrences Out of ' + str(num_trials))
plt.title('Probability of Starting Each Try = '
          + str(prob_of_success))

```

[Figure 17-32](#) Producing a geometric distribution



[Figure 17-33](#) A geometric distribution

The histogram implies that most of the time you'll get the car going within a few attempts. On the other hand, the long tail suggests

that on occasion you may run the risk of draining your battery before the car gets going.

### 17.4.6 Benford's Distribution

Benford's law defines a really strange distribution. Let  $S$  be a large set of decimal integers. How frequently would you expect each nonzero digit to appear as the first digit? Most of us would probably guess one ninth of the time. And when people are making up sets of numbers (e.g., faking experimental data or perpetrating financial fraud) this is typically true. It is not, however, typically true of many naturally occurring data sets. Instead, they follow a distribution predicted by Benford's law.

A set of decimal numbers is said to satisfy **Benford's law**<sup>121</sup> if the probability of the first digit being  $d$  is consistent with  $P(d) = \log_{10}(1 + 1/d)$ .

For example, this law predicts that the probability of the first digit being 1 is about 30%! Shockingly, many actual data sets seem to observe this law. It is possible to show that the Fibonacci sequence, for example, satisfies it perfectly. That's kind of plausible since the sequence is generated by a formula. It's less easy to understand why such diverse data sets as iPhone pass codes, the number of Twitter followers per user, the population of countries, or the distances of stars from the Earth closely approximate Benford's law.<sup>122</sup>

---

## 17.5 Hashing and Collisions

In Section 12.3 we pointed out that by using a larger hash table, we could reduce the incidence of collisions, and thus reduce the expected time to retrieve a value. We now have the intellectual tools we need to examine that tradeoff more precisely.

First, let's get a precise formulation of the problem.

- Assume:
  - The range of the hash function is 1 to  $n$
  - The number of insertions is  $k$

- The hash function produces a perfectly uniform distribution of the keys used in insertions, i.e., for all keys, key, and for all integers,  $i$ , in the range 1 to  $n$ , the probability that  
 $\text{hash}(\text{key}) = i$  is  $1/n$
- What is the probability that at least one collision occurs?

The question is exactly equivalent to asking, “given  $K$  randomly generated integers in the range 1 to  $n$ , what is the probability that at least two of them are equal?” If  $K \geq n$ , the probability is clearly 1. But what about when  $K < n$ ?

As is often the case, it is easiest to start by answering the inverse question, “given  $K$  randomly generated integers in the range 1 to  $n$ , what is the probability that none of them are equal?”

When we insert the first element, the probability of not having a collision is clearly 1. How about the second insertion? Since there are  $n-1$  hash results left that are not equal to the result of the first hash,  $n-1$  out of  $n$  choices will not yield a collision. So, the probability of not getting a collision on the second insertion is  $\frac{n-1}{n}$ , and the probability of not getting a collision on either of the first two insertions is  $1 * \frac{n-1}{n}$ . We can multiply these probabilities because for each insertion, the value produced by the hash function is independent of anything that has preceded it.

The probability of not having a collision after three insertions is  $1 * \frac{n-1}{n} * \frac{n-2}{n}$ . After  $K$  insertions it is  $1 * \frac{n-1}{n} * \frac{n-2}{n} * \dots * \frac{n-(K-1)}{n}$ .

To get the probability of having at least one collision, we subtract this value from 1, i.e., the probability is

$$1 - \left( \frac{n-1}{n} * \frac{n-2}{n} * \dots * \frac{n-(K-1)}{n} \right)$$

Given the size of the hash table and the number of expected insertions, we can use this formula to calculate the probability of at least one collision. If  $K$  were reasonably large, say 10,000, it would be a bit tedious to compute the probability with pencil and paper. That

leaves two choices, mathematics and programming. Mathematicians have used some fairly advanced techniques to find a way to approximate the value of this series. But unless  $k$  is very large, it is easier to run some code to compute the exact value of the series:

```
def collision_prob(n, k):
    prob = 1.0
    for i in range(1, k):
        prob = prob * ((n - i)/n)
    return 1 - prob
```

If we try `collision_prob(1000, 50)` we get a probability of about 0.71 of there being at least one collision. If we consider 200 insertions, the probability of a collision is nearly 1. Does that seem a bit high to you? Let's write a simulation, [Figure 17-34](#), to estimate the probability of at least one collision, and see if we get similar results.

```
def sim_insertions(num_indices, num_insertions):
    """Assumes num_indices and num_insertions are positive ints.
       Returns 1 if there is a collision; 0 otherwise"""
    choices = range(num_indices) #list of possible indices
    used = []
    for i in range(num_insertions):
        hash_val = random.choice(choices)
        if hash_val in used: #there is a collision
            return 1
        else:
            used.append(hash_val)
    return 0

def find_prob(num_indices, num_insertions, num_trials):
    collisions = 0
    for t in range(num_trials):
        collisions += sim_insertions(num_indices, num_insertions)
    return collisions/num_trials
```

[Figure 17-34](#) Simulating a hash table

If we run the code

```
print('Actual probability of a collision =',
      collision_prob(1000, 50))
print('Est. probability of a collision =', find_prob(1000,
```

```
    50, 10000))
print('Actual probability of a collision =',
      collision_prob(1000, 200))
print('Est. probability of a collision =', find_prob(1000,
200, 10000))
```

it prints

```
Actual probability of a collision = 0.7122686568799875
Est. probability of a collision = 0.7128
Actual probability of a collision = 0.9999999994781328
Est. probability of a collision = 1.0
```

The simulation results are comfortably similar to what we derived analytically.

Should the high probability of a collision make us think that hash tables have to be enormous to be useful? No. The probability of there being at least one collision tells us little about the expected lookup time. The expected time to look up a value depends upon the average length of the lists implementing the buckets holding the values that collided. Assuming a uniform distribution of hash values, this is simply the number of insertions divided by the number of buckets.

---

## 17.6 How Often Does the Better Team Win?

Almost every October, two teams from American Major League Baseball meet in something called the World Series. They play each other repeatedly until one of the teams has won four games, and that team is called (at least in the U.S.) the “world champion.”

Setting aside the question of whether there is reason to believe that one of the participants in the World Series is indeed the best team in the world, how likely is it that a contest that can be at most seven games long will determine which of the two participants is better?

Clearly, each year one team will emerge victorious. So the question is whether we should attribute that victory to skill or to luck.

[Figure 17-35](#) contains code that can provide us with some insight into that question. The function `sim_series` has one argument, `num_series`, a positive integer describing the number of seven-game

series to be simulated. It plots the probability of the better team winning the series against the probability of that team winning a single game. It varies the probability of the better team winning a single game from 0.5 to 1.0, and produces the plot in [Figure 17-36](#).

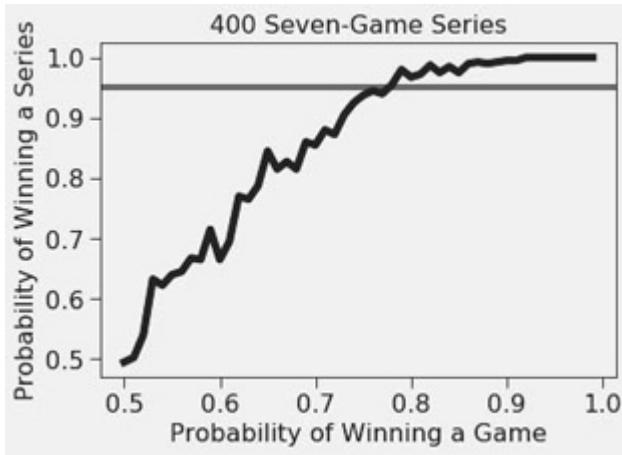
Notice that for the better team to win 95% of the time (0.95 on the y-axis), it needs to be so much better that it would win more than three out of every four games between the two teams. For comparison, in 2019, the two teams in the World Series had regular season winning percentages of 66% (Houston Astros) and 57.4% (Washington Nationals).[123](#)

```
def play_series(num_games, team_prob):
    numWon = 0
    for game in range(num_games):
        if random.random() <= team_prob:
            numWon += 1
    return (numWon > num_games//2)

def fraction_won(team_prob, num_series, series_len):
    won = 0
    for series in range(num_series):
        if play_series(series_len, team_prob):
            won += 1
    return won/float(num_series)

def sim_series(num_series):
    prob = 0.5
    fracsWon, probs = [], []
    while prob <= 1.0:
        fracsWon.append(fraction_won(prob, num_series, 7))
        probs.append(prob)
        prob += 0.01
    plt.axhline(0.95) #Draw line at 95%
    plt.plot(probs, fracsWon, 'k', linewidth = 5)
    plt.xlabel('Probability of Winning a Game')
    plt.ylabel('Probability of Winning a Series')
    plt.title(str(num_series) + ' Seven-Game Series')
```

[Figure 17-35](#) World Series simulation



[Figure 17-36](#) Probability of winning a 7-game series

---

## 17.7 Terms Introduced in Chapter

causal nondeterminism  
predictive nondeterminism  
deterministic program  
pseudorandom numbers  
independent event  
probability  
multiplicative law  
inferential statistics  
law of large numbers  
Bernoulli's theorem  
gambler's fallacy  
regression to the mean  
linear scaling  
variance  
standard deviation

coefficient of variation  
histogram  
frequency distribution  
probability distribution  
discrete random variable  
continuous random variable  
discrete probability distribution  
continuous probability distribution  
probability density function (PDF)  
normal (Gaussian) distribution  
bell curve  
area under curve  
empirical (68-95-99.7) rule  
confidence interval  
confidence level  
error bars  
continuous uniform distribution  
rectangular distribution  
discrete uniform distribution  
categorical (nominal) variable  
binomial distribution  
binomial coefficient  
Bernoulli trial  
multinomial distribution  
exponential distribution  
memoryless  
exponential decay

half-life

rate of decay

exponential growth

geometric distribution

Benford's Law

---

[108](#) Of course, this doesn't stop people from believing they are, and losing a lot of money based on that belief.

[109](#) A roll is fair if each of the six possible outcomes is equally likely. This is not always to be taken for granted. Excavations of Pompeii discovered "loaded" dice in which small lead weights had been inserted to bias the outcome of a roll. More recently, an online vendor's site said, "Are you unusually unlucky when it comes to rolling dice? Investing in a pair of dice that's more, uh, reliable might be just what you need."

[110](#) In point of fact, the values returned by `random.random` are not truly random. They are what mathematicians call **pseudorandom**. For almost all practical purposes, this distinction is not relevant and we shall ignore it.

[111](#) Though the law of large numbers had been discussed in the sixteenth century by Cardano, the first proof was published by Jacob Bernoulli in the early eighteenth century. It is unrelated to the theorem about fluid dynamics called Bernoulli's theorem, which was proved by Jacob's nephew Daniel.

[112](#) "On August 18, 1913, at the casino in Monte Carlo, black came up a record twenty-six times in succession [in roulette]. ... [There] was a near-panicky rush to bet on red, beginning about the time black had come up a phenomenal fifteen times. In application of the maturity [of the chances] doctrine, players doubled and tripled their stakes, this doctrine leading them to believe after black came up the twentieth time that there was not a chance in a million of another repeat. In the end the unusual run enriched the

Casino by some millions of francs.” Huff and Geis, *How to Take a Chance*, pp. 28-29.

113 The term “regression to the mean” was first used by Francis Galton in 1885 in a paper titled “Regression Toward Mediocrity in Hereditary Stature.” In that study he observed that children of unusually tall parents were likely to be shorter than their parents.

114 Random number generators in different versions of Python may not be identical. This means that even if you set the seed, you cannot assume that a program will behave the same way across versions of the language.

115 You'll probably never need to implement these yourself. Common libraries implement these and many other standard statistical functions. However, we present the code here on the off chance that some readers prefer looking at code to looking at equations.

116  $e$  is one of those magic irrational constants, like  $\pi$ , that show up all over the place in mathematics. The most common use is as the base of what are called “natural logarithms.” There are many equivalent ways of defining  $e$ , including as the value of  $(1 + \frac{1}{x})^x$  as  $x$  approaches infinity.

117 For polls, confidence intervals are not typically estimated by looking at the standard deviation of multiple polls. Instead, they use something called standard error; see Section 19.3.

118 The operative word here is “intended.” Political polls can go very wrong for reasons unrelated to the underlying statistical theory.

119 The parameter would have been called `lambda`, but as we saw in 4.4, `lambda` is a reserved word in Python.

120 The name “geometric distribution” arises from its similarity to a “geometric progression.” A geometric progression is any sequence of numbers in which each number other than the first is derived by multiplying the previous number by a constant

nonzero number. Euclid's *Elements* proves a number of interesting theorems about geometric progressions.

**121** The law is named after the physicist Frank Benford, who published a paper in 1938 showing that the law held on over 20,000 observations drawn from 20 domains. However, it was first postulated in 1881 by the astronomer Simon Newcomb.

**122** <http://testingbenfordslaw.com/>

**123** The Nationals won the series four games to three. Strangely, all seven games were won by the visiting team.

# 18

## MONTE CARLO SIMULATION

In Chapters 16 and 17, we looked at different ways of using randomness in computations. Many of the examples we presented fall into the class of computation known as **Monte Carlo simulation**. Monte Carlo simulation is a technique used to approximate the probability of an event by running the same simulation multiple times and averaging the results.

Stanislaw Ulam and Nicholas Metropolis coined the term Monte Carlo simulation in 1949 in homage to the games of chance played in the casino in the Principality of Monaco. Ulam, who is best known for designing the hydrogen bomb with Edward Teller, described the invention of the method as follows:

*The first thoughts and attempts I made to practice [the Monte Carlo Method] were suggested by a question which occurred to me in 1946 as I was convalescing from an illness and playing solitaires. The question was what are the chances that a Canfield solitaire laid out with 52 cards will come out successfully? After spending a lot of time trying to estimate them by pure combinatorial calculations, I wondered whether a more practical method than “abstract thinking” might not be to lay it out say one hundred times and simply observe and count the number of successful plays. This was already possible to envisage with the beginning of the new era of fast computers,<sup>124</sup> and I immediately thought of problems of neutron diffusion and other questions of mathematical physics, and more generally how to change processes described by certain differential equations into an equivalent form interpretable as a succession of random operations. Later ... [in 1946, I] described the idea to John von Neumann, and we began to plan actual calculations.<sup>125</sup>*

The technique was used during the Manhattan Project to predict what would happen during a nuclear fission reaction, but did not really take off until the 1950s, when computers became both more common and more powerful.

Ulam was not the first mathematician to think about using the tools of probability to understand a game of chance. The history of probability is intimately connected to the history of gambling. It is uncertainty that makes gambling possible. And the existence of gambling provoked the development of much of the mathematics needed to reason about uncertainty. Contributions to the foundations of probability theory by Cardano, Pascal, Fermat, Bernoulli, de Moivre, and Laplace were all motivated by a desire to better understand (and perhaps profit from) games of chance.

---

## 18.1 Pascal's Problem

Most of the early work on probability theory revolved around games using dice.<sup>126</sup> Reputedly, Pascal's interest in the field that came to be

known as probability theory began when a friend asked him whether it would be profitable to bet that within 24 rolls of a pair of dice he would roll a double 6. This was considered a hard problem in the mid-seventeenth century. Pascal and Fermat, two pretty smart guys, exchanged a number of letters about how to resolve the issue, but it now seems like an easy question to answer:

- On the first roll the probability of rolling a 6 on each die is  $1/6$ , so the probability of rolling a 6 with both dice is  $1/36$ .
- Therefore, the probability of not rolling a double 6 on the first roll is  $1 - 1/36 = 35/36$ .
- Therefore, the probability of not rolling 6 on both die 24 consecutive times is  $(35/36)^{24}$ , nearly 0.51, and therefore the probability of rolling a double 6 is  $1 - (35/36)^{24}$ , about 0.49. In the long run, it would not be profitable to bet on rolling a double 6 within 24 rolls.

Just to be safe, let's write a little program, [Figure 18-1](#), to simulate Pascal's friend's game and confirm that we get the same answer as Pascal. (All of the code in this chapter assumes that

```
import random
import numpy as np
```

occur at the start of the file in which the code occurs.) When run the first time, the call `check_pascal(1000000)` printed

```
Probability of winning = 0.490761
```

This is indeed quite close to  $1 - (35/36)^{24}$ ; typing `1 - (35.0/36.0) ** 24` into the Python shell produces 0.49140387613090342.

```

def roll_die():
    return random.choice([1,2,3,4,5,6])

def check_pascal(num_trials):
    """Assumes num_trials is an int > 0
       Prints an estimate of the probability of winning"""
    num_wins = 0
    for i in range(num_trials):
        for j in range(24):
            d1 = roll_die()
            d2 = roll_die()
            if d1 == 6 and d2 == 6:
                num_wins += 1
                break
    print('Probability of winning =', num_wins/num_trials)

```

[Figure 18-1](#) Checking Pascal's analysis

---

## 18.2 Pass or Don't Pass?

Not all questions about games of chance are so easily answered. In the game craps, the shooter (the person who rolls the dice) chooses between making a “pass line” or a “don’t pass line” bet.

- **Pass Line:** Shooter wins if the first roll is a “natural” (7 or 11) and loses if it is “craps” (2, 3, or 12). If some other number is rolled, that number becomes the “point,” and the shooter keeps rolling. If the shooter rolls the point before rolling a 7, the shooter wins. Otherwise the shooter loses.
- **Don't Pass Line:** Shooter loses if the first roll is 7 or 11, wins if it is 2 or 3, and ties (a “push” in gambling jargon) if it is 12. If some other number is rolled, that number becomes the point, and the shooter keeps rolling. If the shooter rolls a 7 before rolling the point, the shooter wins. Otherwise the shooter loses.

Is one of these a better bet than the other? Is either a good bet? It is possible to analytically derive the answer to these questions, but it seems easier (at least to us) to write a program that simulates a craps game and see what happens. [Figure 18-2](#) contains the heart of such a simulation.

```

class Craps_game(object):
    def __init__(self):
        self.pass_wins, self.pass_losses = 0, 0
        self.dp_wins, self.dp_losses, self.dp_pushes = 0, 0, 0

    def play_hand(self):
        throw = roll_die() + roll_die()
        if throw == 7 or throw == 11:
            self.pass_wins += 1
            self.dp_losses += 1
        elif throw == 2 or throw == 3 or throw == 12:
            self.pass_losses += 1
            if throw == 12:
                self.dp_pushes += 1
            else:
                self.dp_wins += 1
        else:
            point = throw
            while True:
                throw = roll_die() + roll_die()
                if throw == point:
                    self.pass_wins += 1
                    self.dp_losses += 1
                    break
                elif throw == 7:
                    self.pass_losses += 1
                    self.dp_wins += 1
                    break

    def pass_results(self):
        return (self.pass_wins, self.pass_losses)

    def dp_results(self):
        return (self.dp_wins, self.dp_losses, self.dp_pushes)

```

[Figure 18-2](#) Craps\_game class

The values of the instance variables of an instance of class `Craps_game` record the performance of the pass and don't pass lines since the start of the game. The observer methods `pass_results` and `dp_results` return these values. The method `play_hand` simulates one hand of a game. A “hand” starts when the shooter is “coming out,” the term used in craps for a roll before a point is established. A hand ends when the shooter has won or lost his or her initial bet. The bulk of the code in `play_hand` is merely an algorithmic description of the

rules stated above. Notice that there is a loop in the `else` clause corresponding to what happens after a point is established. It is exited using a `break` statement when either a seven or the point is rolled.

[Figure 18-3](#) contains a function that uses class `Craps_game` to simulate a series of craps games.

```
def craps_sim(hands_per_game, num_games):
    """Assumes hands_per_game and num_games are ints > 0
       Play num_games games of hands_per_game hands; print results"""
    games = []

    #Play num_games games
    for t in range(num_games):
        c = Craps_game()
        for i in range(hands_per_game):
            c.play_hand()
        games.append(c)

    #Produce statistics for each game
    p_ROI_per_game, dp_ROI_per_game = [], []
    for g in games:
        wins, losses = g.pass_results()
        p_ROI_per_game.append((wins - losses)/float(hands_per_game))
        wins, losses, pushes = g.dp_results()
        dp_ROI_per_game.append((wins - losses)/float(hands_per_game))

    #Produce and print summary statistics
    mean_ROI = str(round((100*sum(p_ROI_per_game)/num_games), 4)) + '%'
    sigma = str(round(100*np.std(p_ROI_per_game), 4)) + '%'
    print('Pass:', 'Mean ROI =', mean_ROI, 'Std. Dev. =', sigma)
    mean_ROI = str(round((100*sum(dp_ROI_per_game)/num_games), 4)) + '%'
    sigma = str(round(100*np.std(dp_ROI_per_game), 4)) + '%'
    print('Don\'t pass:', 'Mean ROI =', mean_ROI, 'Std Dev =', sigma)
```

[Figure 18-3](#) Simulating a craps game

The structure of `craps_sim` is typical of many simulation programs:

1. It runs multiple games (think of each game as analogous to a trial in our earlier simulations) and accumulates the results. Each game includes multiple hands, so there is a nested loop.

2. It then produces and stores statistics for each game.
3. Finally, it produces and outputs summary statistics. In this case, it prints the expected return on investment (ROI) for each kind of betting line and the standard deviation of that ROI.

**Return on investment** is defined by the equation[127](#)

$$ROI = \frac{gain\ from\ investment - cost\ of\ investment}{cost\ of\ investment}$$

Since the pass and don't pass lines pay even money (if you bet \$1 and win, you gain \$1), the ROI is

$$ROI = \frac{number\ of\ wins - number\ of\ losses}{number\ of\ bets}$$

For example, if you made 100 pass line bets and won half, your ROI would be

$$\frac{50 - 50}{100} = 0$$

If you bet the don't pass line 100 times and had 25 wins and 5 pushes, the ROI would be

$$\frac{25 - 70}{100} = \frac{-45}{100} = -4.5$$

Let's run our craps game simulation and see what happens when we try `craps_sim(20, 10)`:[128](#)

```
Pass: Mean ROI = -7.0% Std. Dev. = 23.6854%
Don't pass: Mean ROI = 4.0% Std Dev = 23.5372%
```

It looks as if it would be a good idea to avoid the pass line—where the expected return on investment is a 7% loss. But the don't pass line looks like a pretty good bet. Or does it?

Looking at the standard deviations, it seems that perhaps the don't pass line is not such a safe bet after all. Recall that under the assumption that the distribution is normal, the 95% confidence interval is encompassed by 1.96 standard deviations on either side of the mean. For the don't pass line, the 95% confidence interval is [4.0 - 1.96\*23.5372, 4.0+1.96\*23.5372]—roughly [-43%, +51%]. That certainly doesn't suggest that betting the don't pass line is a sure thing.

Time to put the law of large numbers to work;  
`craps_sim(1000000, 10)` prints

```
Pass: Mean ROI = -1.4204% Std. Dev. = 0.0614%
Don't pass: Mean ROI = -1.3571% Std Dev = 0.0593%
```

We can now be pretty safe in assuming that neither of these is a good bet.<sup>129</sup> It looks as if the don't pass line might be slightly less bad, but we probably shouldn't count on that. If the 95% confidence intervals for the pass and don't pass lines did not overlap, it would be safe to assume that the difference in the two means was statistically significant.<sup>130</sup> However, they do overlap, so no conclusion can be safely drawn.

Suppose that instead of increasing the number of hands per game, we increased the number of games, e.g., by making the call `craps_sim(20, 1000000)`:

```
Pass: Mean ROI = -1.4133% Std. Dev. = 22.3571%
Don't pass: Mean ROI = -1.3649% Std Dev = 22.0446%
```

The standard deviations are high—indicating that the outcome of a single game of 20 hands is highly uncertain.

One of the nice things about simulations is that they make it easy to perform “what if” experiments. For example, what if a player could sneak in a pair of cheater's dice that favored 5 over 2 (5 and 2 are on

the opposite sides of a die)? To test this out, all we have to do is replace the implementation of `roll_die` by something like

```
def roll_die():
    return random.choice([1,1,2,3,3,4,4,5,5,5,6,6])
```

This relatively small change in the die makes a dramatic difference in the odds. Running `craps_sim(1000000, 10)` yields

```
Pass: Mean ROI = 6.6867% Std. Dev. = 0.0993%
Don't pass: Mean ROI = -9.469% Std Dev = 0.1024%
```

No wonder casinos go to a lot of trouble to make sure that players don't introduce their own dice into the game!

**Finger exercise:** A “big 6” bet pays even money if a 6 is rolled before a 7. Assuming 30 \$5 bets per hour, write a Monte Carlo simulation that estimates the cost per hour and the standard deviation of that cost of playing “big 6” bets.

---

### 18.3 Using Table Lookup to Improve Performance

You might not want to try running `craps_sim(100000000, 10)` at home. It takes a long time to complete on most computers. That raises the question of whether there is a simple way to speed up the simulation.

The complexity of the implementation of the function `craps_sim` is roughly  $\theta(\text{play\_hand}) * \text{hands\_per\_game} * \text{num\_games}$ . The running time of `play_hand` depends upon the number of times the loop in it is executed. In principle, the loop could be executed an unbounded number of times since there is no bound on how long it could take to roll either a 7 or the point. In practice, of course, we have every reason to believe it will always terminate.

Notice, however, that the result of a call to `play_hand` does not depend on how many times the loop is executed, but only on which exit condition is reached. For each possible point, we can easily calculate the probability of rolling that point before rolling a 7. For example, using a pair of dice we can roll a 4 in three different ways:  $\langle 1, 3 \rangle$ ,  $\langle 3, 1 \rangle$ , and  $\langle 2, 2 \rangle$ . We can roll a 7 in six different ways:

$\langle 1, 6 \rangle$ ,  $\langle 6, 1 \rangle$ ,  $\langle 2, 5 \rangle$ ,  $\langle 5, 2 \rangle$ ,  $\langle 3, 4 \rangle$ , and  $\langle 4, 3 \rangle$ . Therefore, exiting the loop by rolling a 7 is twice as likely as exiting the loop by rolling a 4.

[Figure 18-4](#) contains an implementation of `play_hand` that exploits this thinking. We first compute the probability of making the point before rolling a 7 for each possible value of the point and store those values in a dictionary. Suppose, for example, that the point is 8. The shooter continues to roll until he either rolls the point or rolls craps. There are five ways of rolling an 8 ( $\langle 6, 2 \rangle$ ,  $\langle 2, 6 \rangle$ ,  $\langle 5, 3 \rangle$ ,  $\langle 3, 5 \rangle$ , and  $\langle 4, 4 \rangle$ ) and six ways of rolling a 7. So, the value for the dictionary key 8 is the value of the expression  $5/11$ . Having this table allows us to replace the inner loop, which contained an unbounded number of rolls, with a test against one call to `random.random`. The asymptotic complexity of this version of `play_hand` is  $O(1)$ .

The idea of replacing computation by **table lookup** has broad applicability and is frequently used when speed is an issue. Table lookup is an example of the general idea of **trading time for space**. As we saw in Chapter 15, it is the key idea behind dynamic programming. We saw another example of this technique in our analysis of hashing: the larger the table, the fewer the collisions, and the faster the average lookup. In this case, the table is small, so the space cost is negligible.

```

def play_hand(self):
    #An alternative, faster, implementation of play_hand
    points_dict = {4:1/3, 5:2/5, 6:5/11, 8:5/11, 9:2/5, 10:1/3}
    throw = roll_die() + roll_die()
    if throw == 7 or throw == 11:
        self.pass_wins += 1
        self.dp_losses += 1
    elif throw == 2 or throw == 3 or throw == 12:
        self.pass_losses += 1
        if throw == 12:
            self.dp_pushes += 1
        else:
            self.dp_wins += 1
    else:
        if random.random() <= points_dict[throw]: # point before 7
            self.pass_wins += 1
            self.dp_losses += 1
        else:                                     # 7 before point
            self.pass_losses += 1
            self.dp_wins += 1

```

[Figure 18-4](#) Using table lookup to improve performance

---

## 18.4 Finding $\pi$

It is easy to see how Monte Carlo simulation is useful for tackling problems in which nondeterminism plays a role. Interestingly, however, Monte Carlo simulation (and randomized algorithms in general) can be used to solve problems that are not inherently stochastic, i.e., for which there is no uncertainty about outcomes.

Consider  $\pi$ . For thousands of years, people have known that there is a constant (called  $\pi$  since the eighteenth century) such that the circumference of a circle is equal to  $\pi * \text{diameter}$  and the area of the circle equal to  $\pi * \text{radius}^2$ . What they did not know was the value of this constant.

One of the earliest estimates,  $4 * (8/9)^2 = 3.16$ , can be found in Egyptian *Rhind Papyrus*, circa 1650 BC. More than a thousand years later, the *Old Testament* (1 Kings 7.23) implied a different value for  $\pi$  when giving the specifications of one of King Solomon's construction projects,

*And he made a molten sea, ten cubits from the one brim to the other: it was round all about, and his height was five cubits: and a line of 30 cubits did compass it round about.*

Solving for  $\pi$ ,  $10\pi = 30$ , so  $\pi = 3$ . Perhaps the *Bible* is simply wrong, or perhaps the molten sea wasn't perfectly circular, or perhaps the circumference was measured from the outside of the wall and the diameter from the inside, or perhaps it's just poetic license. We leave it to the reader to decide.

Archimedes of Syracuse (287-212 BCE) derived upper and lower bounds on the value of  $\pi$  by using a high-degree polygon to approximate a circular shape. Using a polygon with 96 sides, he concluded that  $223/71 < \pi < 22/7$ . Giving upper and lower bounds was a rather sophisticated approach for the time. If we take his best estimate as the average of his two bounds, we obtain 3.1418, an error of about 0.0002. Not bad! But about 700 years later, the Chinese mathematician Zu Chongzhi used a polygon with 24,576 sides to conclude that  $3.1415962 < \pi < 3.1415927$ . About 800 years after that, the Dutch cartographer Adriaan Anthonisz (1527-1607) estimated it as  $355/113$ , roughly 3.1415929203539825. That estimate is good enough for most practical purposes, but it didn't keep mathematicians from working on the problem.

Long before computers were invented, the French mathematicians Buffon (1707-1788) and Laplace (1749-1827) proposed using a stochastic simulation to estimate the value of  $\pi$ .<sup>131</sup> Think about inscribing a circle in a square with sides of length 2, so that the radius,  $r$ , of the circle is of length 1.

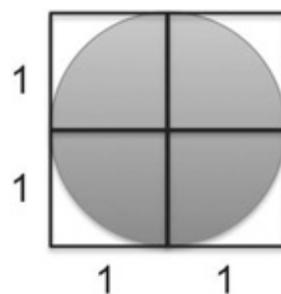


Figure 18-5 Unit circle inscribed in a square

By the definition of  $\pi$ ,  $\text{area} = \pi r^2$ . Since  $r$  is 1,  $\pi = \text{area}$ . But what's the area of the circle? Buffon suggested that he could estimate the area of a circle by dropping a large number of needles (which he argued would follow a random path as they fell) in the vicinity of the square. The ratio of the number of needles with tips lying within the square to the number of needles with tips lying within the circle could then be used to estimate the area of the circle.

If the locations of the needles are truly random, we know that

$$\frac{\text{needles in circle}}{\text{needles in square}} = \frac{\text{area of circle}}{\text{area of square}}$$

and solving for the area of the circle,

$$\text{area of circle} = \frac{\text{area of square} * \text{needles in circle}}{\text{needles in square}}$$

Recall that the area of a 2 by 2 square is 4, so,

$$\text{area of circle} = \frac{4 * \text{needles in circle}}{\text{needles in square}}$$

In general, to estimate the area of some region  $R$ :

1. Pick an enclosing region,  $E$ , such that the area of  $E$  is easy to calculate and  $R$  lies completely within  $E$ .
2. Pick a set of random points that lie within  $E$ .
3. Let  $F$  be the fraction of the points that fall within  $R$ .
4. Multiply the area of  $E$  by  $F$ .

If you try Buffon's experiment, you'll soon realize that the places where the needles land are not truly random. Moreover, even if you

could drop them randomly, it would take a very large number of needles to get an approximation of  $\pi$  as good as even the *Bible's*. Fortunately, computers can randomly drop simulated needles at a ferocious rate.<sup>132</sup>

[Figure 18-6](#) contains a program that estimates  $\pi$  using the Buffon-Laplace method. For simplicity, it considers only those needles that fall in the upper-right quadrant of the square.

```
def throw_needles(num_needles):
    in_circle = 0
    for Needles in range(1, num_needles + 1):
        x = random.random()
        y = random.random()
        if (x*x + y*y)**0.5 <= 1:
            in_circle += 1
    #Counting needles in one quadrant only, so multiply by 4
    return 4*(in_circle/num_needles)

def get_est(num_needles, num_trials):
    estimates = []
    for t in range(num_trials):
        pi_guess = throw_needles(num_needles)
        estimates.append(pi_guess)
    std_dev = np.std(estimates)
    cur_est = sum(estimates)/len(estimates)
    print('Est. =', str(round(cur_est, 5)) + ',',
          'Std. dev. =', str(round(std_dev, 5)) + ',',
          'Needles =', num_needles)
    return (cur_est, std_dev)

def est_pi(precision, num_trials):
    num_needles = 1000
    std_dev = precision
    while std_dev > precision/1.96:
        cur_est, std_dev = get_est(num_needles, num_trials)
        num_needles *= 2
    return cur_est
```

[Figure 18-6](#) Estimating  $\pi$

The function `throw_needles` simulates dropping a needle by first using `random.random` to get a pair of positive Cartesian coordinates (`x` and `y` values) representing the position of the needle with respect to

the center of the square. It then uses the Pythagorean theorem to compute the hypotenuse of the right triangle with base  $x$  and height  $y$ . This is the distance of the tip of the needle from the origin (the center of the square). Since the radius of the circle is  $1$ , we know that the needle lies within the circle if and only if the distance from the origin is no greater than  $1$ . We use this fact to count the number of needles in the circle.

The function `get_est` uses `throw_needles` to find an estimate of  $\pi$  by first dropping `num_needles` needles, and then averaging the result over `num_trials` trials. It then returns the mean and standard deviation of the trials.

The function `est_pi` calls `get_est` with an ever-growing number of needles until the standard deviation returned by `get_est` is no larger than `precision/1.96`. Under the assumption that the errors are normally distributed, this implies that 95% of the values lie within `precision` of the mean.

When we ran `est_pi(0.01, 100)`, it printed

```
Est. = 3.14844, Std. dev. = 0.04789, Needles = 1000
Est. = 3.13918, Std. dev. = 0.0355, Needles = 2000
Est. = 3.14108, Std. dev. = 0.02713, Needles = 4000
Est. = 3.14143, Std. dev. = 0.0168, Needles = 8000
Est. = 3.14135, Std. dev. = 0.0137, Needles = 16000
Est. = 3.14131, Std. dev. = 0.00848, Needles = 32000
Est. = 3.14117, Std. dev. = 0.00703, Needles = 64000
Est. = 3.14159, Std. dev. = 0.00403, Needles = 128000
```

As we would expect, the standard deviations decreased monotonically as we increased the number of samples. In the beginning the estimates of the value of  $\pi$  also improved steadily. Some were above the true value and some below, but each increase in `num_needles` led to an improved estimate. With 1000 samples per trial, the simulation's estimate was already better than those of the *Bible* and the *Rhind Papyrus*.

Curiously, the estimate got worse when the number of needles increased from 8,000 to 16,000, since 3.14135 is farther from the true value of  $\pi$  than is 3.14143. However, if we look at the ranges defined by one standard deviation around each of the means, both ranges contain the true value of  $\pi$ , and the range associated with the larger sample size is smaller. Even though the estimate generated with

$16,000$  samples happens to be farther from the actual value of  $\pi$ , we should have more confidence in its accuracy. This is an extremely important notion. It is not sufficient to produce a good answer. We have to have a valid reason to be confident that it is in fact a good answer. And when we drop a large enough number of needles, the small standard deviation gives us reason to be confident that we have a correct answer. Right?

Not exactly. Having a small standard deviation is a necessary condition for having confidence in the validity of the result. It is not a sufficient condition. The notion of a statistically valid conclusion should never be confused with the notion of a correct conclusion.

Each statistical analysis starts with a set of assumptions. The key assumption here is that our simulation is an accurate model of reality. Recall that the design of our Buffon-Laplace simulation started with a little algebra demonstrating how we could use the ratio of two areas to find the value of  $\pi$ . We then translated this idea into code that depended upon a little geometry and on the randomness of `random.random`.

Let's see what happens if we get any of this wrong. Suppose, for example, we replace the `4` in the last line of the function `throw_needles` by a `2`, and again run `est_pi(0.01, 100)`. This time it prints

```
Est. = 1.57422, Std. dev. = 0.02394, Needles = 1000
Est. = 1.56959, Std. dev. = 0.01775, Needles = 2000
Est. = 1.57054, Std. dev. = 0.01356, Needles = 4000
Est. = 1.57072, Std. dev. = 0.0084, Needles = 8000
Est. = 1.57068, Std. dev. = 0.00685, Needles = 16000
Est. = 1.57066, Std. dev. = 0.00424, Needles = 32000
```

The standard deviation for a mere  $32,000$  needles suggests that we should have a fair amount of confidence in the estimate. But what does that really mean? It means that we can be reasonably confident that if we were to draw more samples from the same distribution, we would get a similar value. It says nothing about whether this value is close to the actual value of  $\pi$ . If you are going to remember only one thing about statistics, remember this: a statistically valid conclusion should not be confused with a correct conclusion!

Before believing the results of a simulation, we need to have confidence both that our conceptual model is correct and that we

have correctly implemented that model. Whenever possible, you should attempt to validate results against reality. In this case, you could use some other means to compute an approximation to the area of a circle (e.g., physical measurement) and check that the computed value of  $\pi$  is at least in the right neighborhood.

---

## 18.5 Some Closing Remarks about Simulation Models

For most of the history of science, theorists used mathematical techniques to construct purely analytical models that could be used to predict the behavior of a system from a set of parameters and initial conditions. This led to the development of important mathematical tools ranging from calculus to probability theory. These tools helped scientists develop a reasonably accurate understanding of the macroscopic physical world.

As the twentieth century progressed, the limitations of this approach became increasingly clear. Reasons for this include:

- An increased interest in the social sciences, e.g., economics, led to a desire to construct good models of systems that were not mathematically tractable.
- As the systems to be modeled grew increasingly complex, it seemed easier to successively refine a series of simulation models than to construct accurate analytic models.
- It is often easier to extract useful intermediate results from a simulation than from an analytical model, e.g., to play “what if” games.
- The availability of computers made it feasible to run large-scale simulations. Until the advent of the modern computer in the middle of the twentieth century the utility of simulation was limited by the time required to perform calculations by hand.

Simulation models are **descriptive**, not **prescriptive**. They tell how a system works under given conditions; not how to arrange the conditions to make the system work best. A simulation does not optimize, it merely describes. That is not to say that simulation cannot be used as part of an optimization process. For example,

simulation is often used as part of a search process in finding an optimal set of parameter settings.

Simulation models can be classified along three dimensions:

- Deterministic versus stochastic
- Static versus dynamic
- Discrete versus continuous

The behavior of a **deterministic simulation** is completely defined by the model. Rerunning a simulation will not change the outcome. Deterministic simulations are typically used when the system being modeled is itself deterministic but is too complex to analyze analytically, e.g., the performance of a processor chip. **Stochastic simulations** incorporate randomness in the model. Multiple runs of the same model may generate different values. This random element forces us to generate many outcomes to see the range of possibilities. The question of whether to generate 10 or 1000 or 100,000 outcomes is a statistical question, as discussed earlier.

In a **static model**, time plays no essential role. The needle-dropping simulation used to estimate  $\pi$  in this chapter is an example of a static simulation. In a **dynamic model**, time, or some analog, plays an essential role. In the series of random walks simulated in Chapter 16, the number of steps taken was used as a surrogate for time.

In a **discrete model**, the values of pertinent variables are enumerable, e.g., they are integers. In a **continuous model**, the values of pertinent variables range over non-enumerable sets, e.g., the real numbers. Imagine analyzing the flow of traffic along a highway. We might choose to model each individual car, in which case we have a discrete model. Alternatively, we might choose to treat traffic as a flow, where changes in the flow can be described by differential equations. This leads to a continuous model. In this example, the discrete model more closely resembles the physical situation (nobody drives half a car, though some cars are half the size of others), but is more computationally complex than a continuous one. In practice, models often have both discrete and continuous components. For example, we might choose to model the flow of blood through the human body using a discrete model for blood (i.e.,

modeling individual corpuscles) and a continuous model for blood pressure.

---

## 18.6 Terms Introduced in Chapter

Monte Carlo simulation  
return on investment (ROI)  
table lookup  
time/space tradeoff  
descriptive model  
prescriptive model  
deterministic simulation  
stochastic simulation  
static model  
dynamic model  
discrete model  
continuous model

---

[124](#) Ulam was probably referring to the ENIAC, which performed about  $10^3$  additions a second (and weighed 25 tons). Today's computers perform about  $10^9$  additions a second.

[125](#) Eckhardt, Roger (1987). “Stan Ulam, John von Neumann, and the Monte Carlo method,” *Los Alamos Science*, Special Issue (15), 131-137.

[126](#) Archeological excavations suggest that dice are the human race's oldest gambling implement. The oldest known “modern” six-sided die dates to about 600 BCE, but Egyptian tombs dating to about 2000 BCE contain artifacts resembling dice. Typically,

these early dice were made from animal bones; in gambling circles people still use the phrase “rolling the bones.”

**127** More precisely, this equation defines what is often called “simple ROI.” It does not account for the possibility that there might be a gap in time between when the investment is made and when the gain attributable to that investment occurs. This gap should be accounted for when the time between making an investment and seeing the financial return is large (e.g., investing in a college education). This is probably not an issue at the craps table.

**128** Since these programs incorporate randomness, you should not expect to get identical results if you run the code yourself. More important, do not place any bets until you have read the entire section!

**129** In fact, the means of the estimated ROIs are close to the actual ROIs. Grinding through the probabilities yields an ROI of  $-1.414\%$  for the pass line and  $-1.364\%$  for the don't pass line.

**130** We discuss statistical significance in more detail in Chapter 21.

**131** Buffon proposed the idea first, but there was an error in his formulation that was later corrected by Laplace.

**132** For a more dramatic demonstration of the Buffon-Laplace method take a look at <https://www.youtube.com/watch?v=oYM6MIjZ8IY>.

# 19

## SAMPLING AND CONFIDENCE

Recall that inferential statistics involves making inferences about a **population** of examples by analyzing a randomly chosen subset of that population. This subset is called a **sample**.

Sampling is important because it is often not possible to observe the entire population of interest. A physician cannot count the number of a species of bacterium in a patient's blood stream, but it is possible to measure the population in a small sample of the patient's blood, and from that to infer characteristics of the total population. If you wanted to know the average weight of eighteen-year-old Americans, you could try and round them all up, put them on a very large scale, and then divide by the number of people. Alternatively, you could round up 50 randomly chose eighteen-year-olds, compute their mean weight, and assume that their mean weight was a reasonable estimate of the mean weight of the entire population of eighteen-year-olds.

The correspondence between the sample and the population of interest is of overriding importance. If the sample is not representative of the population, no amount of fancy mathematics will lead to valid inferences. A sample of 50 women or 50 Asian-Americans or 50 football players cannot be used to make valid inferences about the average weight of the population of all eighteen-year-olds in America.

In this chapter, we focus on **probability sampling**. With probability sampling, each member of the population of interest has some nonzero probability of being included in the sample. In a **simple random sample**, each member of the population has an equal chance of being chosen for the sample. In **stratified sampling**, the population is first partitioned into subgroups, and then the sample is built by randomly sampling from each subgroup.

Stratified sampling can be used to increase the probability that a sample is representative of the population as a whole. For example, ensuring that the fraction of men and women in a sample matches the fraction of men and women in the population increases the probability that the mean weight of the sample, the **sample mean**, will be a good estimate of the mean weight of the whole population, the **population mean**.

The code in the chapter assumes the following import statements

```
import random
import numpy as np
import matplotlib.pyplot as plt
import scipy
```

---

## 19.1 Sampling the Boston Marathon

Each year since 1897, athletes (mostly runners, but since 1975 there has been a wheelchair division) have gathered in Massachusetts to participate in the Boston Marathon.<sup>133</sup> In recent years, around 20,000 hardy souls per year have successfully taken on the 42.195 km (26 mile, 385 yard) course.

A file containing data from the 2012 race is available on the website associated with this book. The file `bm_results2012.csv` is in a comma-separated format, and contains the name, gender,<sup>134</sup> age, division, country, and time for each participant. [Figure 19-1](#) contains the first few lines of the contents of the file.

Name,Gender,Age,Div,Ctry,Time
Gebremariam Gebregziabher,M,27,14,ETH,142.93
Matebo Levy,M,22,2,KEN,133.10
Cherop Sharon,F,28,1,KEN,151.83
Chebet Wilson,M,26,5,KEN,134.93
Dado Firehiwot,F,28,4,ETH,154.93
Korir Laban,M,26,6,KEN,135.48
Jeptoo Rita,F,31,6,KEN,155.88
Korir Wesley,M,29,1,KEN,132.67

[Figure 19-1](#) The first few lines in `bm_results2012.csv`

Since complete data about the results of each race is easily available, there is no pragmatic need to use sampling to derive statistics about a race. However, it is pedagogically useful to compare statistical estimates derived from samples to the actual value being estimated.

The code in [Figure 19-2](#) produces the plot shown in [Figure 19-3](#). The function `get_BM_data` reads data from a file containing information about each of the competitors in the race. It returns the data in a dictionary with six elements. Each key describes the type of data (e.g., `'name'` or `'gender'`) contained in the elements of a list associated with that key. For example, `data['time']` is a list of floats containing the finishing time of each competitor, `data['name'][i]` is the name of the  $i^{\text{th}}$  competitor, and `data['time'][i]` is the finishing time of the  $i^{\text{th}}$  competitor. The function `make_hist` produces a visual representation of the finishing times. (In Chapter 23, we will look at a Python module, Pandas, that could be used to simplify a lot of the code in this chapter, including `get_BM_data` and `make_hist`.)

```

def get_BM_data(filename):
    """Read the contents of the given file. Assumes the file is
    in a comma-separated format, with 6 elements in each entry:
    0. Name (string), 1. Gender (string), 2. Age (int)
    3. Division (int), 4. Country (string), 5. Overall time (float)
    Returns: dict containing a list for each of the 6 variables."""

    data = {}
    with open(filename, 'r') as f:
        f.readline() #discard first line
        line = f.readline()
        for k in ('name', 'gender', 'age', 'division',
                  'country', 'time'):
            data[k] = []
        while line != '':
            split = line.split(',')
            data['name'].append(split[0])
            data['gender'].append(split[1])
            data['age'].append(int(split[2]))
            data['division'].append(int(split[3]))
            data['country'].append(split[4])
            data['time'].append(float(split[5][:-1])) #remove \n
            line = f.readline()
    return data

def make_hist(data, bins, title, xlabel, ylabel):
    plt.hist(data, bins)
    plt.title(title)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    mean = sum(data)/len(data)
    std = np.std(data)
    plt.annotate('Mean = ' + str(round(mean, 2)) +
                 '\nSD = ' + str(round(std, 2)), fontsize = 14,
                 xy = (0.65, 0.75), xycoords = 'axes fraction')

```

[Figure 19-2](#) Read data and produce plot of Boston Marathon

## The code

```

times = get_BM_data('bm_results2012.csv')['time']
make_hist(times, 20, '2012 Boston Marathon',
          'Minutes to Complete Race', 'Number of Runners')

```

produces the plot in [Figure 19-3](#).



[Figure 19-3](#) Boston Marathon finishing times

The distribution of finishing times resembles a normal distribution but is clearly not normal because of the fat tail on the right.

Now, let's pretend that we don't have access to the data about all competitors, and instead want to estimate some statistics about the finishing times of the entire field by sampling a small number of randomly chosen competitors.

The code in [Figure 19-4](#) creates a simple random sample of the elements of `times`, and then uses that sample to estimate the mean and standard deviation of `times`. The function `sample_times` uses `random.sample(times, num_examples)` to extract the sample. The invocation of `random.sample` returns a list of size `num_examples` of randomly chosen distinct elements from the list `times`. After extracting the sample, `sample_times` produces a histogram showing the distribution of values in the sample.

```

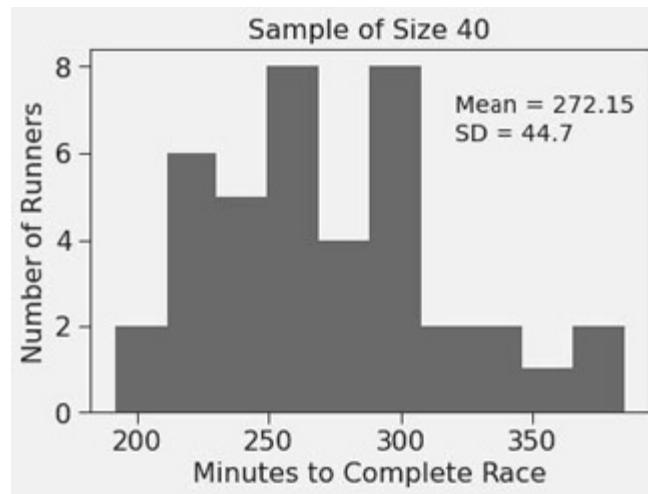
def sample_times(times, num_examples):
    """Assumes times is a list of floats representing finishing
       times of all runners. num_examples is an int
       Generates a random sample of size num_examples, and produces
       a histogram showing the distribution along with its mean and
       standard deviation"""
    sample = random.sample(times, num_examples)
    make_hist(sample, 10, 'Sample of Size ' + str(num_examples),
              'Minutes to Complete Race', 'Number of Runners')

sample_size = 40
sample_times(times, sample_size)

```

[Figure 19-4](#) Sampling finishing times

As [Figure 19-5](#) shows, the distribution of the sample is much farther from normal than the distribution from which it was drawn. This is not surprising, given the small sample size. What's more surprising is that despite the small sample size (40 out of about 21,000) the estimated mean differs from the population mean by around 3%. Did we get lucky, or is there reason to expect that the estimate of the mean will be pretty good? To put it another way, can we express in a quantitative way how much confidence we should have in our estimate?



[Figure 19-5](#) Analyzing a small sample

As we discussed in Chapters 17 and 18, it is often useful to provide a confidence interval and confidence level to indicate the reliability of the estimate. Given a single sample (of any size) drawn from a larger population, the best estimate of the mean of the population is the mean of the sample. Estimating the width of the confidence interval required to achieve a desired confidence level is trickier. It depends, in part, upon the size of the sample.

It's easy to understand why the size of the sample is important. The law of large numbers tells us that as the sample size grows, the distribution of the values of the sample is more likely to resemble the distribution of the population from which the sample is drawn. Consequently, as the sample size grows, the sample mean and the sample standard deviation are likely to be closer to the population mean and population standard deviation.

So, bigger is better, but how big is big enough? That depends upon the variance of the population. The higher the variance, the more samples are needed. Consider two normal distributions, one with a mean of 0 and standard deviation of 1, and the other with a mean of 0 and a standard deviation of 100. If we were to select one randomly chosen element from one of these distributions and use it to estimate the mean of the distribution, the probability of that estimate being within any desired accuracy,  $\epsilon$ , of the true mean ( $\mu$ ), would be equal to the area under the probability density function between  $-\epsilon$  and  $\epsilon$  (see Section 17.4.1). The code in [Figure 19-6](#) computes and prints these probabilities for  $\epsilon = 3$  minutes.

```
def gaussian(x, mu, sigma):
    factor1 = (1/(sigma*((2*np.pi)**0.5)))
    factor2 = np.e**-((x-mu)**2)/(2*sigma**2)
    return factor1*factor2

area = round(scipy.integrate.quad(gaussian, -3, 3, (0, 1))[0], 4)
print('Probability of being within 3',
      'of true mean of tight dist. =', area)
area = round(scipy.integrate.quad(gaussian, -3, 3, (0, 100))[0], 4)
print('Probability of being within 3',
      'of true mean of wide dist. =', area)
```

[Figure 19-6](#) Effect of variance on estimate of mean

When the code in [Figure 19-6](#) is run, it prints

```
Probability of being within 3 of true mean of tight dist. =
0.9973
Probability of being within 3 of true mean of wide dist. =
0.0239
```

The code in [Figure 19-7](#) plots the mean of each of 1000 samples of size 40 from two normal distributions. Again, each distribution has a mean of 0, but one has a standard deviation of 1 and the other a standard deviation of 100.

```
def test_samples(num_trials, sample_size):
    tight_means, wide_means = [], []
    for t in range(num_trials):
        sample_tight, sample_wide = [], []
        for i in range(sample_size):
            sample_tight.append(random.gauss(0, 1))
            sample_wide.append(random.gauss(0, 100))
        tight_means.append(sum(sample_tight)/len(sample_tight))
        wide_means.append(sum(sample_wide)/len(sample_wide))
    return tight_means, wide_means

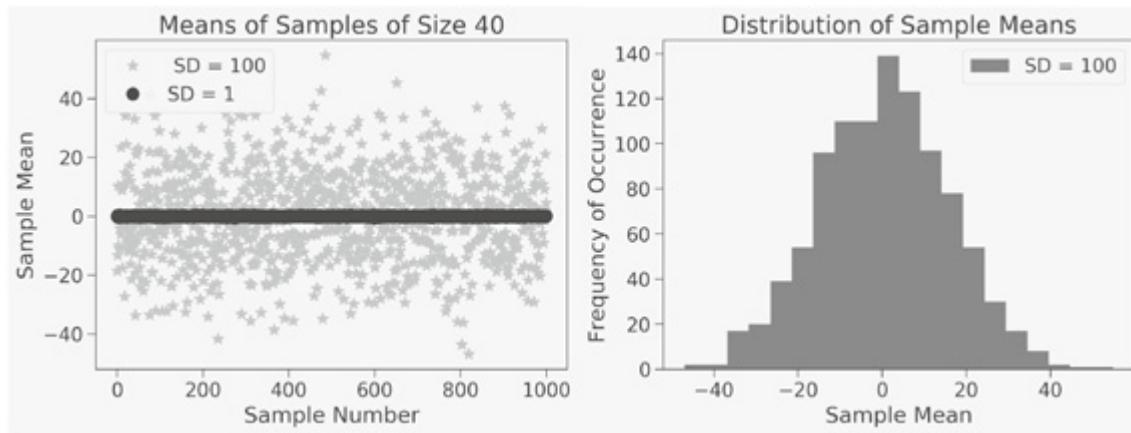
tight_means, wide_means = test_samples(1000, 40)
plt.plot(wide_means, 'y*', label = ' SD = 100')
plt.plot(tight_means, 'bo', label = 'SD = 1')
plt.xlabel('Sample Number')
plt.ylabel('Sample Mean')
plt.title('Means of Samples of Size ' + str(40))
plt.legend()

plt.figure()
plt.hist(wide_means, bins = 20, label = 'SD = 100')
plt.title('Distribution of Sample Means')
plt.xlabel('Sample Mean')
plt.ylabel('Frequency of Occurrence')
plt.legend()
```

[Figure 19-7](#) Compute and plot sample means

The left side of [Figure 19-8](#) shows the mean of each sample. As expected, when the population standard deviation is 1, the sample means are all near the population mean of 0, which is why no distinct circles are visible—they are so dense that they merge into

what appears to be a bar. In contrast, when the standard deviation of the population is 100, the sample means are scattered in a hard-to-discriminate pattern.



[Figure 19-8](#) Sample means

However, when we look at a histogram of the means when the standard deviation is 100, the right side of [Figure 19-8](#), something important emerges: the means form a distribution that resembles a normal distribution centered around 0. That the right side of [Figure 19-8](#) looks the way it does is not an accident. It is a consequence of the central limit theorem, the most famous theorem in all of probability and statistics.

---

## 19.2 The Central Limit Theorem

The central limit theorem explains why it is possible to use a single sample drawn from a population to estimate the variability of the means of a set of hypothetical samples drawn from the same population.

A version of the **central limit theorem (CLT)** to its friends was first published by Laplace in 1810, and then refined by Poisson in the 1820s. But the CLT as we know it today is a product of work done by a sequence of prominent mathematicians in the first half of the twentieth century.

Despite (or maybe because of) the impressive list of mathematicians who have worked on it, the CLT is really quite simple. It says that

- Given a set of sufficiently large samples drawn from the same population, the means of the samples (the sample means) will be approximately normally distributed.
- This normal distribution will have a mean close to the mean of the population.
- The variance (computed using `numpy.var`) of the sample means will be close to the variance of the population divided by the sample size.

Let's look at an example of the CLT in action. Imagine that you had a die with the property that each roll would yield a random real number between 0 and 5. The code in [Figure 19-9](#) simulates rolling such a die many times, prints the mean and variance (the function `variance` is defined in Figure 17-8), and then plots a histogram showing the probability of ranges of numbers getting rolled. It also simulates rolling 100 dice many times and plots (on the same figure) a histogram of the mean value of those 100 dice. The `hatch` keyword argument is used to visually distinguish one histogram from the other.

```

def plot_means(num_dice_per_trial, num_dice_thrown, num_bins,
              legend, color, style):
    means = []
    num_trials = num_dice_thrown//num_dice_per_trial
    for i in range(num_trials):
        vals = 0
        for j in range(num_dice_per_trial):
            vals += 5*random.random()
        means.append(vals/num_dice_per_trial)
    plt.hist(means, num_bins, color = color, label = legend,
             weights = np.array(len(means)*[1])/len(means),
             hatch = style)
    return sum(means)/len(means), np.var(means)

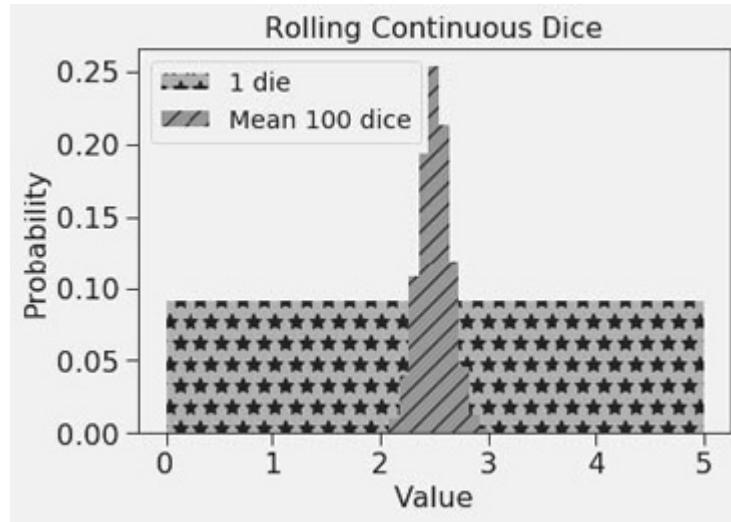
mean, var = plot_means(1, 1000000, 11, '1 die', 'y', '*')
print('Mean of rolling 1 die =', round(mean,4),
      'Variance =', round(var,4))
mean, var = plot_means(100, 1000000, 11,
                      'Mean 100 dice', 'c', '//')
print('Mean of rolling 100 dice =', round(mean, 4),
      'Variance =', round(var, 4))
plt.title('Rolling Continuous Dice')
plt.xlabel('Value')
plt.ylabel('Probability')
plt.legend(loc = 'upper left')

```

[Figure 19-9](#) Estimating the mean of a continuous die

The `weights` keyword is bound to an array of the same length as the first argument to `hist`, and is used to assign a weight to each element in the first argument. In the resulting histogram, each value in a bin contributes its associated weight towards the bin count (instead of the usual 1). In this example, we use `weights` to scale the y values to the relative (rather than absolute) size of each bin. Therefore, for each bin, the value on the y-axis is the probability of the mean falling within that bin.

When run, the code produced the plot in [Figure 19-10](#), and printed,



[Figure 19-10](#) An illustration of the CLT

```
Mean of rolling 1 die = 2.5003 Variance = 2.0814
Mean of rolling 100 dice = 2.4999 Variance = 0.0211
```

In each case the mean was quite close to the expected mean of 2.5. Since our die is fair, the probability distribution for one die is almost perfectly uniform,<sup>135</sup> i.e., very far from normal. However, when we look at the average value of 100 dice, the distribution is almost perfectly normal, with the peak including the expected mean. Furthermore, the variance of the mean of the 100 rolls is close to the variance of the value of a single roll divided by 100. All is as predicted by the CLT.

It's nice that the CLT seems to work, but what good is it? Perhaps it could prove useful in winning bar bets for those who drink in particularly nerdy bars. However, the primary value of the CLT is that it allows us to compute confidence levels and intervals even when the underlying population distribution is not normal. When we looked at confidence intervals in Section 17.4.2, we pointed out that the empirical rule is based on assumptions about the nature of the space being sampled. We assumed that

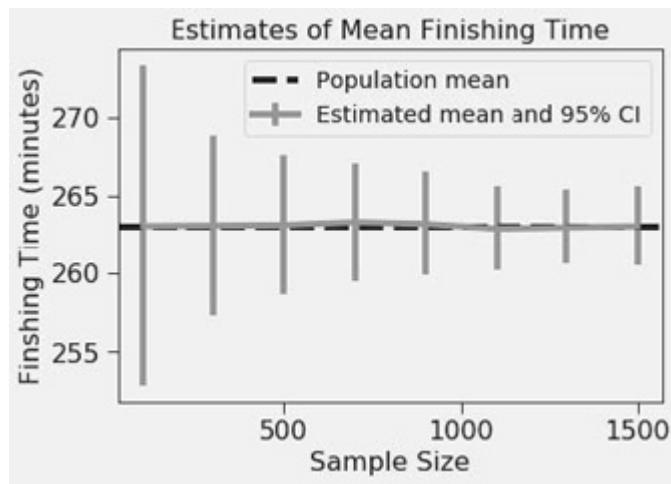
- The mean estimation error is 0.
- The distribution of the errors in the estimates is normal.

When these assumptions hold, the empirical rule for normal distributions provides a handy way to estimate confidence intervals and levels given the mean and standard deviation.

Let's return to the Boston Marathon example. The code in [Figure 19-11](#), which produced the plot in [Figure 19-12](#), draws 200 simple random samples for each of a variety of sample sizes. For each sample size, it computes the mean of each of the 200 samples; it then computes the mean and standard deviation of those means. Since the CLT tells us that the sample means will be normally distributed, we can use the standard deviation and the empirical rule to compute a 95% confidence interval for each sample size.

```
times = get_BM_data('bm_results2012.csv')['time']
mean_of_means, std_of_means = [], []
sample_sizes = range(100, 1501, 200)
for sample_size in sample_sizes:
    sample_means = []
    for t in range(200):
        sample = random.sample(times, sample_size)
        sample_means.append(sum(sample)/sample_size)
    mean_of_means.append(sum(sample_means)/len(sample_means))
    std_of_means.append(np.std(sample_means))
plt.errorbar(sample_sizes, mean_of_means, color = 'c',
             yerr = 1.96*np.array(std_of_means),
             label = 'Estimated mean and 95% CI')
plt.axhline(sum(times)/len(times), linestyle = '--', color = 'k',
            label = 'Population mean')
plt.title('Estimates of Mean Finishing Time')
plt.xlabel('Sample Size')
plt.ylabel('Finshing Time (minutes)')
plt.legend(loc = 'best')
```

[Figure 19-11](#) Produce plot with error bars



[Figure 19-12](#) Estimates of finishing times with error bars

As the plot in [Figure 19-12](#) shows, all of the estimates are reasonably close to the actual population mean. Notice, however, that the error in the estimated mean does not decrease monotonically with the size of the samples—the estimate using 700 examples happens to be worse than the estimate using 50 examples. What does change monotonically with the sample size is our confidence in our estimate of the mean. As the sample size grows from 100 to 1500, the confidence interval decreases from about  $\pm 15$  to about  $\pm 2.5$ . This is important. It's not good enough to get lucky and happen to get a good estimate. We need to know how much confidence to have in our estimate.

---

### 19.3 Standard Error of the Mean

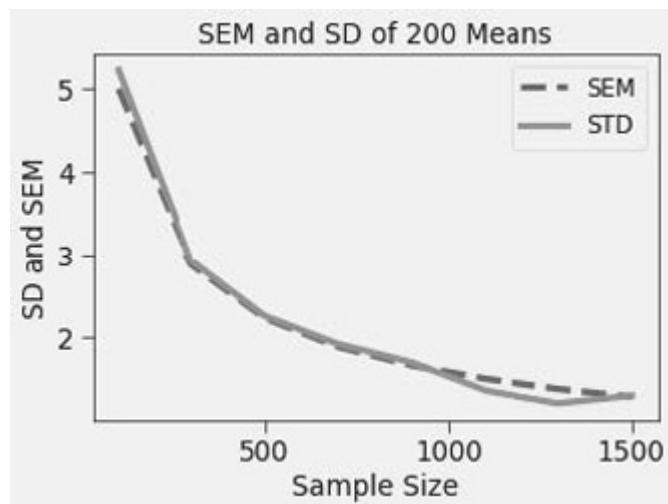
We just saw that if we chose 200 random samples of 1,500 competitors, we could, with 95% confidence, estimate the mean finishing time within a range of about five minutes. We did this using the standard deviation of the sample means. Unfortunately, since this involves using more total examples ( $200 * 1500 = 300,000$ ) than there were competitors, it doesn't seem like a useful result. We would have been better off computing the actual mean directly using the entire population. What we need is a way to estimate a confidence

interval using a single example. Enter the concept of the **standard error of the mean (SE or SEM)**.

The SEM for a sample of size  $n$  is the standard deviation of the means of an infinite number of samples of size  $n$  drawn from the same population. Unsurprisingly, it depends upon both  $n$  and  $\sigma$ , the standard deviation of the population:

$$SEM = \frac{\sigma}{\sqrt{n}}$$

[Figure 19-13](#) compares the SEM for the sample sizes used in [Figure 19-12](#) to the standard deviation of the means of the 200 samples we generated for each sample size.



[Figure 19-13](#) Standard error of the mean

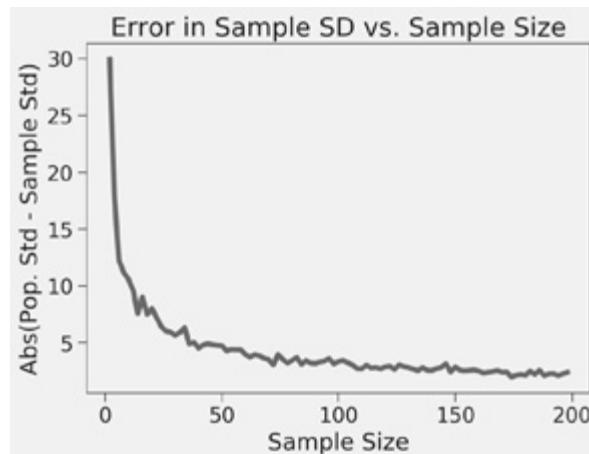
The actual standard deviations of the means of our 200 samples closely tracks the SE. Notice that both the SEM and the SD drop rapidly at the start and then more slowly as the sample size gets large. This is because the value depends upon the square root of the sample size. In other words, to cut the standard deviation in half, we need to quadruple the sample size.

Alas, if all we have is a single sample, we don't know the standard deviation of the population. Typically, we assume that the standard deviation of the sample, the sample standard deviation, is a reasonable proxy for the standard deviation of the population. This will be the case when the population distribution is not terribly skewed.

The code in [Figure 19-14](#) creates 100 samples of various sizes from the Boston Marathon data, and compares the mean standard deviation of the samples of each size to the standard deviation of the population. It produces the plot in [Figure 19-15](#).

```
times = get_BM_data('bm_results2012.csv')['time']
pos_std = np.std(times)
sample_sizes = range(2, 200, 2)
diffs_means = []
for sample_size in sample_sizes:
    diffs = []
    for t in range(100):
        diffs.append(abs(pos_std - np.std(random.sample(times,
                                                        sample_size))))
    diffs_means.append(sum(diffs)/len(diffs))
plt.plot(sample_sizes, diffs_means)
plt.xlabel('Sample Size')
plt.ylabel('Abs(Pop. Std - Sample Std)')
plt.title('Error in Sample SD vs. Sample Size')
```

[Figure 19-14](#). Sample standard deviation vs. population standard deviation



[Figure 19-15](#). Sample standard deviations

By the time the sample size reaches 100, the difference between the sample standard deviation and the population standard deviation is relatively small (about 1.2% of the actual mean finishing time).

In practice, people usually use the sample standard deviation in place of the (usually unknown) population standard deviation to estimate the SE. If the sample size is large enough,<sup>136</sup> and the population distribution is not too far from normal, it is safe to use this estimate to compute confidence intervals using the empirical rule.

What does this imply? If we take a single sample of say 200 runners, we can

- Compute the mean and standard deviation of that sample.
- Use the standard deviation of that sample to estimate the SE.
- Use the estimated SE to generate confidence intervals around the sample mean.

The code in [Figure 19-16](#) does this 10,000 times and then prints the fraction of times the sample mean is more than 1.96 estimated SEs from the population mean. (Recall that for a normal distribution 95% of the data falls within 1.96 standard deviations of the mean.)

```
times = get_BM_data('bm_results2012.csv')['time']
pop_mean = sum(times)/len(times)
sample_size = 200
num_bad = 0
for t in range(10000):
    sample = random.sample(times, sample_size)
    sample_mean = sum(sample)/sample_size
    se = np.std(sample)/sample_size**0.5
    if abs(pop_mean - sample_mean) > 1.96*se:
        num_bad += 1
print('Fraction outside 95% confidence interval =', num_bad/10000)
```

[Figure 19-16](#) Estimating the population mean 10,000 times

When the code is run it prints,

```
Fraction outside 95% confidence interval = 0.0533
```

That is pretty much what the theory predicts. Score one for the CLT!

---

## 19.4 Terms Introduced in Chapter

population

sample

sample size

probability sampling

simple random sample

stratified sampling

sample mean

population mean

central limit theorem

standard error (SE, SEM)

---

**133** In 2020, the race was cancelled for the first time in its history. The cause was the Covid-19 pandemic.

**134** In the world of sports, the question of whether athletes should be classified using “gender” or “sex” is a complicated and sensitive one. We have nothing to add to that discussion.

**135** “Almost” because we rolled the die a finite number of times.

**136** Don't you just love following instructions with phrases like, “choose a large enough sample.” Unfortunately, there is no simple recipe for choosing a sufficient sample size when you know little about the underlying population. Many statisticians say that a sample size of 30-40 is large enough when the population distribution is roughly normal. For smaller sample sizes, it is better to use something called the t-distribution to compute the size of the interval. The t-distribution is similar to a normal

distribution, but it has fatter tails, so the confidence intervals will be a bit wider.

# 20

## UNDERSTANDING EXPERIMENTAL DATA

This chapter is about understanding experimental data. We will make extensive use of plotting to visualize the data and show how to use linear regression to build a model of experimental data. We will also talk about the interplay between physical and computational experiments. We defer our discussion of how to draw valid statistical conclusions about data to Chapter 21.

---

### 20.1 The Behavior of Springs

Springs are wonderful things. When they are compressed or stretched by some force, they store energy. When that force is no longer applied, they release the stored energy. This property allows them to smooth the ride in cars, help mattresses conform to our bodies, retract seat belts, and launch projectiles.

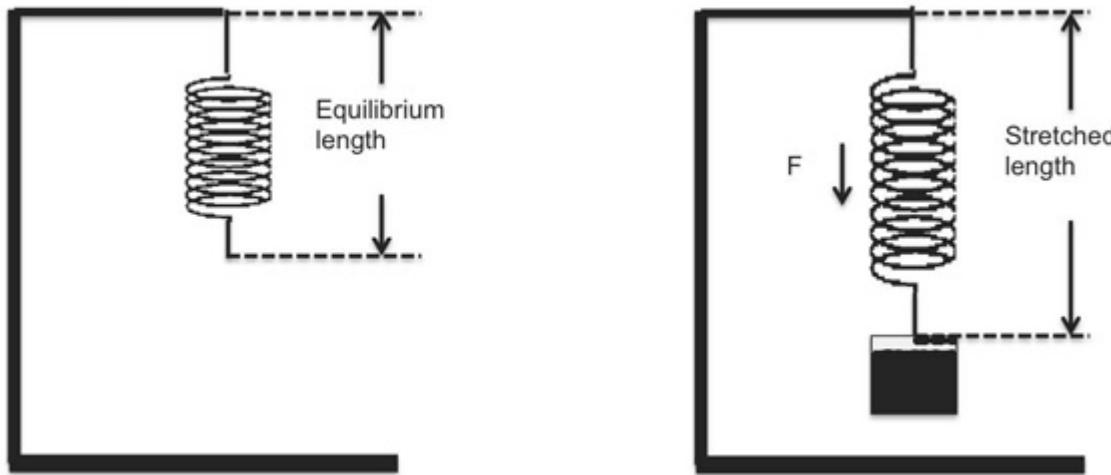
In 1676 the British physicist Robert Hooke formulated **Hooke's law** of elasticity: *Ut tensio, sic vis*, in English,  $F = -kx$ . In other words, the force  $F$  stored in a spring is linearly related to the distance the spring has been compressed (or stretched). (The minus sign indicates that the force exerted by the spring is in the opposite direction of the displacement.) Hooke's law holds for a wide variety of materials and systems, including many biological systems. Of course, it does not hold for an arbitrarily large force. All springs have an **elastic limit**, beyond which the law fails. Those of you who have stretched a Slinky too far know this all too well.

The constant of proportionality,  $k$ , is called the **spring constant**. If the spring is stiff (like the ones in the suspension of a

car or the limbs of an archer's bow),  $k$  is large. If the spring is weak, like the spring in a ballpoint pen,  $k$  is small.

Knowing the spring constant of a particular spring can be a matter of some import. The calibrations of both simple scales and atomic force microscopes depend upon knowing the spring constants of components. The mechanical behavior of a strand of DNA is related to the force required to compress it. The force with which a bow launches an arrow is related to the spring constant of its limbs. And so on.

Generations of physics students have learned to estimate spring constants using an experimental apparatus similar to one pictured in [Figure 20-1](#).



[Figure 20-1](#) A classic experiment

We start with a spring with no attached weight, and measure the distance to the bottom of the spring from the top of the stand. We then hang a known mass on the spring and wait for it to stop moving. At this point, the force stored in the spring is the force exerted on the spring by the weight hanging from it. This is the value of  $F$  in Hooke's law. We again measure the distance from the bottom of the spring to the top of the stand. The difference between this distance and the distance before we hung the weight becomes the value of  $x$  in Hooke's law.

We know that the force,  $F$ , being exerted on the spring is equal to the mass,  $m$ , multiplied by the acceleration due to gravity,  $g$  ( $9.81 \text{ m/s}^2$  is a pretty good approximation of  $g$  on the surface of this planet), so we substitute  $m * g$  for  $F$ . By simple algebra, we know that  $k = -(m * g) / x$ .

Suppose, for example, that  $m = 1\text{kg}$  and  $x = 0.1\text{m}$ , then

$$k = -\frac{1\text{kg} * 9.81\text{m/s}^2}{0.1\text{m}} = -\frac{9.81\text{N}}{0.1\text{m}} = -98.1\text{N/m}$$

According to this calculation, it will take  $98.1$  Newtons<sup>137</sup> of force to stretch the spring one meter.

This would all be well and good if

- We had complete confidence that we would conduct this experiment perfectly. In that case, we could take one measurement, perform the calculation, and know that we had found  $k$ . Unfortunately, experimental science hardly ever works this way.
- We could be sure that we were operating below the elastic limit of the spring.

A more robust experiment would be to hang a series of increasingly heavier weights on the spring, measure the stretch of the spring each time, and plot the results. We ran such an experiment, and typed the results into a file named `springData.csv`:

```
Distance (m), Mass (kg)
0.0865, 0.1
0.1015, 0.15
...
0.4416, 0.9
0.4304, 0.95
0.437, 1.0
```

The function in [Figure 20-2](#) reads data from a file such as the one we saved, and returns lists containing the distances and masses.

```
def get_data(input_file):
    with open(input_file, 'r') as data_file:
        distances = []
        masses = []
        data_file.readline() #ignore header
        for line in data_file:
            d, m = line.split(',')
            distances.append(float(d))
            masses.append(float(m))
    return (masses, distances)
```

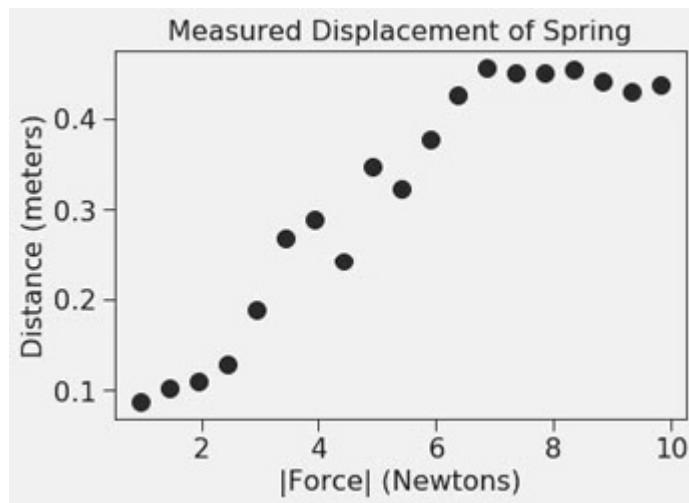
[Figure 20-2](#) Extracting the data from a file

The function in [Figure 20-3](#) uses `get_data` to extract the experimental data from the file and then produces the plot in [Figure 20-4](#).

```
def plot_data(input_file):
    masses, distances = get_data(input_file)
    distances = np.array(distances)
    masses = np.array(masses)
    forces = masses*9.81
    plt.plot(forces, distances, 'bo',
              label = 'Measured displacements')
    plt.title('Measured Displacement of Spring')
    plt.xlabel('|Force| (Newtons)')
    plt.ylabel('Distance (meters)')

plot_data('springData.csv')
```

[Figure 20-3](#) Plotting the data



[Figure 20-4](#) Displacement of spring

This is not what Hooke's law predicts. Hooke's law tells us that the distance should increase linearly with the mass, i.e., the points should lie on a straight line the slope of which is determined by the spring constant. Of course, we know that when we take real measurements, the experimental data are rarely a perfect match for the theory. Measurement error is to be expected, so we should expect the points to lie around a line rather than on it.

Still, it would be nice to see a line that represents our best guess of where the points would have been if we had no measurement error. The usual way to do this is to fit a line to the data.

### 20.1.1 Using Linear Regression to Find a Fit

Whenever we fit any curve (including a line) to data, we need some way to decide which curve is the best **fit** for the data. This means that we need to define an objective function that provides a quantitative assessment of how well the curve fits the data. Once we have such a function, finding the best fit can be formulated as finding a curve that minimizes (or maximizes) the value of that function, i.e., as an optimization problem (see Chapters 14 and 15).

The most commonly used objective function is called **least squares**. Let  $\text{observed}$  and  $\text{predicted}$  be vectors of equal length, where  $\text{observed}$  contains the measured points and  $\text{predicted}$  the corresponding data points on the proposed fit.

The objective function is then defined as:

$$\sum_{i=0}^{\text{len(observed)}-1} (\text{observed}[i] - \text{predicted}[i])^2$$

Squaring the difference between observed and predicted points makes large differences between observed and predicted points relatively more important than small differences. Squaring the difference also discards information about whether the difference is positive or negative.

How might we go about finding the best least-squares fit? One way is to use a successive approximation algorithm similar to the Newton–Raphson algorithm in Chapter 3. Alternatively, there are analytic solutions that are often applicable. But we don't have to implement either Newton–Raphson or an analytic solution, because `numpy` provides a built-in function, `polyfit`, that finds an approximation to the best least-squares fit. The call

```
np.polyfit(observed_x_vals, observed_y_vals, n)
```

finds the coefficients of a polynomial of degree `n` that provides a best least-squares fit for the set of points defined by the two arrays `observed_x_vals` and `observed_y_vals`. For example, the call

```
np.polyfit(observed_x_vals, observed_y_vals, 1)
```

will find a line described by the polynomial  $y = ax + b$ , where `a` is the slope of the line and `b` the y-intercept. In this case, the call returns an array with two floating-point values. Similarly, a parabola is described by the quadratic equation  $y = ax^2 + bx + c$ . Therefore, the call

```
np.polyfit(observed_x_vals, observed_y_vals, 2)
```

returns an array with three floating-point values.

The algorithm used by `polyfit` is called **linear regression**. This may seem a bit confusing, since we can use it to fit curves other than lines. Some authors do make a distinction between linear regression

(when the model is a line) and **polynomial regression** (when the model is a polynomial with degree greater than 1), but most do not.<sup>138</sup>

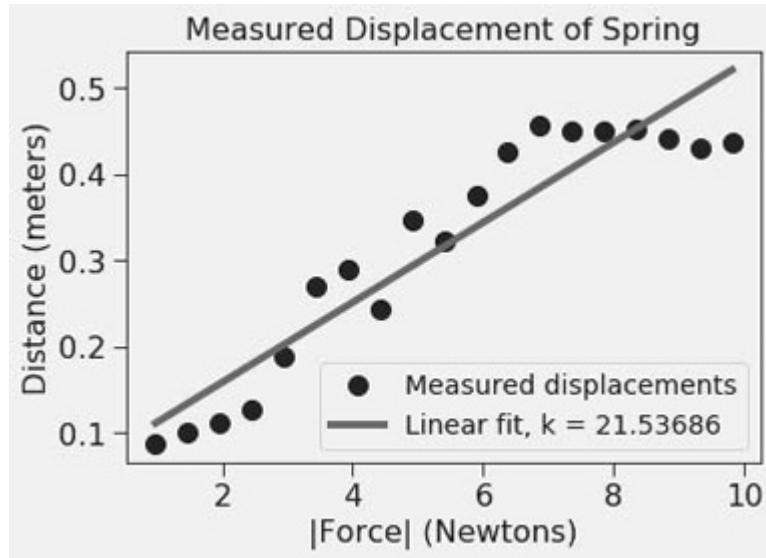
The function `fit_data` in [Figure 20-5](#) extends the `plot_data` function in [Figure 20-3](#) by adding a line that represents the best fit for the data. It uses `polyfit` to find the coefficients `a` and `b`, and then uses those coefficients to generate the predicted spring displacement for each force. Notice that there is an asymmetry in the way `forces` and `distance` are treated. The values in `forces` (which are derived from the mass suspended from the spring) are treated as independent, and used to produce the values in the dependent variable `predicted_distances` (a prediction of the displacements produced by suspending the mass).

```
def fit_data(input_file):
    masses, distances = get_data(input_file)
    distances = np.array(distances)
    forces = np.array(masses)*9.81
    plt.plot(forces, distances, 'ko',
              label = 'Measured displacements')
    plt.title('Measured Displacement of Spring')
    plt.xlabel('|Force| (Newtons)')
    plt.ylabel('Distance (meters)')
    #find linear fit
    a,b = np.polyfit(forces, distances, 1)
    predicted_distances = a*np.array(forces) + b
    k = 1.0/a #see explanation in text
    plt.plot(forces, predicted_distances,
              label = f'Linear fit, k = {k:.4f}')
    plt.legend(loc = 'best')
```

[Figure 20-5](#) Fitting a curve to data

The function also computes the spring constant, `k`. The slope of the line, `a`, is  $\Delta\text{distance}/\Delta\text{force}$ . The spring constant, on the other hand, is  $\Delta\text{force}/\Delta\text{distance}$ . Consequently, `k` is the inverse of `a`.

The call `fit_data('springData.csv')` produces the plot in [Figure 20-6](#).



[Figure 20-6](#) Measured points and linear model

It is interesting to observe that very few points actually lie on the least-squares fit. This is plausible because we are trying to minimize the sum of the squared errors, rather than maximize the number of points that lie on the line. Still, it doesn't look like a great fit. Let's try a cubic fit by adding to `fit_data` the code

```
#find cubic fit
fit = np.polyfit(forces, distances, 3)
predicted_distances = np.polyval(fit, forces)
plt.plot(forces, predicted_distances, 'k:', label = 'cubic
fit')
```

In this code, we have used the function `polyval` to generate the points associated with the cubic fit. This function takes two arguments: a sequence of polynomial coefficients and a sequence of values at which the polynomial is to be evaluated. The code fragments

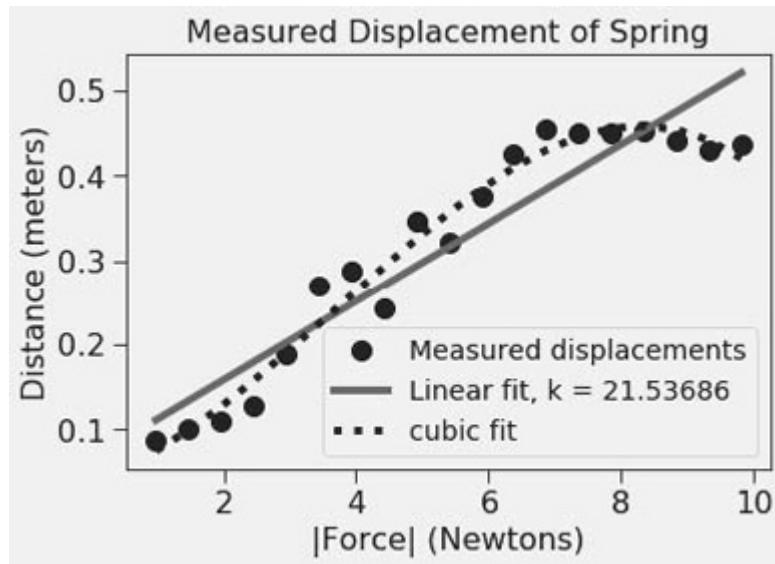
```
fit = np.polyfit(forces, distances, 3)
predicted_distances = np.polyval(fit, forces)
```

and

```
a,b,c,d = np.polyfit(forces, distances, 3)
predicted_distances = a*(forces**3) + b*forces**2 + c*forces
+ d
```

are equivalent.

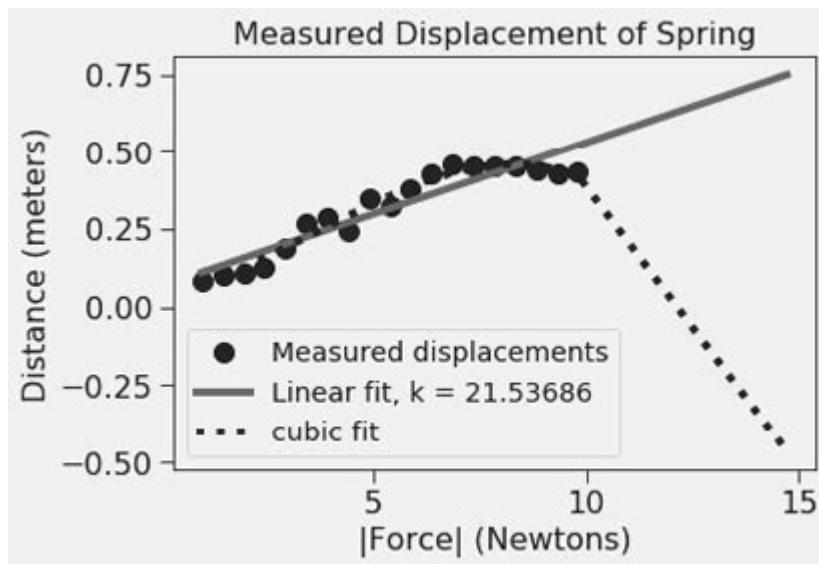
This produces the plot in [Figure 20-7](#). The cubic fit looks like a much better model of the data than the linear fit, but is it? Probably not.



[Figure 20-7](#) Linear and cubic fits

Both technical and popular articles frequently include plots like this that show both raw data and a curve fit to the data. All too often, however, the authors then go on to assume that the fitted curve is the description of the real situation, and the raw data merely an indication of experimental error. This can be dangerous.

Recall that we started with a theory that there should be a linear relationship between the  $x$  and  $y$  values, not a cubic one. Let's see what happens if we use our linear and cubic fits to predict where the point corresponding to hanging a 1.5kg weight would lie, [Figure 20-8](#).



[Figure 20-8](#) Using the model to make a prediction

Now the cubic fit doesn't look so good. In particular, it seems highly unlikely that by hanging a large weight on the spring we can cause the spring to rise above (the  $y$  value is negative) the bar from which it is suspended. What we have is an example of **overfitting**. Overfitting typically occurs when a model is excessively complex, e.g., it has too many parameters relative to the amount of data. When this happens, the fit can capture noise in the data rather than meaningful relationships. A model that has been overfit usually has poor predictive power, as seen in this example.

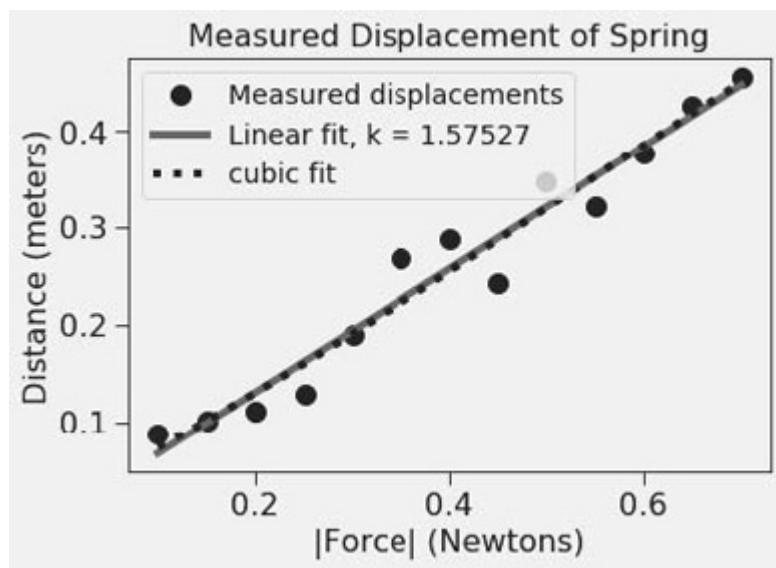
**Finger exercise:** Modify the code in [Figure 20-5](#) so that it produces the plot in [Figure 20-8](#).

Let's go back to the linear fit. For the moment, forget the line and study the raw data. Does anything about it seem odd? If we were to fit a line to the rightmost six points it would be nearly parallel to the x-axis. This seems to contradict Hooke's law—until we recall that Hooke's law holds only up to some elastic limit. Perhaps that limit is reached for this spring somewhere around  $7\text{N}$  (approximately  $0.7\text{kg}$ ).

Let's see what happens if we eliminate the last six points by replacing the second and third lines of the `fit_data` by

```
distances = np.array(distances[:-6])
masses = np.array(masses[:-6])
```

As [Figure 20-9](#) shows, eliminating those points certainly makes a difference:  $k$  has dropped dramatically and the linear and cubic fits are almost indistinguishable. But how do we know which of the two linear fits is a better representation of how our spring performs up to its elastic limit? We could use some statistical test to determine which line is a better fit for the data, but that would be beside the point. This is not a question that can be answered by statistics. After all we could throw out all the data except any two points and know that `polyfit` would find a line that would be a perfect fit for those two points. It is never appropriate to throw out experimental results merely to get a better fit.<sup>139</sup> Here we justified throwing out the rightmost points by appealing to the theory underlying Hooke's law, i.e., that springs have an elastic limit. That justification could not have been appropriately used to eliminate points elsewhere in the data.



[Figure 20-9](#) A model up to the elastic limit

---

## 20.2 The Behavior of Projectiles

Growing bored with merely stretching springs, we decided to use one of our springs to build a device capable of launching a projectile.<sup>140</sup> We used the device four times to fire a projectile at a target 30 yards

(1080 inches) from the launching point. Each time, we measured the height of the projectile at various distances from the launching point. The launching point and the target were at the same height, which we treated as 0.0 in our measurements.

The data was stored in a file, part of which shown in [Figure 20-10](#). The first column contains distances of the projectile from the target. The other columns contain the height of the projectile at that distance for each of the four trials. All of the measurements are in inches.

```
Distance,Trial1,Trial2,Trial3,Trial4
1080,0.0,0.0,0.0,0.0
1044,2.25,3.25,4.5,6.5
...
180,13.0,13.0,13.0,13.0
0,0.0,0.0,0.0,0.0
```

[Figure 20-10](#) Data from projectile experiment

The code in [Figure 20-11](#) was used to plot the mean altitude of the projectile in the four trials against the distance from the point of launch. It also plots the best linear and quadratic fits to those points. (In case you have forgotten the meaning of multiplying a list by an integer, the expression `[0]*len(distances)` produces a list of `len(distances)` 0's.)

```

def get_trajectory_data(file_name):
    distances = []
    heights1, heights2, heights3, heights4 = [],[],[],[]
    with open(file_name, 'r') as data_file:
        data_file.readline()
        for line in data_file:
            d, h1, h2, h3, h4 = line.split(',')
            distances.append(float(d))
            heights1.append(float(h1))
            heights2.append(float(h2))
            heights3.append(float(h3))
            heights4.append(float(h4))
    return (distances, [heights1, heights2, heights3, heights4])

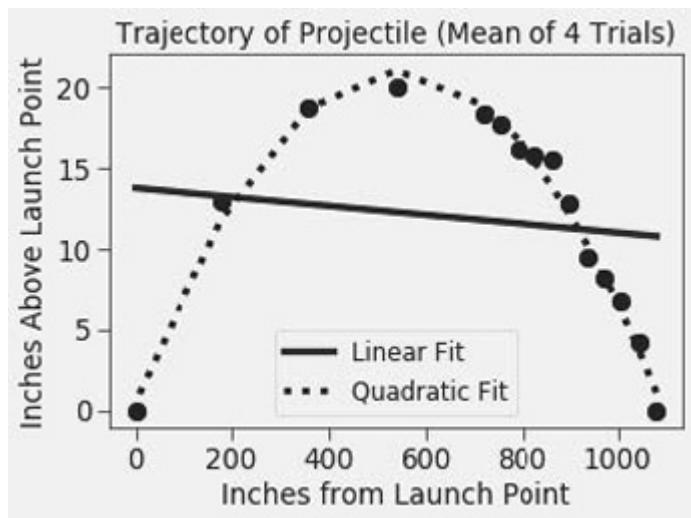
def process_trajectories(file_name):
    distances, heights = get_trajectory_data(file_name)
    num_trials = len(heights)
    distances = np.array(distances)
    #Get array containing mean height at each distance
    tot_heights = np.array([0]*len(distances))
    for h in heights:
        tot_heights = tot_heights + np.array(h)
    mean_heights = tot_heights/len(heights)
    plt.title('Trajectory of Projectile (Mean of '\
              + str(num_trials) + ' Trials)')
    plt.xlabel('Inches from Launch Point')
    plt.ylabel('Inches Above Launch Point')
    plt.plot(distances, mean_heights, 'ko')
    fit = np.polyfit(distances, mean_heights, 1)
    altitudes = np.polyval(fit, distances)
    plt.plot(distances, altitudes, 'b', label = 'Linear Fit')
    fit = np.polyfit(distances, mean_heights, 2)
    altitudes = np.polyval(fit, distances)
    plt.plot(distances, altitudes, 'k:', label = 'Quadratic Fit')
    plt.legend()

process_trajectories('launcherData.csv')

```

[Figure 20-11](#) Plotting the trajectory of a projectile

A quick look at the plot in [Figure 20-12](#) makes it quite clear that a quadratic fit is far better than a linear one.<sup>141</sup> But, in an absolute sense, just how bad a fit is the line and how good is the quadratic fit?



[Figure 20-12](#) Plot of trajectory

### 20.2.1 Coefficient of Determination

When we fit a curve to a set of data, we are finding a function that relates an independent variable (inches horizontally from the launch point in this example) to a predicted value of a dependent variable (inches above the launch point in this example). Asking about the **goodness of a fit** is equivalent to asking about the accuracy of these predictions. Recall that the fits were found by minimizing the mean square error. This suggests that we could evaluate the goodness of a fit by looking at the mean square error. The problem with that approach is that while there is a lower bound for the mean square error (0), there is no upper bound. This means that while the mean square error is useful for comparing the relative goodness of two fits to the same data, it is not particularly useful for getting a sense of the absolute goodness of a fit.

We can calculate the absolute goodness of a fit using the **coefficient of determination**, often written as  $R^2$ .<sup>142</sup> Let  $y_i$  be the  $i^{th}$  observed value,  $p_i$  be the corresponding value predicted by the model, and  $\mu$  be the mean of the observed values.

$$R^2 = 1 - \frac{\sum_i (y_i - p_i)^2}{\sum_i (y_i - \mu)^2}$$

By comparing the estimation errors (the numerator) with the variability of the original values (the denominator),  $R^2$  is intended to capture the proportion of variability (relative to the mean) in a data set that is accounted for by the statistical model provided by the fit. When the model being evaluated is produced by a linear regression, the value of  $R^2$  always lies between 0 and 1. If  $R^2 = 1$ , the model is a perfect fit to the data. If  $R^2 = 0$ , there is no relationship between the values predicted by the model and the way the data is distributed around the mean.

The code in [Figure 20-13](#) provides a straightforward implementation of this statistical measure. Its compactness stems from the expressiveness of the operations on numpy arrays. The expression `(predicted - measured)**2` subtracts the elements of one array from the elements of another, and then squares each element in the result. The expression `(measured - mean_of_measured)**2` subtracts the scalar value `mean_of_measured` from each element of the array `measured`, and then squares each element of the results.

```
def r_squared(measured, predicted):
    """Assumes measured a one-dimensional array of measured values
       predicted a one-dimensional array of predicted values
       Returns coefficient of determination"""
    estimated_error = ((predicted - measured)**2).sum()
    mean_of_measured = measured.sum()/len(measured)
    variability = ((measured - mean_of_measured)**2).sum()
    return 1 - estimated_error/variability
```

[Figure 20-13](#) Computing  $R^2$

When the lines of code

```
print('r**2 of linear fit =', r_squared(mean_heights,
altitudes))
```

and

```
print('r**2 of quadratic fit =', r_squared(mean_heights,
altitudes))
```

are inserted after the appropriate calls to `plt.plot` in `process_trajectories` (see [Figure 20-11](#)), they print

```
r**2 of linear fit = 0.0177433205440769  
r**2 of quadratic fit = 0.9857653692869693
```

Roughly speaking, this tells us that less than 2% of the variation in the measured data can be explained by the linear model, but more than 98% of the variation can be explained by the quadratic model.

### 20.2.2 Using a Computational Model

Now that we have what seems to be a good model of our data, we can use this model to help answer questions about our original data. One interesting question is the horizontal speed at which the projectile is traveling when it hits the target. We might use the following train of thought to design a computation that answers this question:

1. We know that the trajectory of the projectile is given by a formula of the form  $y = ax^2 + bx + c$ , i.e., it is a parabola. Since every parabola is symmetrical around its vertex, we know that its peak occurs halfway between the launch point and the target; call this distance `xMid`. The peak height, `yPeak`, is therefore given by  $yPeak = a*xMid^2 + b*xMid + c$ .
2. If we ignore air resistance (remember that no model is perfect), we can compute the amount of time it takes for the projectile to fall from `yPeak` to the height of the target, because that is purely a function of gravity. It is given by the equation  $t = \sqrt{(2 * yPeak)/g}$ .<sup>143</sup> This is also the amount of time it takes for the projectile to travel the horizontal distance from `xMid` to the target, because once it reaches the target it stops moving.
3. Given the time to go from `xMid` to the target, we can easily compute the average horizontal speed of the projectile over that interval. If we assume that the projectile was neither accelerating nor decelerating in the horizontal direction during that interval, we can use the average horizontal speed as an estimate of the horizontal speed when the projectile hits the target.

[Figure 20-14](#) implements this technique for estimating the horizontal velocity of the projectile.[144](#)

```
def get_horizontal_speed(quad_fit, min_x, max_x):
    """Assumes quad_fit has coefficients of a quadratic polynomial
       min_x and max_x are distances in inches
       Returns horizontal speed in feet per second"""
    inches_per_foot = 12
    x_mid = (max_x - min_x)/2
    a,b,c = quad_fit[0], quad_fit[1], quad_fit[2]
    y_peak = a*x_mid**2 + b*x_mid + c
    g = 32.16*inches_per_foot #accel. of gravity in inches/sec/sec
    t = (2*y_peak/g)**0.5 #time in seconds from peak to target
    print('Horizontal speed =',
          int(x_mid/(t*inches_per_foot)), 'feet/sec')
```

[Figure 20-14](#). Computing the horizontal speed of a projectile

When the line `get_horizontal_speed(fit, distances[-1], distances[0])` is inserted at the end of `process_trajectories` ([Figure 20-11](#)), it prints

```
Horizontal speed = 136 feet/sec
```

The sequence of steps we have just worked through follows a common pattern.

1. We started by performing an experiment to get some data about the behavior of a physical system.
2. We then used computation to find and evaluate the quality of a model of the behavior of the system.
3. Finally, we used some theory and analysis to design a simple computation to derive an interesting consequence of the model.

**Finger exercise:** In a vacuum, the speed of a falling object is defined by the equation  $v = v_0 + gt$ , where  $v_0$  is the initial velocity of the object,  $t$  is the number of seconds the object has been falling, and  $g$  is the gravitational constant, roughly  $9.8 \text{ m/sec}^2$  on the surface of

the Earth and  $3.711 \text{ m/sec}^2$  on Mars. A scientist measures the velocity of a falling object on an unknown planet. She does this by measuring the downward velocity of an object at different points in time. At time  $t_0$ , the object has an unknown velocity of  $v_0$ . Implement a function that fits a model to the time and velocity data and estimates  $g$  for that planet and  $v_0$  for the experiment. It should return its estimates for  $g$  and  $v_0$ , and also r-squared for the model.

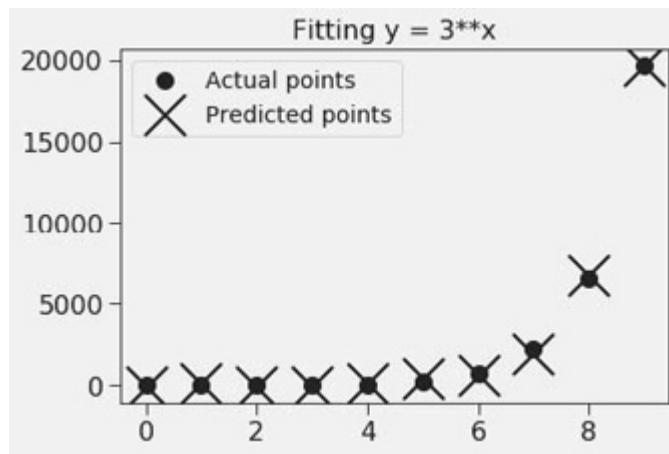
---

## 20.3 Fitting Exponentially Distributed Data

`Polyfit` uses linear regression to find a polynomial of a given degree that is the best least-squares fit for some data. It works well if the data can be directly approximated by a polynomial. But this is not always possible. Consider, for example, the simple exponential growth function  $y = 3^x$ . The code in [Figure 20-15](#) fits a fifth-degree polynomial to the first ten points and plots the results as shown in [Figure 20-16](#). It uses the function call `np.arange(10)`, which returns an array containing the integers 0-9. The parameter setting `markeredgewidth = 2` sets the width of the lines used in the marker.

```
vals = []
for i in range(10):
    vals.append(3**i)
plt.plot(vals, 'ko', label = 'Actual points')
xVals = np.arange(10)
fit = np.polyfit(xVals, vals, 5)
y_vals = np.polyval(fit, xVals)
plt.plot(y_vals, 'kx', label = 'Predicted points',
          markeredgewidth = 2, markersize = 25)
plt.title('Fitting y = 3**x')
plt.legend(loc = 'upper left')
```

[Figure 20-15](#) Fitting a polynomial curve to an exponential distribution



[Figure 20-16](#) Fitting an exponential distribution

The fit is clearly a good one for these data points. However, let's look at what the model predicts for  $3^{20}$ . When we add the code

```
print('Model predicts that 3**20 is roughly',
      np.polyval(fit, [3**20])[0])
print('Actual value of 3**20 is', 3**20)
```

to the end of [Figure 20-15](#), it prints,

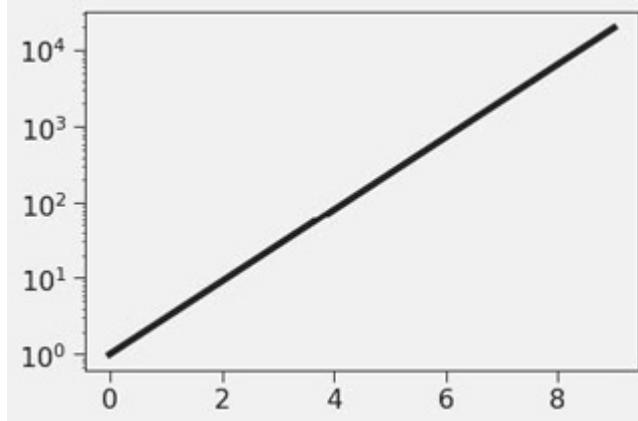
```
Model predicts that 3**20 is roughly 2.4547827637212492e+48
Actual value of 3**20 is 3486784401
```

Oh dear! Despite fitting the data, the model produced by `polyfit` is apparently not a good one. Is it because 5 was not the right degree? No. It is because no polynomial is a good fit for an exponential distribution. Does this mean that we cannot use `polyfit` to build a model of an exponential distribution? Fortunately, it does not, because we can use `polyfit` to find a curve that fits the original independent values and the log of the dependent values.

Consider the exponential sequence  $[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]$ . If we take the log base 2 of each value, we get the sequence  $[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]$ , i.e., a sequence that grows linearly. In fact, if a function  $y = f(x)$  exhibits exponential growth, the log (to any base) of  $f(x)$  grows linearly. This can be visualized by plotting an exponential function with a logarithmic y-axis. The code

```
x_vals, y_vals = [], []
for i in range(10):
    x_vals.append(i)
    y_vals.append(3**i)
plt.plot(x_vals, y_vals, 'k')
plt.semilogy()
```

produces the plot in [Figure 20-17](#).



[Figure 20-17](#) An exponential on a semilog plot

The fact that taking the log of an exponential function produces a linear function can be used to construct a model for an exponentially distributed set of data points, as illustrated by the code in [Figure 20-18](#). We use `polyfit` to find a curve that fits the `x` values and the log of the `y` values. Notice that we use yet another Python standard library module, `math`, which supplies a `log` function. (We could have used `np.log2`, but wanted to point out that `math` has a more general log function.)

```

def create_data(f, x_vals):
    """Assumes f is a function of one argument
       x_vals is an array of suitable arguments for f
       Returns array containing results of applying f to the
       elements of x_vals"""
    y_vals = []
    for i in x_vals:
        y_vals.append(f(x_vals[i]))
    return np.array(y_vals)

def fit_exp_data(x_vals, y_vals):
    """Assumes x_vals and y_vals arrays of numbers such that
       y_vals[i] == f(x_vals[i]), where f is an exponential function
       Returns a, b, base such that log(f(x), base) == ax + b"""
    log_vals = []
    for y in y_vals:
        log_vals.append(math.log(y, 2)) #get log base 2
    fit = np.polyfit(x_vals, log_vals, 1)
    return fit, 2

```

[Figure 20-18](#) Using `polyfit` to fit an exponential

When run, the code

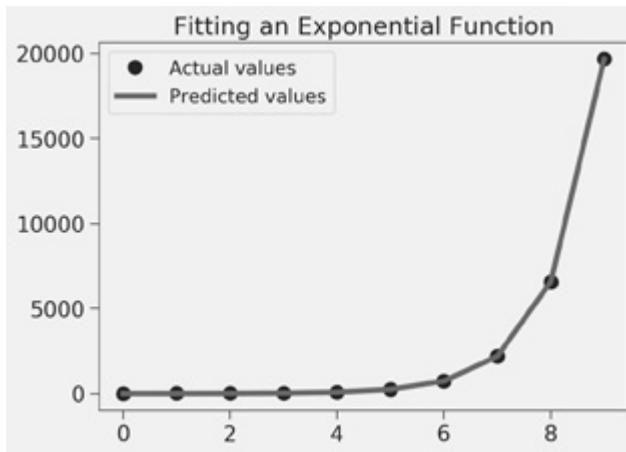
```

x_vals = range(10)
f = lambda x: 3**x
y_vals = create_data(f, x_vals)
plt.plot(x_vals, y_vals, 'ko', label = 'Actual values')
fit, base = fit_exp_data(x_vals, y_vals)
predictedy_vals = []
for x in x_vals:
    predictedy_vals.append(base**np.polyval(fit, x))
plt.plot(x_vals, predictedy_vals, label = 'Predicted
values')
plt.title('Fitting an Exponential Function')
plt.legend(loc = 'upper left')
#Look at a value for x not in original data
print('f(20) =', f(20))
print('Predicted value =', int(base**(np.polyval(fit,
[20]))))

```

produces the plot in [Figure 20-19](#), in which the actual values and the predicted values coincide. Moreover, when the model is tested on a value (20) that was not used to produce the fit, it prints

```
f(20) = 3486784401  
Predicted value = 3486784401
```



[Figure 20-19](#) A fit for an exponential function

This method of using `polyfit` to find a model for data works when the relationship can be described by an equation of the form  $y = \text{base}^{ax+b}$ . If used on data for which such a model is not a reasonably good fit, it will yield poor results.

To see this, let's create `yVals` using

```
f = lambda x: 3**x + x
```

The model now makes a poor prediction, printing

```
f(20) = 3486784421  
Predicted value = 2734037145
```

---

## 20.4 When Theory Is Missing

In this chapter, we have emphasized the interplay between theoretical, experimental, and computational science. Sometimes, however, we find ourselves with lots of interesting data, but little or no theory. In such cases, we often resort to using computational techniques to develop a theory by building a model that seems to fit the data.

In an ideal world, we would run a controlled experiment (e.g., hang weights from a spring), study the results, and retrospectively formulate a model consistent with those results. We would then run a new experiment (e.g., hang different weights from the same spring), and compare the results of that experiment to what the model predicted.

Unfortunately, in many cases it is impossible to run even one controlled experiment. Imagine, for example, building a model designed to shed light on how interest rates affect stock prices. Very few of us are in a position to set interest rates and see what happens. On the other hand, there is no shortage of relevant historical data.

In such situations, we can simulate a set of experiments by dividing the existing data into a **training set** and a **holdout set** to use as a **test set**. Without looking at the holdout set, we build a model that seems to explain the training set. For example, we find a curve that has a reasonable  $R^2$  for the training set. We then test that model on the holdout set. Most of the time the model will fit the training set more closely than it fits the holdout set. But if the model is a good one, it should fit the holdout set reasonably well. If it doesn't, the model should probably be discarded.

How do we choose the training set? We want it to be representative of the data set as a whole. One way to do this is to randomly choose the samples for the training set. If the data set is sufficiently large, this often works pretty well.

A related but slightly different way to check a model is to train on many randomly selected subsets of the original data and see how similar the models are to one another. If they are quite similar, then we can feel pretty good. This approach is known as **cross validation**.

Cross validation is discussed in more detail in Chapters 21 and 24.

---

## 20.5 Terms Introduced in Chapter

Hooke's law

elastic limit

spring constant  
curve fitting  
least squares  
linear regression  
polynomial regression  
overfitting  
goodness of fit  
coefficient of determination ( $R^2$ )  
training set  
test set  
holdout set  
cross validation

---

**137** The Newton, written  $N$ , is the standard international unit for measuring force. It is the amount of force needed to accelerate a mass of one kilogram at a rate of one meter per second per second. A Slinky, by the way, has a spring constant of approximately  $1N/m$ .

**138** The reason they do not is that although polynomial regression fits a nonlinear model to the data, the model is linear in the unknown parameters that it estimates.

**139** Which isn't to say that people never do.

**140** A projectile is an object that is propelled through space by the exertion of a force that stops after the projectile is launched. In the interest of public safety, we will not describe the launching device used in this experiment. Suffice it to say that it was awesome.

**141** Don't be misled by this plot into thinking that the projectile had a steep angle of ascent. It only looks that way because of the

difference in scale between the vertical and horizontal axes on the plot.

142 There are other definitions of the coefficient of determination. The definition supplied here is usually used to evaluate the quality of a fit produced by a linear regression.

143 This equation can be derived from first principles, but it is easier to just look it up. We found it at

[http://en.wikipedia.org/wiki/Equations\\_for\\_a\\_falling\\_body](http://en.wikipedia.org/wiki/Equations_for_a_falling_body).

144 The vertical component of the velocity is also easily estimated, since it is merely the product of the  $g$  and  $t$  in Figure 20-14.

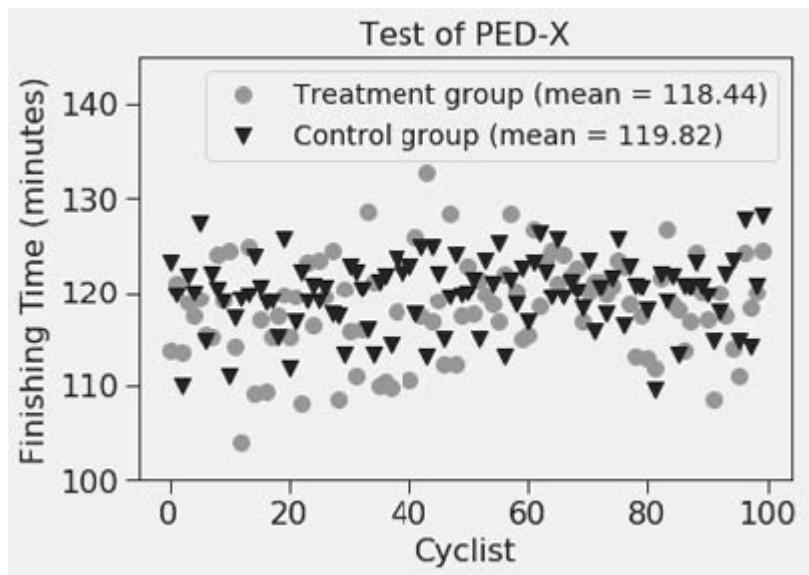
# 21

## RANDOMIZED TRIALS AND HYPOTHESIS CHECKING

Dr. X invented a drug, PED-X, designed to help professional bicycle racers ride faster. When he tried to market it, the racers insisted that Dr. X demonstrate that his drug was superior to PED-Y, the banned drug that they had been using for years. Dr. X raised money from some investors and launched a **randomized trial**.

He persuaded 200 professional cyclists to participate in his trial. He then divided them randomly into two groups: treatment and control. Each member of the **treatment group** received a dose of PED-X. Members of the **control group** were told that they were being given a dose of PED-X, but were instead given a dose of PED-Y.

Each cyclist was asked to bike 50 miles as fast as possible. The finishing times for each group were normally distributed. The mean finishing time of the treatment group was 118.61 minutes, and that of the control group was 120.62 minutes. [Figure 21-1](#) shows the time for each cyclist.



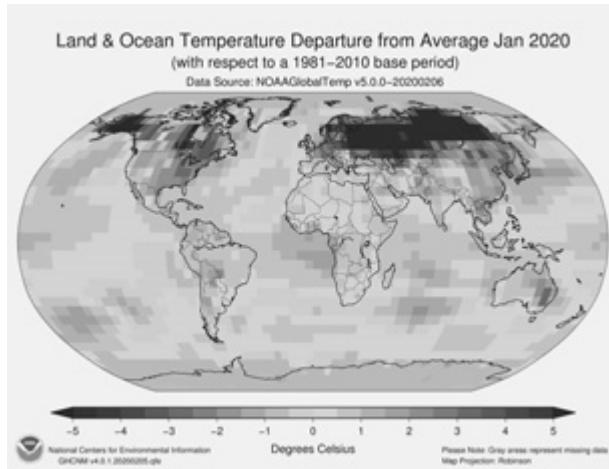
[Figure 21-1](#) Finishing times for cyclists

Dr. X was elated until he ran into a statistician who pointed out that it was almost inevitable that one of the two groups would have a lower mean than the other, and perhaps the difference in means was merely a random occurrence. When she saw the crestfallen look on the scientist's face, the statistician offered to show him how to check the statistical significance of his study.

---

## 21.1 Checking Significance

In any experiment that involves drawing samples at random from a population, there is always the possibility that an observed effect occurred purely by chance. [Figure 21-2](#) is a visualization of how temperatures in January of 2020 varied from the average temperatures in January from 1981 to 2010. Now, imagine that you constructed a sample by choosing 20 random spots on the planet, and then discovered that the mean change in temperature for the sample was +1 degree Celsius. What is the probability that the observed change in mean temperature was an artifact of the sites you happened to sample rather than an indication that the planet as a whole is warming? Answering this kind of question is what **statistical significance** is all about.



[Figure 21-2](#) January 2020 temperature difference from the 1981-2010 average<sup>145</sup>

In the early part of the twentieth century, Ronald Fisher developed an approach to statistical **hypothesis testing** that has become the most common approach for evaluating the probability an observed effect occurred purely by chance. Fisher says he invented the method in response to a claim by Dr. Muriel Bristol-Roach that when she drank tea with milk in it, she could detect whether the tea or the milk was poured into the teacup first. Fisher challenged her to a “tea test” in which she was given eight cups of tea (four for each order of adding tea and milk), and asked to identify those cups into which the tea had been poured before the milk. She did this perfectly. Fisher then calculated the probability of her having done this purely by chance. As we saw in Section 17.4.4,  $\binom{8}{4} = 70$ , i.e., there are 70 ways to choose 4 cups out of 8. Since only one of these 70 combinations includes all 4 cups in which the tea was poured first, Fisher calculated that the probability of Dr. Bristol-Roach having chosen correctly by pure luck was  $\frac{1}{70} \approx 0.014$ . From this, he concluded that it was highly unlikely her success could be attributed to luck.

Fisher's approach to significance testing can be summarized as

1. State a **null hypothesis** and an **alternative hypothesis**.

The null hypothesis is that the “treatment” has no interesting effect. For the “tea test,” the null hypothesis was that Dr. Bristol-Roach could not taste the difference. The alternative hypothesis is a hypothesis that can be true only if the null

hypothesis is false, e.g., that Dr. Bristol-Roach could taste the difference.<sup>[146](#)</sup>

2. Understand statistical assumptions about the sample being evaluated. For the “tea test” Fisher assumed that Dr. Bristol-Roach was making independent decisions for each cup.
3. Compute a relevant **test statistic**. In this case, the test statistic was the fraction of correct answers given by Dr. Bristol-Roach.
4. Derive the probability of that test statistic under the null hypothesis. In this case, it is the probability of getting all of the cups right by accident, i.e., 0.014.
5. Decide whether that probability is sufficiently small that you are willing to assume the null hypothesis is false, i.e., to **reject** the null hypothesis. Common values for the rejection level, which should be chosen in advance, are 0.05 and 0.01.

Returning to our cyclists, imagine that the times for the treatment and control groups were samples drawn from infinite populations of finishing times for PED-X users and PED-Y users. The null hypothesis for this experiment is that the means of those two larger populations are the same, i.e., the difference between the population mean of the treatment group and the population mean of the control group is 0. The alternative hypothesis is that they are not the same, i.e., the difference in means is not equal to 0.

Next, we go about trying to reject the null hypothesis. We choose a threshold,  $\alpha$ , for statistical significance, and try to show that the probability of the data having been drawn from distributions consistent with the null hypothesis is less than  $\alpha$ . We then say that we can reject the null hypothesis with confidence  $\alpha$ , and accept the negation of the null hypothesis with probability  $1 - \alpha$ .

The choice of  $\alpha$  affects the kind of errors we make. The larger  $\alpha$ , the more often we will reject a null hypothesis that is actually true. These are known as **type I errors**. When  $\alpha$  is smaller, we will more often accept a null hypothesis that is actually false. These are known as **type II errors**.

Typically, people choose  $\alpha = 0.05$ . However, depending upon the consequences of being wrong, it might be preferable to choose a

smaller or larger  $\alpha$ . Imagine, for example, that the null hypothesis is that there is no difference in the rate of premature death between those taking PED-X and those taking PED-Y. We might well want to choose a small  $\alpha$ , say 0.001, as the basis for rejecting that hypothesis before deciding whether one drug was safer than the other. On the other hand, if the null hypothesis were that there is no difference in the performance-enhancing effect of PED-X and PED-Y, we might comfortably choose a pretty large  $\alpha$ .<sup>147</sup>

The next step is to compute the test statistic. The most common test statistic is the **t-statistic**. The t-statistic tells us how different, measured in units of standard error, the estimate derived from the data is from the null hypothesis. The larger the t-statistic, the more likely the null hypothesis can be rejected. For our example, the t-statistic tells us how many standard errors the difference in the two means ( $118.44 - 119.82 = -1.38$ ) is from 0. The t-statistic for our PED-X example is -2.11 (you'll see how to compute this shortly). What does this mean? How do we use it?

We use the t-statistic in much the same way we use the number of standard deviations from the mean to compute confidence intervals (see Section 17.4.2). Recall that for all normal distributions, the probability of an example lying within a fixed number of standard deviations of the mean is fixed. Here we do something slightly more complex that accounts for the number of samples used to compute the standard error. Instead of assuming a normal distribution, we assume a **t-distribution**.

T-distributions were first described, in 1908, by William Gosset, a statistician working for the Arthur Guinness and Son brewery.<sup>148</sup> The t-distribution is actually a family of distributions, since the shape of the distribution depends upon the degrees of freedom in the sample.

The **degrees of freedom** describes the amount of independent information used to derive the t-statistic. In general, we can think of degrees of freedom as the number of independent observations in a sample that are available to estimate some statistic about the population from which that sample is drawn.

A t-distribution resembles a normal distribution, and the larger the degrees of freedom, the closer it is to a normal distribution. For small degrees of freedom, the t-distributions have notably fatter tails

than normal distributions. For degrees of freedom of 30 or more, t-distributions are very close to normal.

Now, let's use the sample variance to estimate the population variance. Consider a sample containing the three examples 100, 200, and 300. Recall that

$$\text{variance}(X) = \frac{\sum_{x \in X} (x - \mu)^2}{|X|}$$

so the variance of our sample is

$$\frac{(100 - 200)^2 + (200 - 200)^2 + (300 - 200)^2}{3}$$

It might appear that we are using three independent pieces of information, but we are not. The three terms in the numerator are not independent of each other, because all three observations were used to compute the mean of the sample of 200 riders. The degrees of freedom is 2, since if we know the mean and any two of the three observations, the value of the third observation is fixed.

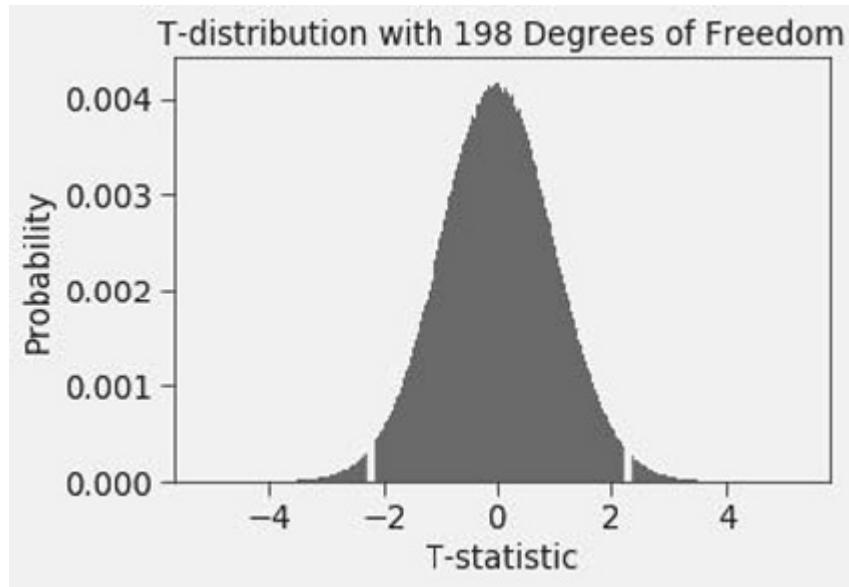
The larger the degrees of freedom, the higher the probability that the sample statistic is representative of the population. The degrees of freedom in a t-statistic computed from a single sample is one less than the sample size, because the mean of the sample is used in calculating the t-statistic. If two samples are used, the degrees of freedom is two less than the sum of the sample sizes, because the mean of each sample is used in calculating the t-statistic. For example, for the PED-X/PED-Y experiment, the degrees of freedom is 198.

Given the degrees of freedom, we can draw a plot showing the appropriate t-distribution, and then see where the t-statistic we have computed for our PED-X example lies on the distribution. The code in [Figure 21-3](#) does that, and produces the plot in [Figure 21-4](#). The code first uses the function `scipy.random.standard_t` to generate many examples drawn from a t-distribution with 198 degrees of

freedom. It then draws white lines at the t-statistic and the negative of the t-statistic for the PED-X sample.

```
t_stat = -2.26 #t-statistic for PED-X example
t_dist = []
num_bins = 1000
for i in range(10000000):
    t_dist.append(scipy.random.standard_t(198))
plt.hist(t_dist, bins = num_bins,
         weights = np.array(len(t_dist)*[1.0])/len(t_dist))
plt.axvline(t_stat, color = 'w')
plt.axvline(-t_stat, color = 'w')
plt.title('T-distribution with 198 Degrees of Freedom')
plt.xlabel('T-statistic')
plt.ylabel('Probability')
```

[Figure 21-3](#) Plotting a t-distribution



[Figure 21-4](#) Visualizing the t-statistic

The sum of the fractions of the area of the histogram to the left and right of the white lines equals the probability of getting a value at least as extreme as the observed value if

- the sample is representative of the population, and

- the null hypothesis is true.

We need to look at both tails because our null hypothesis is that the population means are equal. So, the test should fail if the mean of the treatment group is either statistically significantly larger or smaller than the mean of the control group.

Under the assumption that the null hypothesis holds, the probability of getting a value at least as extreme as the observed value is called a **p-value**. For our PED-X example, the p-value is the probability of seeing a difference in the means at least as large as the observed difference, under the assumption that the actual population means of the treatment and controls are identical.

It may seem odd that p-values tell us something about the probability of an event occurring if the null hypothesis holds, when what we are usually hoping is that the null hypothesis doesn't hold. However, it is not so different in character from the classic **scientific method**, which is based upon designing experiments that have the potential to refute a hypothesis. The code in [Figure 21-5](#) computes and prints the t-statistic and p-value for our two samples, one containing the times of the control group and the other the times of the treatment group. The library function `scipy.stats.ttest_ind` performs a two-tailed two-sample **t-test** and returns both the t-statistic and the p-value. Setting the parameter `equal_var` to `False` indicates that we don't know whether the two populations have the same variance.

```
control_mean = round(sum(control_times)/len(control_times), 2)
treatment_mean = round(sum(treatment_times)/len(treatment_times), 2)
print('Treatment mean - control mean =',
      round(treatment_mean - control_mean, 2), 'minutes')
two_sample_test = scipy.stats.ttest_ind(treatment_times,
                                         control_times,
                                         equal_var = False)
print('The t-statistic from two-sample test is',
      round(two_sample_test[0], 2))
print('The p-value from two-sample test is',
      round(two_sample_test[1], 2))
```

[Figure 21-5](#) Compute and print t-statistic and p-value

When we run the code, it reports

```
Treatment mean - control mean = -1.38 minutes
The t-statistic from two-sample test is -2.11
The p-value from two-sample test is 0.04
```

“Yes,” Dr. X crowed, “it seems that the probability of PED-X being no better than PED-Y is only 4%, and therefore the probability that PED-X has an effect is 96%. Let the cash registers start ringing.” Alas, his elation lasted only until he read the next section of this chapter.

---

## 21.2 Beware of P-values

It is way too easy to read something into a p-value that it doesn't really imply. It is tempting to think of a p-value as the probability of the null hypothesis being true. But this is not what it actually is.

The null hypothesis is analogous to a defendant in the Anglo-American criminal justice system. That system is based on a principle called “presumption of innocence,” i.e., innocent until proven guilty. Analogously, we assume that the null hypothesis is true unless we see enough evidence to the contrary. In a trial, a jury can rule that a defendant is “guilty” or “not guilty.” A “not guilty” verdict implies that the evidence was insufficient to convince the jury that the defendant was guilty “beyond a reasonable doubt.”<sup>149</sup> Think of it as equivalent to “guilt was not proven.” A verdict of “not guilty” does not imply that the evidence was sufficient to convince the jury that the defendant was innocent. And it says nothing about what the jury would have concluded had it seen different evidence. Think of a p-value as a jury verdict where the standard “beyond a reasonable doubt” is defined by  $\alpha$ , and the evidence is the data from which the t-statistic was constructed.

A small p-value indicates that a particular sample is unlikely if the null hypothesis is true. It is akin to a jury concluding that it was unlikely that it would have been presented with this set of evidence if the defendant were innocent, and therefore reaching a guilty verdict. Of course, that doesn't mean that the defendant is actually guilty. Perhaps the jury was presented with misleading evidence. Analogously, a low p-value might be attributable to the null

hypothesis actually being false, or it could simply be that the sample is unrepresentative of the population from which it is drawn, i.e., the evidence is misleading.

As you might expect, Dr. X staunchly claimed that his experiment showed that the null hypothesis was probably false. Dr. Y insisted that the low p-value was probably attributable to an unrepresentative sample and funded another experiment of the same size as Dr. X's. When the statistics were computed using the samples from her experiment, the code printed

```
Treatment mean - control mean = 0.18 minutes  
The t-statistic from two-sample test is -0.27  
The p-value from two-sample test is 0.78
```

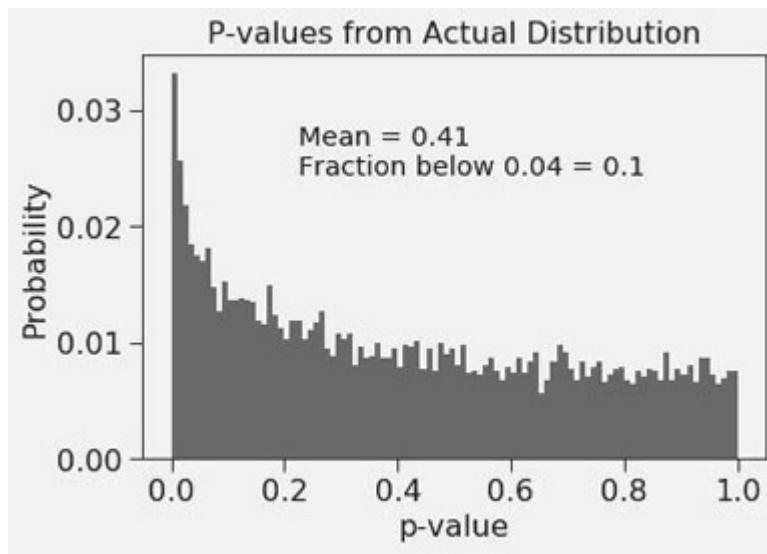
This p-value is more than 17 times larger than that obtained from Dr. X's experiment, and certainly provides no reason to doubt the null hypothesis. Confusion reigned. But we can clear it up!

You may not be surprised to discover that this is not a true story —after all, the idea of a cyclist taking a performance-enhancing drug strains credulity. In fact, the samples for the experiments were generated by the code in [Figure 21-6](#).

```
random.seed(148)  
treatment_dist = (119.5, 5.0)  
control_dist = (120, 4.0)  
sample_size = 100  
treatment_times, control_times = [], []  
for s in range(sample_size):  
    treatment_times.append(random.gauss(treatment_dist[0],  
                                         treatment_dist[1]))  
    control_times.append(random.gauss(control_dist[0],  
                                      control_dist[1])))
```

[Figure 21-6](#) Code for generating racing examples

Since the experiment is purely computational, we can run it many times to get many different samples. When we generated 10,000 pairs of samples (one from each distribution) and plotted the probability of the p-values, we got the plot in [Figure 21-7](#).



[Figure 21-7](#) Probability of p-values

Since about 10% of the p-values lie below 0.04, it is not terribly surprising that an experiment happened to show significance at the 4% level. On the other hand, that the second experiment yielded a completely different result is also not surprising. What does seem surprising is that given we know that the means of the two distributions are actually different, we get a result that is significant at the 5% level only about 12% of the time. Roughly 88% of the time we would fail to reject a fallacious null hypothesis at the 5% level.

That p-values can be unreliable indicators of whether it is truly appropriate to reject a null hypothesis is one of the reasons so many of the results appearing in the scientific literature cannot be reproduced by other scientists. One problem is that there is a strong relationship between the **study power** (the size of the samples) and the credibility of the statistical finding.<sup>150</sup> If we increase the sample size in our example to 3000, we fail to reject the fallacious null hypothesis only about 1% of the time.

Why are so many studies under-powered? If we were truly running an experiment with people (rather than a simulation), it would be 20 times more expensive to draw samples of size 2000 than samples of size 100.

The problem of sample size is an intrinsic attribute of what is called the frequentist approach to statistics. In Section 21.7, we

discuss an alternative approach that attempts to mitigate this problem.

---

## 21.3 One-tail and One-sample Tests

Thus far in this chapter, we have looked only at two-tailed two-sample tests. Sometimes, it is more appropriate to use a **one-tailed** and/or a **one-sample** t-test.

Let's first consider a one-tailed two-sample test. In our two-tailed test of the relative effectiveness of PED-X and PED-Y, we considered three cases: 1) they were equally effective, 2) PED-X was more effective than PED-Y, and 3) PED-Y was more effective than PED-X. The goal was to reject the null hypothesis (case 1) by arguing that if it were true, it would be unlikely to see as large a difference as observed in the means of the PED-X and PED-Y samples.

Suppose, however, that PED-X were substantially less expensive than PED-Y. To find a market for his compound, Dr. X would only need to show that PED-X is at least as effective as PED-Y. One way to think about this is that we want to reject the hypothesis that the means are equal or that the PED-X mean is larger. Note that this is strictly weaker than the hypothesis that the means are equal. (Hypothesis  $A$  is strictly weaker than hypothesis  $B$ , if whenever  $B$  is true  $A$  is true, but not vice versa.)

To do this, we start with a two-sample test with the original null hypothesis computed by the code in [Figure 21-5](#). It printed

```
Treatment mean - control mean = -1.38 minutes
The t-statistic from two-sample test is -2.11
The p-value from two-sample test is 0.04
```

allowing us to reject the null hypothesis at about the 4% level.

How about our weaker hypothesis? Recall [Figure 21-4](#). We observed that under the assumption that the null hypothesis holds, the sum of the fractions of the areas of the histogram to the left and right of the white lines equals the probability of getting a value at least as extreme as the observed value. However, to reject our weaker hypothesis we don't need to take into account the area under the left tail, because that corresponds to PED-X being more effective than PED-Y (a negative time difference), and we're interested only in

rejecting the hypothesis that PED-X is less effective. That is, we can do a one-tailed test.

Since the t-distribution is symmetric, to get the value for a one-tailed test we divide the p-value from the two-tailed test in half. So the p-value for the one-tailed test is  $0.02$ . This allows us to reject our weaker hypothesis at the  $2\%$  level, something that we could not do using the two-tailed test.

Because a one-tailed test provides more power to detect an effect, it is tempting to use a one-tailed test whenever one has a hypothesis about the direction of an effect. This is usually not a good idea. A one-tailed test is appropriate only if the consequences of missing an effect in the untested direction are negligible.

Now let's look at a one-sample test. Suppose that, after years of experience of people using PED-Y, it was well established that the mean time for a racer on PED-Y to complete a 50-mile course is  $120$  minutes. To discover whether PED-X had a different effect than PED-Y, we would test the null hypothesis that the mean time for a single PED-X sample is equal to  $120$ . We can do this using the function `scipy.stats.ttest_1samp`, which takes as arguments a single sample and the population mean against which it is compared. It returns a tuple containing the t-statistic and p-value. For example, if we append to the end of the code in [Figure 21-5](#) the code

```
one_sample_test = scipy.stats.ttest_1samp(treatment_times,  
120)  
print('The t-statistic from one-sample test is',  
one_sample_test[0])  
print('The p-value from one-sample test is',  
one_sample_test[1])
```

it prints

```
The t-statistic from one-sample test is -2.9646117910591645  
The p-value from one-sample test is 0.0037972083811954023
```

It is not surprising that the p-value is smaller than the one we got using the two-sample two-tail test. By assuming that we know one of the two means, we have removed a source of uncertainty.

So, after all this, what have we learned from our statistical analysis of PED-X and PED-Y? Even though there is a difference in the expected performance of PED-X and PED-Y users, no finite

sample of PED-X and PED-Y users is guaranteed to reveal that difference. Moreover, because the difference in the expected means is small (less than half a percent), it is unlikely that an experiment of the size Dr. X ran (100 riders in each group) will yield evidence that would allow us to conclude at the 95% confidence level that there is a difference in means. We could increase the likelihood of getting a result that is statistically significant at the 95% level by using a one-tailed test, but that would be misleading, because we have no reason to assume that PED-X is not less effective than PED-Y.

---

## 21.4 Significant or Not?

Lyndsay and John have wasted an inordinate amount of time over the last several years playing a game called Words with Friends. They have played each other 1,273 times, and Lyndsay has won 666 of those games, prompting her to boast, “I’m way better at this game than you are.” John asserted that Lyndsay’s claim was nonsense, and that the difference in wins could be (and probably should be) attributed entirely to luck.

John, who had recently read a book about statistics, proposed the following way to find out whether it was reasonable to attribute Lyndsay’s relative success to skill:

- Treat each of the 1,273 games as an experiment returning 1 if Lyndsay was the victor and 0 if she was not.
- Choose the null hypothesis that the mean value of those experiments is 0.5.
- Perform a two-tailed one-sample test for that null hypothesis.

When he ran the code

```
num_games = 1273
lyndsay_wins = 666
outcomes = [1.0]*lyndsay_wins + [0.0]*(num_games -
lyndsay_wins)
print('The p-value from a one-sample test is',
      scipy.stats.ttest_1samp(outcomes, 0.5)[1])
```

it printed

The p-value from a one-sample test is 0.0982205871243577

prompting John to claim that the difference wasn't even close to being significant at the 5% level.

Lyndsay, who had not studied statistics, but had read Chapter 18 of this book, was not satisfied. “Let's run a Monte Carlo simulation,” she suggested, and supplied the code in [Figure 21-8](#).

```
num_games = 1273
lyndsay_wins = 666
num_trials = 10000
at_least = 0
for t in range(num_trials):
    l_wins, j_wins = 0, 0
    for g in range(num_games):
        if random.random() < 0.5:
            l_wins += 1
        else:
            j_wins += 1
    if l_wins >= lyndsay_wins or j_wins >= lyndsay_wins:
        at_least += 1
print('Probability of result at least this',
      'extreme by accident =', at_least/num_trials)
```

[Figure 21-8](#) Lyndsay's simulation of games

When Lyndsay's code was run it printed,

```
Probability of result at least this extreme by accident =
0.0491
```

prompting her to claim that John's statistical test was completely bogus and that the difference in wins was statistically significant at the 5% level.

“No,” John explained patiently, “it's your simulation that's bogus. It assumed that you were the better player and performed the equivalent of a one-tailed test. The inner loop of your simulation is wrong. You should have performed the equivalent of a two-tailed test by testing whether, in the simulation, either player won more than the 666 games that you won in actual competition.” John then ran the simulation in [Figure 21-9](#).

```

num_games = 1273
lyndsay_wins = 666
num_trials = 10000
at_least = 0
for t in range(num_trials):
    l_wins, j_wins = 0, 0
    for g in range(num_games):
        if random.random() < 0.5:
            l_wins += 1
        else:
            j_wins += 1
    if l_wins >= lyndsay_wins or j_wins >= lyndsay_wins:
        at_least += 1
print('Probability of result at least this',
      'extreme by accident =', at_least/num_trials)

```

[Figure 21-9](#) Correct simulation of games

### John's simulation printed

Probability of result at least this extreme by accident =  
0.0986

“That’s pretty darned close to what my two-tailed test predicted,” crowed John. Lyndsay’s unladylike response was not appropriate for inclusion in a family-oriented book.

**Finger exercise:** An investigative reporter discovered that not only was Lyndsay employing dubious statistical methods, she was applying them to data she had merely made up.<sup>151</sup> In fact, John had defeated Lyndsay 479 times and lost 443 times. At what level is this difference statistically significant?

---

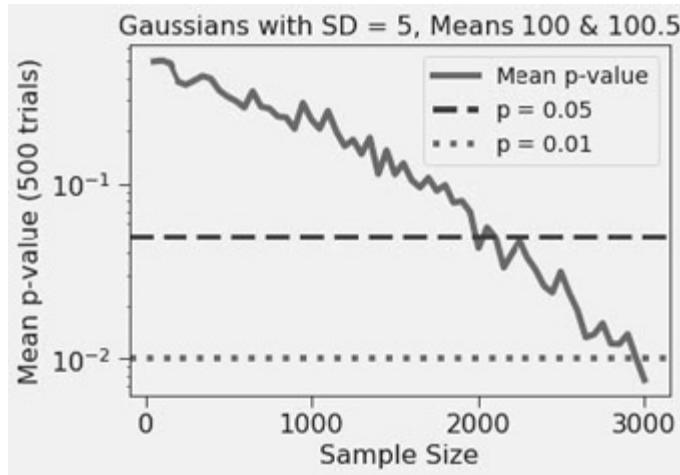
## 21.5 Which N?

A professor wondered whether attending lectures was correlated with grades in his department. He recruited 40 freshmen and gave them all ankle bracelets so that he could track their whereabouts. Half of the students were not allowed to attend any of the lectures in any of their classes,<sup>152</sup> and half were required to attend all of the

lectures.<sup>153</sup> Over the next four years, each student took 40 different classes, yielding 800 grades for each group of students.

When the professor performed a two-tailed t-test on the means of these two samples of size 800, the p-value was about 0.01. This disappointed the professor, who was hoping that there would be no statistically significant effect—so that he would feel less guilty about canceling lectures and going to the beach. In desperation, he took a look at the mean GPAs of the two groups and discovered very little difference. How, he wondered, could such a small difference in means be significant at that level?

When the sample size is large enough, even a small effect can be highly statistically significant.  $n$  matters, a lot. [Figure 21-10](#) plots the mean p-value of 1000 trials against the size of the samples used in those trials. For each sample size and each trial we generated two samples. Each was drawn from a Gaussian with a standard deviation of 5. One had a mean of 100 and the other a mean of 100.5. The mean p-value drops linearly with the sample size. The 0.5% difference in means becomes consistently statistically significant at the 5% level when the sample size reaches about 2000, and at the 1% level when the sample size nears 3000.



[Figure 21-10](#) Impact of sample size on p-value

Returning to our example, was the professor justified in using an  $n$  of 800 for each **arm** of his study? To put it another way, were there

really 800 independent examples for each cohort of 20 students? Probably not. There were 800 grades per sample, but only 20 students, and the 40 grades associated with each student should probably not be viewed as independent examples. After all, some students consistently get good grades, and some students consistently get grades that disappoint.

The professor decided to look at the data a different way. He computed the GPA for each student. When he performed a two-tailed t-test on these two samples, each of size 20, the p-value was about 0.3. He headed to the beach.

---

## 21.6 Multiple Hypotheses

In Chapter 19, we looked at sampling using data from the Boston Marathon. The code in [Figure 21-11](#) reads in data from the 2012 race and looks for statistically significant differences in the mean finishing times of the women from a small set of countries. It uses the `get_BM_data` function defined in Figure 19-2.

```

data = get_BM_data('bm_results2012.csv')
countries_to_compare = ['BEL', 'BRA', 'FRA', 'JPN', 'ITA']

#Build mapping from country to list of female finishing times
country_times = {}
for i in range(len(data['name'])): #for each racer
    if (data['country'][i] in countries_to_compare and
        data['gender'][i] == 'F'):
        try:
            country_times[data['country'][i]].append(data['time'][i])
        except KeyError:
            country_times[data['country'][i]] = [data['time'][i]]

#Compare finishing times of countries
for c1 in countries_to_compare:
    for c2 in countries_to_compare:
        if c1 < c2: # < rather than != so each pair examined once
            pVal = scipy.stats.ttest_ind(country_times[c1],
                                         country_times[c2],
                                         equal_var = False)[1]
        if pVal < 0.05:
            print(c1, 'and', c2,
                  'have significantly different means,',
                  'p-value =', round(pVal, 4))

```

[Figure 21-11](#) Comparing mean finishing times for selected countries

When the code is run, it prints

```
ITA and JPN have significantly different means, p-value =
0.025
```

It looks as if either Italy or Japan can claim to have faster women runners than the other.<sup>154</sup> However, such a conclusion would be pretty tenuous. While one set of runners did have a faster mean time than the other, the sample sizes (20 and 32) were small and perhaps not representative of the capabilities of women marathoners in each country.

More important, there is a flaw in the way we constructed our experiment. We checked 10 null hypotheses (one for each distinct pair of countries) and discovered that one of them could be rejected at the 5% level. One way to think about it is that we were actually checking the null hypothesis: “for all pairs of countries, the mean

finishing times of their female marathon runners are the same.” It might be fine to reject that null hypothesis, but that is not the same as rejecting the null hypothesis that women marathon runners from Italy and Japan are equally fast.

The point is made starkly by the example in [Figure 21-12](#). In that example, we draw 50 pairs of samples of size 200 from the same population, and for each we test whether the means of the samples are statistically different.

```
num_hyps = 50
sample_size = 200
population = []
for i in range(5000): #Create large population
    population.append(random.gauss(0, 1))
sample1s, sample2s = [], []
#Generate many pairs of small samples
for i in range(num_hyps):
    sample1s.append(random.sample(population, sample_size))
    sample2s.append(random.sample(population, sample_size))
#Check pairs for statistically significant difference
numSig = 0
for i in range(num_hyps):
    if scipy.stats.ttest_ind(sample1s[i], sample2s[i])[1] < 0.05:
        numSig += 1
print('# of statistically significantly different (p < 0.05) pairs =',
      numSig)
```

[Figure 21-12](#) Checking multiple hypotheses

Since the samples are all drawn from the same population, we know that the null hypothesis is true. Yet, when we run the code it prints

```
# of statistically significantly different (p < 0.05) pairs
= 2
```

indicating that the null hypothesis can be rejected for two pairs.

This is not particularly surprising. Recall that a p-value of 0.05 indicates that if the null hypothesis holds, the probability of seeing a difference in means at least as large as the difference for the two samples is 0.05. Therefore, it is not surprising that if we examine 50

pairs of samples, two of them have means that are statistically significantly different from each other. Running large sets of related experiments, and then **cherry-picking** the result you like, can be kindly described as sloppy. An unkind person might call it something else.

Returning to our Boston Marathon experiment, we checked whether we could reject the null hypothesis (no difference in means) for 10 pairs of samples. When running an experiment involving multiple hypotheses, the simplest and most conservative approach is to use something called the **Bonferroni correction**. The intuition behind it is simple: when checking a family of  $m$  hypotheses, one way of maintaining an appropriate **family-wise error rate** is to test each individual hypothesis at a level of  $\frac{1}{m} * \alpha$ . Using the Bonferroni correction to see if the difference between Italy and Japan is significant at the  $\alpha = 0.05$  level, we should check if the p-value is less than  $0.05/10$  i.e.,  $0.005$ —which it is not.

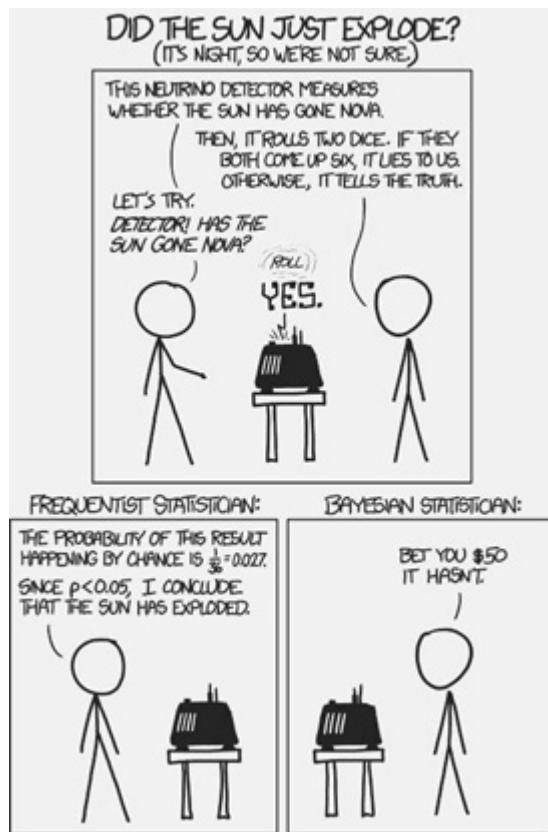
The Bonferroni correction is conservative (i.e., it fails to reject the null hypothesis more often than necessary) if there are many tests or the test statistics for the tests are positively correlated. An additional issue is the absence of a generally accepted definition of “family of hypotheses.” It is obvious that the hypotheses generated by the code in [Figure 21-12](#) are related, and therefore a correction needs to be applied. But the situation is not always so clear cut.

---

## 21.7 Conditional Probability and Bayesian Statistics

Up to this point, we have taken what is called a **frequentist** approach to statistics. We have drawn conclusions from samples based entirely on the frequency or proportion of the data. This is the most commonly used inference framework, and leads to the well-established methodologies of statistical hypothesis testing and confidence intervals covered earlier in this book. In principle, it has the advantage of being unbiased. Conclusions are reached solely on the basis of observed data.

In some situations, however, an alternative approach to statistics, **Bayesian statistics**, is more appropriate. Consider the cartoon in [Figure 21-13](#).<sup>155</sup>



[Figure 21-13](#) Has the sun exploded?

What's going on here? The frequentist knows that there are only two possibilities: the machine rolls a pair of sixes and is lying, or it doesn't roll a pair of sixes and is telling the truth. Since the probability of not rolling a pair of sixes is  $35/36$  (97.22%), the frequentist concludes that the machine is probably telling the truth, and therefore the sun has probably exploded.<sup>[156](#)</sup>

The Bayesian uses additional information in building her probability model. She agrees it is unlikely that the machine rolls a pair of sixes; however, she argues that the probability of that happening needs to be compared to the *a priori* probability that the sun has not exploded. She concludes that the likelihood of the sun having not exploded is even higher than 97.22%, and decides to bet that "the sun will come out tomorrow."

### 21.7.1 Conditional Probabilities

The key idea underlying Bayesian reasoning is **conditional probability**.

In our earlier discussion of probability, we relied on the assumption that events were independent. For example, we assumed that whether a coin flip came up heads or tails was unrelated to whether the previous flip came up heads or tails. This is convenient mathematically, but life doesn't always work that way. In many practical situations, independence is a bad assumption.

Consider the probability that a randomly chosen adult American is male and weighs over 197 pounds. The probability of being male is about 0.5 and the probability of weighing more than 197 pounds (the average weight in the U.S.<sup>157</sup>) is also about 0.5.<sup>158</sup> If these were independent events, the probability of the selected person being both male and weighing more than 197 pounds would be 0.25. However, these events are not independent, since the average American male weighs about 30 pounds more than the average female. So, a better question to ask is 1) what is the probability of the selected person being a male, and 2) given that the selected person is a male, what is the probability of that person weighing more than 197 pounds? The notation of conditional probability makes it easy to say just that.

The notation  $P(A|B)$  stands for the probability of  $A$  being true under the assumption that  $B$  is true. It is often read as “the probability of  $A$ , given  $B$ .” Therefore, the formula

$$P(\text{male}) * P(\text{weight} > 197 | \text{male})$$

expresses exactly the probability we are looking for. If  $P(A)$  and  $P(B)$  are independent,  $P(A|B) = P(A)$ . For the above example,  $B$  is male and  $A$  is weight  $> 197$ .

In general, if  $P(B) \neq 0$ ,

$$P(A|B) = \frac{P(A \text{ and } B)}{P(B)}$$

Like conventional probabilities, conditional probabilities always lie between 0 and 1. Furthermore, if  $\bar{A}$  stands for *not A*,  $P(A|B) + P(\bar{A}|B) = 1$ . People often incorrectly assume that  $P(A|B)$  is equal to  $P(B|A)$ . There is no reason to expect this to be true. For example, the value of  $P(Male|Maltese)$  is roughly 0.5, but  $P(Maltese|Male)$  is about 0.000064.<sup>159</sup>

**Finger exercise:** Estimate the probability that a randomly chosen American is both male and weighs more than 197 pounds. Assume that 50% of the population is male, and that the weights of the male population are normally distributed with a mean of 210 pounds and a standard deviation of 30 pounds. (Hint: think about using the empirical rule.)

The formula  $P(A|B, C)$  stands for the probability of  $A$ , given that both  $B$  and  $C$  hold. Assuming that  $B$  and  $C$  are independent of each other, the definition of a conditional probability and the multiplication rule for independent probabilities imply that

$$P(A|B, C) = \frac{P(A, B, C)}{P(B, C)}$$

where the formula  $P(A, B, C)$  stands for the probability of all of  $A$ ,  $B$ , and  $C$  being true.

Similarly,  $P(A, B|C)$  stands for the probability of  $A$  and  $B$ , given  $C$ . Assuming that  $A$  and  $B$  are independent of each other

$$P(A, B|C) = P(A|C) * P(B|C)$$

### 21.7.2 Bayes' Theorem

Suppose that an asymptomatic woman in her forties goes for a mammogram and receives bad news: the mammogram is “positive.”<sup>160</sup>

The probability that a woman who has breast cancer will get a **true positive** result on a mammogram is 0.9. The probability that a

woman who does not have breast cancer will get a **false positive** on a mammogram is 0.07.

We can use conditional probabilities to express these facts. Let

canc = has breast cancer

TP = true positive

FP = false positive

Using these variables, we write the conditional probabilities

$$P(TP \mid \text{canc}) = 0.9$$

$$P(FP \mid \text{not Canc}) = 0.07$$

Given these conditional probabilities, how worried should a woman in her forties with a positive mammogram be? What is the probability that she actually has breast cancer? Is it 0.93, since the false positive rate is 7%? More? less?

It's a trick question: We haven't supplied enough information to allow you to answer the question in a sensible way. To do that, you need to know the **prior probabilities** for breast cancer for a woman in her forties. The fraction of women in their forties who have breast cancer is 0.008 (8 out of 1000). The fraction who do not have breast cancer is therefore  $1 - 0.008 = 0.992$ :

$$P(\text{canc} \mid \text{woman in her 40s}) = 0.008$$

$$P(\text{not canc} \mid \text{woman in her 40s}) = 0.992$$

We now have all the information we need to address the question of how worried that woman in her forties should be. To compute the probability that she has breast cancer we use something called **Bayes' Theorem**<sup>161</sup> (often called Bayes' Law or Bayes' Rule) :

$$P(A|B) = \frac{P(A) * P(B|A)}{P(B)}$$

In the Bayesian world, probability measures a **degree of belief**. Bayes' theorem links the degree of belief in a proposition before and after accounting for evidence. The formula to the left of the equal sign,  $P(A|B)$ , is the **posterior** probability, the degree of belief in  $A$ , having accounted for  $B$ . The posterior is defined in terms of the

**prior**,  $P(A)$ , and the **support** that the evidence,  $B$ , provides for  $A$ . The support is the ratio of the probability of  $B$  holding if  $A$  holds and the probability of  $B$  holding independently of  $A$ , i.e.,  $\frac{P(B|A)}{P(B)}$ .

If we use Bayes' Theorem to estimate the probability of the woman actually having breast cancer, we get (where  $canc$  plays the role of  $A$ , and  $pos$  the role of  $B$  in our statement of Bayes' Theorem)

$$P(canc|pos) = \frac{P(canc) * P(pos|canc)}{P(pos)}$$

The probability of having a positive test is

$$P(pos) = P(pos|canc) * P(canc) + P(pos|not\ canc) * (1 - P(canc))$$

so

$$P(canc|pos) = \frac{0.008 * 0.9}{0.9 * 0.008 + 0.07 * 0.992} = \frac{0.0072}{0.07664} \approx 0.094$$

That is, approximately 90% of the positive mammograms are false positives!<sup>162</sup> Bayes' Theorem helped us here because we had an accurate estimate of the prior probability of a woman in her forties having breast cancer.

Keep in mind that if we had started with an incorrect prior, incorporating that prior into our probability estimate would make the estimate worse rather than better. For example, if we had started with the prior

$$P(canc | \text{ women in her 40's}) = 0.6$$

we would have concluded that the false positive rate was about 5%, i.e., that the probability of a woman in her forties with a positive mammogram having breast cancer is roughly 0.95.

**Finger exercise:** You are wandering through a forest and see a field of delicious-looking mushrooms. You fill your basket with them, and head home prepared to cook them up and serve them to your husband. Before you cook them, however, he demands that you consult a book about local mushroom species to check whether they are poisonous. The book says that 80% of the mushrooms in the local forest are poisonous. However, you compare your mushrooms to the ones pictured in the book, and decide that you are 95% certain that your mushrooms are safe. How comfortable should you be about serving them to your husband (assuming that you would rather not become a widow)?

---

## 21.8 Terms Introduced in Chapter

randomized trial  
treatment group  
control group  
statistical significance  
hypothesis testing  
null hypothesis  
alternative hypothesis  
test statistic  
hypothesis rejection  
type I error  
type II error  
t-statistic  
t-distribution  
degrees of freedom  
p-value  
scientific method

t-test  
power of a study  
two-tailed p-test  
one-tailed p-test  
arm of a study  
cherry-picking  
Bonferroni correction  
family-wise error rate  
frequentist statistics  
Bayesian statistics  
conditional probability  
true positive  
false positive  
prior probability  
Bayes theorem  
degree of belief  
posterior probability  
prior  
support

---

**145** This is a grayscale version of a color image provided by the U.S. National Oceanic and Atmospheric Administration.

**146** In his formulation, Fisher had only a null hypothesis. The idea of an alternative hypothesis was introduced later by Jerzy Neyman and Egon Pearson.

**147** Many researchers, including the author of this book, believe strongly that the “rejectionist” approach to reporting statistics is

unfortunate. It is almost always preferable to report the actual significance level rather than merely stating that “the null hypothesis has been rejected at the 5% level.”

**148** Guinness forbade Gosset from publishing under his own name. Gosset used the pseudonym “Student” when he published his seminal 1908 paper, “Probable Error of a Mean,” about t-distributions. As a result, the distribution is frequently called “Student’s t-distribution.”

**149.** The “beyond a reasonable doubt” standard implies society believes that in the case of a criminal trial, type I errors (convicting an innocent person) are much less desirable than type II errors (acquitting a guilty person). In civil cases, the standard is “the preponderance of the evidence,” suggesting that society believes that the two kinds of errors are equally undesirable.

**150** Katherine S. Button, John P. A. Ioannidis, Claire Mokrysz, Brian A. Nosek, Jonathan Flint, Emma S. J. Robinson, and Marcus R. Munafò (2013) "Power failure: why small sample size undermines the reliability of neuroscience," *Nature Reviews Neuroscience*, 14: 365-376.

**151** In Lyndsay's defense, the use of “alternative facts” seems to be official policy in some places.

**152** They should have been given a tuition rebate, but weren't.

**153** They should have been given combat pay, but weren't.

**154.** We could easily find out which by looking at the sign of the t-statistic, but in the interest of not offending potential purchasers of this book, we won't.

**155** [http://imgs.xkcd.com/comics/frequentists\\_vs\\_bayesians.png](http://imgs.xkcd.com/comics/frequentists_vs_bayesians.png)

**156** If you are of the frequentist persuasion, keep in mind that this cartoon is a parody—not a serious critique of your religious beliefs.

**157.** This number may strike you as high. It is. The average American adult weighs about 40 pounds more than the average

adult in Japan. The only three countries on Earth with higher average adult weights than the U.S. are Nauru, Tonga, and Micronesia.

158 The probability of weighing more than the *median* weight is 0.5, but that doesn't imply that the probability of weighing more than the *mean* is 0.5. However, for the purposes of this discussion, let's pretend that it does.

159. By "Maltese" we mean somebody from the country of Malta. We have no idea what fraction of the world's males are cute little dogs.

160 In medical jargon, a "positive" test is usually bad news. It implies that a marker of disease has been found.

161 Bayes' theorem is named after Rev. Thomas Bayes (1701–1761), and was first published two years after his death. It was popularized by Laplace, who published the modern formulation of the theorem in 1812 in his *Théorie analytique des probabilités*.

162 This is one of the reasons that there is some controversy in the medical community about the value of mammography as a routine screening tool for some cohorts.

## 22

# LIES, DAMNED LIES, AND STATISTICS

*“If you can't prove what you want to prove, demonstrate something else and pretend they are the same thing. In the daze that follows the collision of statistics with the human mind, hardly anyone will notice the difference.”<sup>163</sup>*

Anyone can lie by simply making up fake statistics. Telling fibs with accurate statistics is more challenging, but still not difficult.



Statistical thinking is a relatively new invention. For most of recorded history, things were assessed qualitatively rather than quantitatively. People must have had an intuitive sense of some statistical facts (e.g., that women are usually shorter than men), but they had no mathematical tools that would allow them to proceed from anecdotal evidence to statistical conclusions. This started to change in the middle of the seventeenth century, most notably with the publication of John Graunt's *Natural and Political Observations Made Upon the Bills of Mortality*. This pioneering work used

statistical analysis to estimate the population of London from death rolls and attempted to provide a model that could be used to predict the spread of plague.

Alas, since that time people have used statistics as much to mislead as to inform. Some have willfully used statistics to mislead; others have merely been incompetent. In this chapter we discuss some of the ways people can be led into drawing inappropriate inferences from statistical data. We trust that you will use this information only for good—to become a better consumer and a more honest purveyor of statistical information.

---

## 22.1 Garbage In Garbage Out (GIGO)

*“On two occasions I have been asked [by members of Parliament], ‘Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?’ I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question.”* — Charles Babbage<sup>[164](#)</sup>.

The message here is a simple one. If the input data is seriously flawed, no amount of statistical massaging will produce a meaningful result.

The 1840 United States census showed that insanity among free blacks and mulattoes was roughly ten times more common than among enslaved blacks and mulattoes. The conclusion was obvious. As U.S. Senator (and former Vice President and future Secretary of State) John C. Calhoun put it, “The data on insanity revealed in this census is unimpeachable. From it our nation must conclude that the abolition of slavery would be to the African a curse.” Never mind that it was soon clear that the census was riddled with errors. As Calhoun reportedly explained to John Quincy Adams, “There were so many errors they balanced one another, and led to the same conclusion as if they were all correct.”

Calhoun's (perhaps willfully) spurious response to Adams was based on a classical error, the **assumption of independence**. Were he more sophisticated mathematically, he might have said something like, “I believe that the measurement errors are unbiased

and independent of each other, and therefore evenly distributed on either side of the mean.” In fact, later analysis showed that the errors were so heavily biased that no statistically valid conclusions could be drawn.<sup>165</sup>

GIGO is a particularly pernicious problem in the scientific literature—because it can be hard to detect. In May 2020, one of the world's most prestigious medical journals (*Lancet*) published a paper about the then raging Covid-19 pandemic. The paper relied on data about 96,000 patients collected from nearly 700 hospitals on six continents. During the review process, the reviewers checked the soundness of the analyses reported in the paper, but not the soundness of the data on which the analyses were based. Less than a month after publication, the paper was retracted based upon the discovery that the data on which it was based were flawed.

---

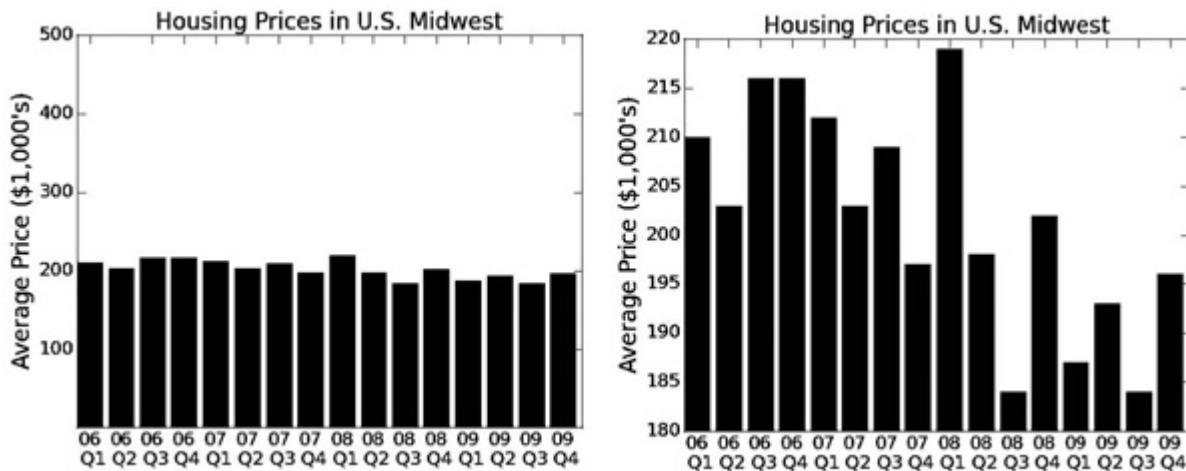
## 22.2 Tests Are Imperfect

Every experiment should be viewed as a potentially flawed test. We can perform a test for a chemical, a phenomenon, a disease, etc. However, the event for which we are testing is not necessarily the same as the result of the test. Professors design exams with the goal of understanding how well a student has mastered some subject matter, but the result of the exam should not be confused with how much a student actually understands. Every test has some inherent error rate. Imagine that a student learning a second language has been asked to learn the meaning of 100 words, but has learned the meaning of only 80 of them. His rate of understanding is 80%, but the probability that he will score 80% on a test with 20 words is certainly not 1.

Tests can have both false negatives and false positives. As we saw in Section 21.7, a negative mammogram does not guarantee absence of breast cancer, and a positive mammogram doesn't guarantee its presence. Furthermore, the test probability and the event probability are not the same thing. This is especially relevant when testing for a rare event, e.g., the presence of a rare disease. If the cost of a false negative is high (e.g., missing the presence of a serious but curable disease), the test should be designed to be highly sensitive, even at the cost of many false positives.

## 22.3 Pictures Can Be Deceiving

There can be no doubt about the utility of graphics for quickly conveying information. However, when used carelessly (or maliciously), a plot can be highly misleading. Consider, for example, the charts in [Figure 22-1](#) depicting housing prices in the U.S. midwestern states.



[Figure 22-1](#) Housing prices in the U.S. Midwest

Looking at the chart on the left of [Figure 22-1](#), it seems as if housing prices were pretty stable during the period 2006-2009. But wait a minute! Wasn't there a collapse of U.S. residential real estate followed by a global financial crisis in late 2008? There was indeed, as shown in the chart on the right.

These two charts show exactly the same data, but convey very different impressions. The chart on the left was designed to give the impression that housing prices had been stable. On the y-axis, the designer used a scale ranging from the absurdly low average price for a house of \$1,000 to the improbably high average price of \$500,000. This minimized the amount of space devoted to the area where prices are changing, giving the impression that the changes were relatively small. The chart on the right was designed to give the impression that housing prices moved erratically, and then crashed. The

designer used a narrow range of prices, so the sizes of the changes were exaggerated.

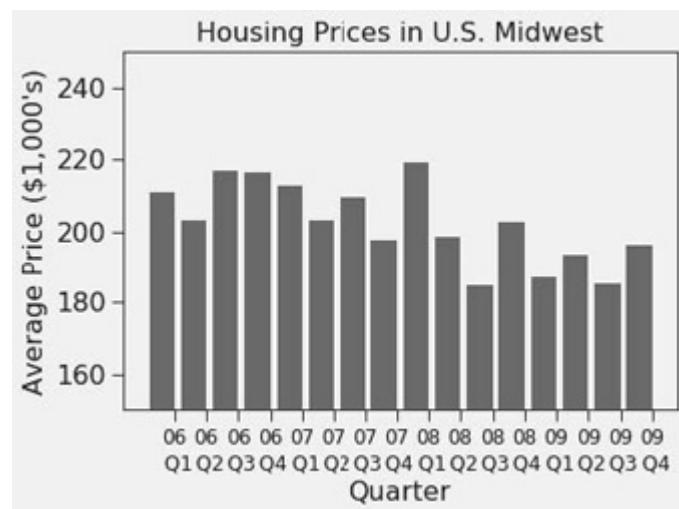
The code in [Figure 22-2](#) produces the two plots we looked at above and a plot intended to give an accurate impression of the movement of housing prices. It uses two plotting facilities that we have not yet seen.

```
def plot_housing(impression):
    """Assumes impression a str. Must be one of 'flat',
    'volatile,' and 'fair'
    Produce bar chart of housing prices over time"""
    labels, prices = ([], [])
    with open('midWestHousingPrices.csv', 'r') as f:
        #Each line of file contains year quarter price
        #for Midwest region of U.S.
        for line in f:
            year, quarter, price = line.split(',')
            label = year[2:4] + '\n Q' + quarter[1]
            labels.append(label)
            prices.append(int(price)/1000)
    quarters = np.arange(len(labels)) #x coords of bars
    width = 0.8 #Width of bars
    plt.bar(quarters, prices, width)
    plt.xticks(quarters+width/2, labels)
    plt.title('Housing Prices in U.S. Midwest')
    plt.xlabel('Quarter')
    plt.ylabel('Average Price ($1,000\'s)')
    if impression == 'flat':
        plt.ylim(1, 500)
    elif impression == 'volatile':
        plt.ylim(180, 220)
    elif impression == 'fair':
        plt.ylim(150, 250)
    else:
        raise ValueError

plot_housing('flat')
plt.figure()
plot_housing('volatile')
plt.figure()
plot_housing('fair')
```

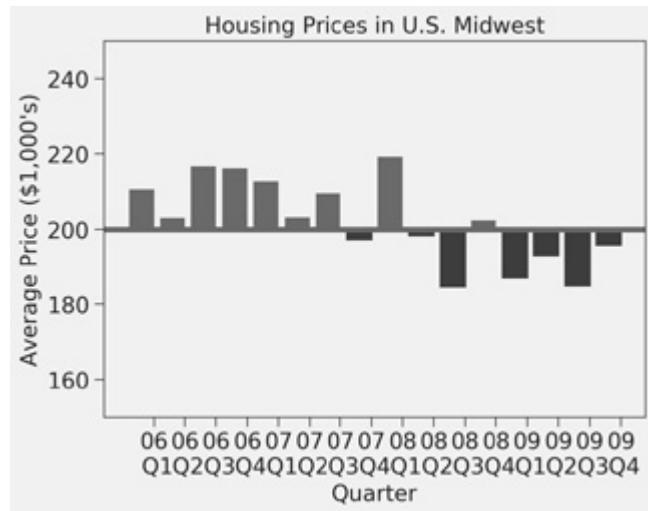
[Figure 22-2](#) Plotting housing prices

The call `plt.bar(quarters, prices, width)` produces a **bar chart** with bars of the given width. The left edges of the bars are the values of the elements of the list `quarters`, and the heights of the bars are the values of the corresponding elements of the list `prices`. The function call `plt.xticks(quarters+width/2, labels)` describes the labels to be associated with the bars. The first argument specifies the placement of each label and the second argument the text of the labels. The function `yticks` behaves analogously. The call `plot_housing('fair')` produces the plot in [Figure 22-3](#).



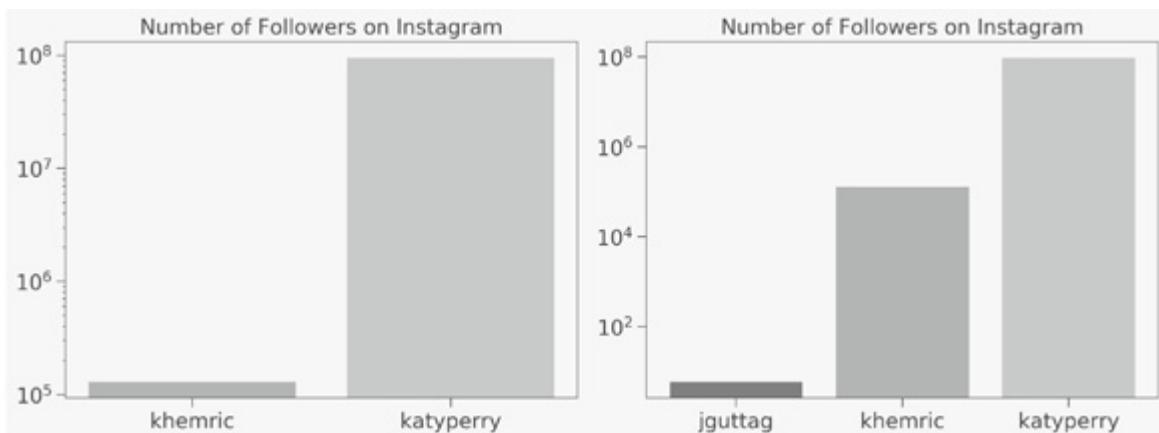
[Figure 22-3](#) A different view of housing prices

**Finger exercise:** It is sometimes illuminating to plot things relative to a baseline, as seen in [Figure 22-4](#). Modify `plot_housing` to produce such plots. The bars below the baseline should be in red. Hint: use the `bottom` keyword argument to `plt.bar`.



[Figure 22-4](#) Housing prices relative to \$200,000

A logarithmic y-axis provides a wonderful tool for making deceptive plots. Consider the bar graphs in [Figure 22-5](#). The plot on the left provides a more accurate impression of the difference in the number of people following khemric and katyperry. The presence of the sparsely followed jguttag in the plot on the right forces the y-axis to devote a larger proportion of its length to smaller values, thus leaving less distance to distinguish between the number of followers of khemric and katyperry.[166](#)



[Figure 22-5](#) Comparing number of Instagram followers

---

## 22.4 Cum Hoc Ergo Propter Hoc<sup>167</sup>

It has been shown that college students who regularly attend class have higher average grades than students who attend class only sporadically. Those of us who teach these classes would like to believe that this is because the students learn something from the classes we teach. Of course, it is at least equally likely that those students get better grades because students who are more likely to attend classes are also more likely to study hard.

**Correlation** is a measure of the degree to which two variables move in the same direction. If  $x$  moves in the same direction as  $y$ , the variables are positively correlated. If they move in opposite directions, they are negatively correlated. If there is no relationship, the correlation is 0. People's heights are positively correlated with the heights of their parents. The correlation between smoking and life span is negative.

When two things are correlated, there is a temptation to assume that one has caused the other. Consider the incidence of flu in North America. The number of cases rises and falls in a predictable pattern. There are almost no cases in the summer; the number of cases starts to rise in the early fall and then starts dropping as summer approaches. Now consider the number of children attending school. There are very few children in school in the summer; enrollment starts to rise in the early fall and then drops as summer approaches.

The correlation between the opening of schools and the rise in the incidence of flu is inarguable. This has led some to conclude that going to school is an important causative factor in the spread of flu. That might be true, but we cannot conclude it based simply on the correlation. Correlation does not imply causation! After all, the correlation could be used just as easily to justify the belief that flu outbreaks cause schools to be in session. Or perhaps there is no causal relationship in either direction, and some **lurking variable** we have not considered causes each. In fact, as it happens, the flu virus survives considerably longer in cool dry air than it does in warm wet air, and in North America both the flu season and school sessions are correlated with cooler and dryer weather.

Given enough retrospective data, it is always possible to find two variables that are correlated, as illustrated by the chart in [Figure 22-](#)

## 6.<sup>168</sup>

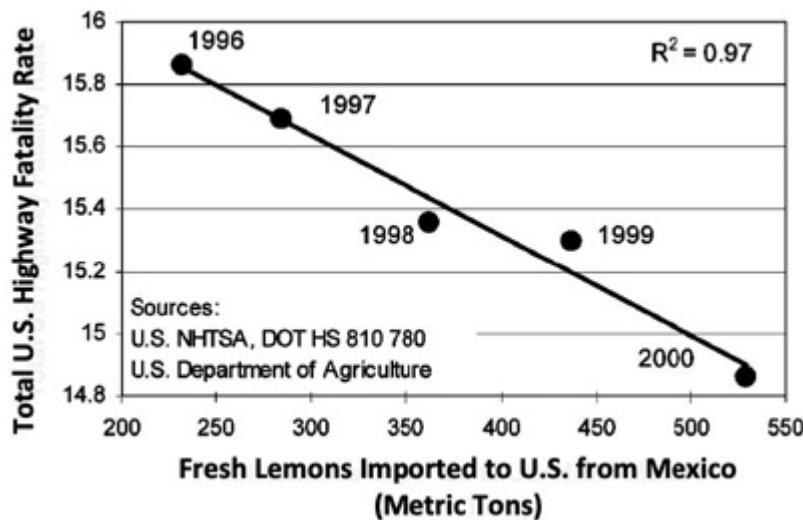


Figure 22-6 Do Mexican lemons save lives?

When such correlations are found, the first thing to do is to ask whether there is a plausible theory explaining the correlation.

Falling prey to the *cum hoc ergo propter hoc* fallacy can be quite dangerous. At the start of 2002, roughly six million American women were being prescribed hormone replacement therapy (HRT) in the belief that it would substantially lower their risk of cardiovascular disease. That belief was supported by several highly reputable published studies that demonstrated a reduced incidence of cardiovascular death among women using HRT.

Many women, and their physicians, were taken by surprise when the *Journal of the American Medical Society* published an article asserting that HRT in fact increased the risk of cardiovascular disease.<sup>169</sup> How could this have happened?

Reanalysis of some of the earlier studies showed that women undertaking HRT were likely to be from groups with better than average diet and exercise regimes. Perhaps the women undertaking HRT were on average more health conscious than the other women in the study, so that taking HRT and improved cardiac health were coincident effects of a common cause.

**Finger exercise:** Over the last 100 years, the number of deaths per year in Canada was positively correlated with the amount of meat consumed per year in Canada. What lurking variable might explain this?

---

## 22.5 Statistical Measures Don't Tell the Whole Story

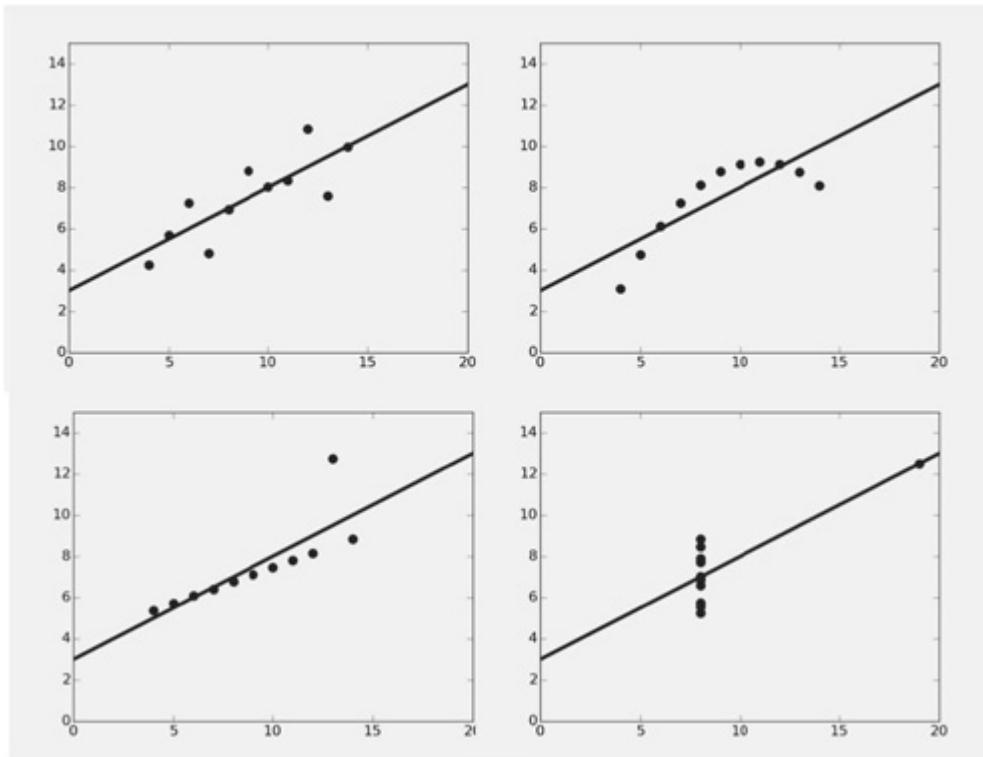
An enormous number of different statistics can be extracted from a data set. By carefully choosing among these, it is possible to convey differing impressions about the same data. A good antidote is to look at the data set itself.

In 1973, the statistician F.J. Anscombe published a paper with the table in [Figure 22-7](#), often called Anscombe's quartet. It contains the  $\langle x, y \rangle$  coordinates of points from each of four data sets. Each of the four data sets has the same mean value for  $x$  (9.0), the same mean value for  $y$  (7.5), the same variance for  $x$  (10.0), the same variance for  $y$  (3.75), and the same correlation between  $x$  and  $y$  (0.816). And if we use linear regression to fit a line to each, we get the same result for each,  $y = 0.5x + 3$ .

<b>x</b>	<b>y</b>	<b>x</b>	<b>y</b>	<b>x</b>	<b>y</b>	<b>x</b>	<b>y</b>
10.0	8.04	10.0	9.14	10.0	7.46	8.0	6.58
8.0	6.95	8.0	8.14	8.0	6.77	8.0	5.76
13.0	7.58	13.0	8.74	13.0	12.74	8.0	7.71
9.0	8.81	9.0	8.77	9.0	7.11	8.0	8.84
11.0	8.33	11.0	9.26	11.0	7.81	8.0	8.47
14.0	9.96	14.0	8.10	14.0	8.84	8.0	7.04
6.0	7.24	6.0	6.13	6.0	6.08	8.0	5.25
4.0	4.26	4.0	3.10	4.0	5.39	19.0	12.50
12.0	10.84	12.0	9.13	12.0	8.15	8.0	5.56
7.0	4.82	7.0	7.26	7.0	6.42	8.0	7.91
5.0	5.68	5.0	4.74	5.0	5.73	8.0	6.89

[Figure 22-7](#) Statistics for Anscombe's quartet

Does this mean that there is no obvious way to distinguish these data sets from each other? No. We simply need to plot the data to see that the data sets are not alike ([Figure 22-8](#)).



[Figure 22-8](#) Data for Anscombe's quartet

The moral is simple: if possible, always take a look at some representation of the raw data.

---

## 22.6 Sampling Bias

During World War II, whenever an Allied plane returned from a mission over Europe, the plane was inspected to see where the flak from antiaircraft artillery had impacted. Based upon this data, mechanics reinforced those areas of the planes that seemed most likely to be hit by flak.

What's wrong with this? They did not inspect the planes that failed to return from missions because they had been downed by flak. Perhaps these unexamined planes failed to return precisely because they were hit in the places where the flak would do the most damage. This particular error is called **non-response bias**. It is quite common in surveys. At many universities, for example, students are asked during one of the lectures late in the term to fill out a form

rating the quality of the professor's lectures. Though the results of such surveys are often unflattering, they could be worse. Those students who think that the lectures are so bad that they aren't worth attending are not included in the survey.<sup>[170](#)</sup>

As discussed in Chapter 19, all statistical techniques are based upon the assumption that by sampling a subset of a population we can infer things about the population as a whole. If random sampling is used, we can make precise mathematical statements about the expected relationship of the sample to the entire population. Unfortunately, many studies, particularly in the social sciences, are based on what is called **convenience** (or **accidental**) **sampling**. This involves choosing samples based on how easy they are to procure. Why do so many psychological studies use populations of undergraduates? Because they are easy to find on college campuses. A convenience sample *might* be representative, but there is no way of knowing whether it actually *is* representative.

**Finger exercise:** The **infection-fatality rate** for a disease is the number of people who contract the disease divided by the number of those people who die from the disease. The **case-fatality rate** for a disease is the number of people who are diagnosed with the disease divided by the number of those people who die from the disease. Which of these is easier to estimate accurately, and why?

---

## 22.7 Context Matters

It is easy to read more into the data than it actually implies, especially when viewing the data out of context. On April 29, 2009, CNN reported that, “Mexican health officials suspect that the swine flu outbreak has caused more than 159 deaths and roughly 2,500 illnesses.” Pretty scary stuff—until we compare it to the approximately 36,000 deaths attributable annually to the seasonal flu in the U.S.

An often quoted, and accurate, statistic is that most auto accidents happen within 10 miles of home. So what? Most driving is done within 10 miles of home! Besides, what does “home” mean in this context? The statistic is computed using the address at which the automobile is registered as “home.” Might you reduce the probability

of getting into an accident by merely registering your car in some distant place?

Opponents of government initiatives to reduce the prevalence of guns in the United States are fond of quoting the statistic that roughly 99.8% of the firearms in the U.S. will not be used to commit a violent crime in any given year. But without some context, it's hard to know what that implies. Does it imply that there is not much gun violence in the U.S.? The National Rifle Association reports that there are roughly 300 million privately owned firearms in the U.S.—0.2% of 300 million is 600,000!

---

## 22.8 Comparing Apples to Oranges

Take a quick look at the image in [Figure 22-9](#).



[Figure 22-9](#) Welfare vs. full-time jobs

What impression does it leave you with? Are many more Americans on welfare than working?

The bar on the left is about 500% taller than the bar on the right. However, the numbers on the bars tell us that the y-axis has been truncated. If it had not been, the bar on the left would have been only 6.8% higher. Still, it is kind of shocking to think that 6.8% more people are on welfare than working. Shocking, and misleading.

The “people on welfare” number is derived from the U.S. Census Bureau's tally of people participating in means-tested programs. This

tally includes anyone residing in a household where at least one person received any benefit. Consider, for example, a household containing two parents and three children in which one parent has a full-time job and the other a part-time job. If that household received food stamps, the household would add five people to the tally of “people on welfare” and one to the tally of full-time jobs.

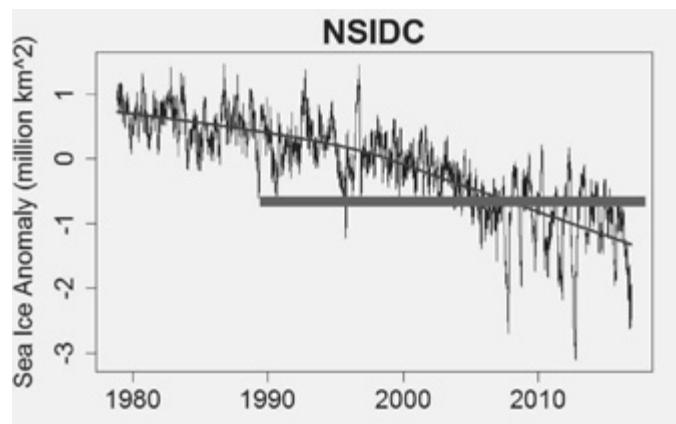
Both numbers are “correct,” but they are not comparable. It’s like concluding that Olga is a better farmer than Mark because she grows 20 tons of potatoes per acre whereas Mark grows only 3 tons of blueberries per acre.

---

## 22.9 Picking Cherries

While we are on the subject of fruit, picking cherries is just as bad as comparing apples and oranges. **Cherry picking** involves choosing specific pieces of data, and ignoring others, for the purpose of supporting some position.

Consider the plot in [Figure 22-10](#). The trend is pretty clear, but if we wish to argue that the planet is not warming using this data, we can cite the fact that there was more ice in April 2013 than in April of 1988, and ignore the rest of the data.



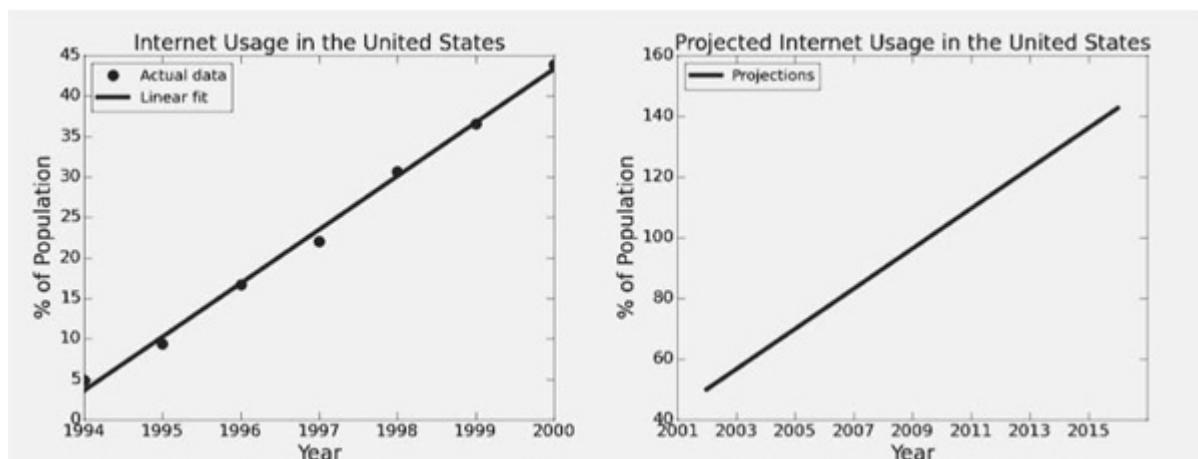
[Figure 22-10](#) Sea ice in the Arctic

---

## 22.10 Beware of Extrapolation

It is all too easy to extrapolate from data. We did that in Section 20.1.1 when we extended fits derived from linear regression beyond the data used in the regression. Extrapolation should be done only when you have a sound theoretical justification for doing so. Be especially wary of straight-line extrapolations.

Consider the plot on the left in [Figure 22-11](#). It shows the growth of Internet usage in the United States from 1994 to 2000. As you can see, a straight line provides a pretty good fit.



[Figure 22-11](#) Growth of Internet usage in U.S.

The plot on the right of [Figure 22-11](#) uses this fit to project the percentage of the U.S. population using the Internet in following years. The projection is hard to believe. It seems unlikely that by 2009 everybody in the U.S. was using the Internet, and even less likely that by 2015 more than 140% of the U.S. population was using the Internet.

---

## 22.11 The Texas Sharpshooter Fallacy

Imagine that you are driving down a country road in Texas. You see a barn that has six targets painted on it, and a bullet hole at the very center of each target. “Yes sir,” says the owner of the barn, “I never

miss.” “That's right,” says his spouse, “there ain't a man in the state of Texas who's more accurate with a paint brush.” Got it? He fired the six shots, and then painted the targets around them.

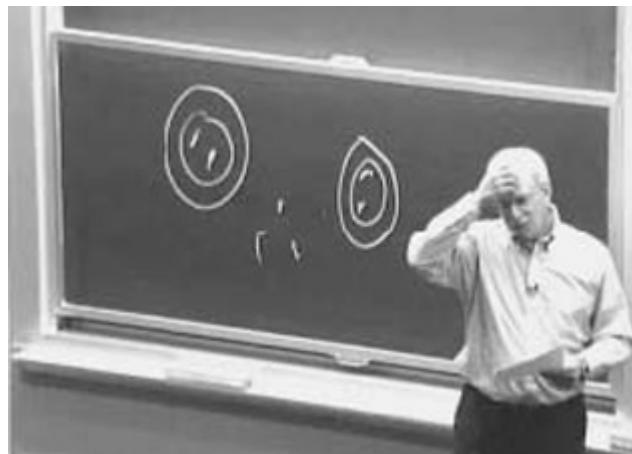


Figure 22-12 Professor puzzles over students' chalk-throwing accuracy

A classic of the genre appeared in 2001.<sup>[171](#)</sup> It reported that a research team at the Royal Cornhill Hospital in Aberdeen had discovered that “anorexic women are most likely to have been born in the spring or early summer... Between March and June there were 13% more anorexics born than average, and 30% more in June itself.”

Let's look at that worrisome statistic for those women born in June. The team studied 446 women who had been diagnosed as anorexic, so the mean number of births per month was slightly more than 37. This suggests that the number born in June was 48 ( $37 * 1.3$ ). Let's write a short program ([Figure 22-13](#)) to estimate the probability that this occurred purely by chance.

```
def june_prob(num_trials):
    june_48 = 0
    for trial in range(num_trials):
        june = 0
        for i in range(446):
            if random.randint(1,12) == 6:
                june += 1
        if june >= 48:
            june_48 += 1
    print('Probability of at least 48 births in June =',
          round(june_48/num_trials, 4))
```

[Figure 22-13](#) Probability of 48 anorexics being born in June

When we ran `june_prob(10000)` it printed

Probability of at least 48 births in June = 0.0427

It looks as if the probability of at least 48 babies being born in June purely by chance is around 4.25%. So perhaps those researchers in Aberdeen are on to something. Well, they might have been on to something had they started with the hypothesis that more babies who will become anorexic are born in June, and then run a study designed to check that hypothesis.

But that is not what they did. Instead, they looked at the data and then, imitating the Texas sharpshooter, drew a circle around June. The right statistical question to have asked is what is the probability that in at least one month (out of 12) at least 48 babies were born. The program in [Figure 22-14](#) answers that question.

```

def any_prob(num_trials):
    any_month_48 = 0
    for trial in range(num_trials):
        months = [0]*12
        for i in range(446):
            months[random.randint(0,11)] += 1
        if max(months) >= 48:
            any_month_48 += 1
    print('Probability of at least 48 births in some month =',
          round(any_month_48/num_trials, 4))

```

[Figure 22-14](#) Probability of 48 anorexics being born in some month

The call `any_prob(10000)` printed

Probability of at least 48 births in some month = 0.4357

It appears that it is not so unlikely after all that the results reported in the study reflect a chance occurrence rather a real association between birth month and anorexia. One doesn't have to come from Texas to fall victim to the Texas Sharpshooter Fallacy.

The statistical significance of a result depends upon the way the experiment was conducted. If the Aberdeen group had started out with the hypothesis that more anorexics are born in June, their result would be worth considering. But if they started with the hypothesis that there exists a month in which an unusually large proportion of anorexics are born, their result is not very compelling. In effect, they were testing multiple hypotheses and cherry-picking a result. They probably should have applied a Bonferroni correction (see Section 21.6).

What next steps might the Aberdeen group have taken to test their newfound hypothesis? One possibility is to conduct a **prospective study**. In a prospective study, one starts with a set of hypotheses, recruits subjects before they have developed the outcome of interest (anorexia in this case), and then follows the subjects for a period of time. If the group had conducted a prospective study with a specific hypothesis and gotten similar results, we might be convinced.

Prospective studies can be expensive and time-consuming to perform. In a **retrospective study**, existing data must be analyzed

in ways that reduce the likelihood of getting misleading results. One common technique, as discussed in Section 20.4, is to split the data into a training set and a held out test set. For example, they could have chosen  $446/2$  women at random from their data (the training set) and tallied the number of births for each month. They could have then compared that to the number of births each month for the remaining women (the holdout set).

---

## 22.12 Percentages Can Confuse

An investment advisor called a client to report that the value of his stock portfolio had risen  $16\%$  over the last month. The advisor admitted that there had been some ups and downs over the year but was pleased to report that the average monthly change was  $+0.5\%$ . Imagine the client's surprise when he got his statement for the year and observed that the value of his portfolio had declined over the year.

He called his advisor and accused him of being a liar. "It looks to me," he said, "like my portfolio declined by about  $8\%$ , and you told me that it went up by  $0.5\%$  a month." "I did not," the financial advisor replied. "I told you that the average monthly change was  $+0.5\%$ ." When he examined his monthly statements, the investor realized that he had not been lied to, just misled. His portfolio went down by  $15\%$  in each month during the first half of the year, and then went up by  $16\%$  in each month during the second half of the year.

When thinking about percentages, we always need to pay attention to the basis on which the percentage is computed. In this case, the  $15\%$  declines were on a higher average basis than the  $16\%$  increases.

Percentages can be particularly misleading when applied to a small basis. You might read about a drug that has a side effect of increasing the incidence of some illness by  $200\%$ . But if the base incidence of the disease is very low, say one in  $1,000,000$ , you might decide that the risk of taking the drug is more than counterbalanced by the drug's positive effects.

**Finger exercise:** On May 19, 2020, the *New York Times* reported a  $123\%$  increase in U.S. air travel in a single month (from 95,161

passengers to 212,508 passengers). It also reported that this increase followed a recent 96% drop in air travel. What was the total net percentage change?

---

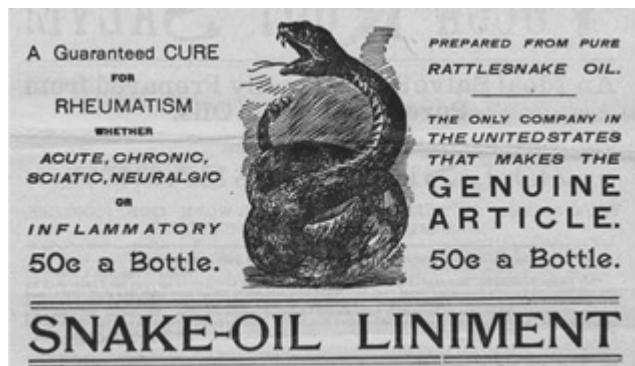
## 22.13 The Regressive Fallacy

The **regressive fallacy** occurs when people fail to account for the natural fluctuations of events.

All athletes have good days and bad days. When they have good days, they try not to change anything. When they have a series of unusually bad days, however, they often try to make changes. Even if the changes are not actually helpful, regression to the mean (Section 17.3) makes it likely that over the next few days the athlete's performance will be better than the unusually poor performances preceding the changes. This may mislead the athlete into assuming that there is a **treatment effect**, i.e., attributing the improved performance to the changes he or she made.

The Nobel prize-winning psychologist Daniel Kahneman tells a story about an Israeli Air Force flight instructor who rejected Kahneman's assertion that "rewards for improved performance work better than punishment for mistakes." The instructor's argument was "On many occasions I have praised flights cadets for clean execution of some aerobatic maneuver. The next time they try the same maneuver they usually do worse. On the other hand, I have often screamed into a cadet's earphone for bad execution, and in general he does better on the next try."<sup>172</sup> It is natural for humans to imagine a treatment effect, because we like to think causally. But sometimes it is simply a matter of luck.

Imagining a treatment effect when there is none can be dangerous. It can lead to the belief that vaccinations are harmful, that snake oil cures all aches and pains, or that investing exclusively in funds that "beat the market" last year is a good strategy.



---

## 22.14 Statistically Significant Differences Can Be Insignificant

An admissions officer at the Maui Institute of Technology (MIT), wishing to convince the world that MIT's admissions process is "gender-blind," trumpeted, "At MIT, there is no significant difference between the grade point averages of men and women." The same day, an ardent female chauvinist proclaimed that "At MIT, the women have a significantly higher grade point average than the men." A puzzled reporter at the student newspaper decided to examine the data and expose the liar. But when she finally managed to pry the data out of the university, she concluded that both were telling the truth.

What does the sentence "At MIT, the women have a significantly higher grade point average than the men," actually mean? People who have not studied statistics (most of the population) would probably conclude that there is a "meaningful" difference between the GPAs of women and men attending MIT. In contrast, those who have recently studied statistics might conclude only that 1) the average GPA of women is higher than that of men, and 2) the null hypothesis that the difference in GPA can be attributed to randomness can be rejected at the 5% level.

Suppose, for example, that 2,500 women and 2,500 men were studying at MIT. Suppose further that the mean GPA of men was 3.5, the mean GPA of women was 3.51, and the standard deviation of the GPA for both men and women was 0.25. Most sensible people would consider the difference in GPAs "insignificant." However, from a

statistical point of view the difference is “significant” at close to the 2% level. What is the root of this strange dichotomy? As we showed in Section 21.5, when a study has enough power—i.e., enough examples—even insignificant differences can be statistically significant.

A related problem arises when a study is very small. Suppose you flipped a coin twice and it came up heads both times. Now, let's use the two-tailed one-sample t-test we saw in Section 21.3 to test the null hypothesis that the coin is fair. If we assume that the value of heads is 1 and the value of tails is 0, we can get the p-value using the code

```
scipy.stats.ttest_1samp([1, 1], 0.5)[1]
```

It returns a p-value of 0, indicating that if the coin is fair, the probability of getting two consecutive heads is nil. We would have gotten a different answer if we had taken a Bayesian approach starting with the prior that the coin is fair.

---

## 22.15 Just Beware

It would be easy, and fun, to fill a few hundred pages with a history of statistical abuses. But by now you probably got the message: It's just as easy to lie with numbers as it is to lie with words. Make sure that you understand what is actually being measured and how those “statistically significant” results were computed before you jump to conclusions. As the Nobel Prize winning economist Ronald Coase said, “If you torture the data long enough, it will confess to anything.”

---

## 22.16 Terms Introduced in Chapter

GIGO

assumption of independence

bar chart

correlation

causation  
lurking variable  
non-response bias  
convenience (accidental) sampling  
infection-fatality rate  
case-fatality rate  
cherry picking  
prospective study  
retrospective study  
regressive fallacy  
treatment effect

---

163 Darrell Huff, *How to Lie with Statistics*, 1954.

164 Charles Babbage, 1791-1871, was an English mathematician and mechanical engineer who is credited with having designed the first programmable computer. He never succeeded in building a working machine, but in 1991 a working mechanical device for evaluating polynomials was built from his original plans.

165 We should note that Calhoun was in office over 150 years ago. It goes without saying that no contemporary politician would use spurious statistics to support a wrong-headed position.

166 Given that he doesn't post, it is not surprising that jguttag has few followers. The difference between khemric and katyperry is harder to explain.

167 Statisticians, like attorneys and physicians, sometimes use Latin for no obvious reason other than to seem erudite. This phrase means, "with this, therefore because of this."

168 Stephen R. Johnson, "The Trouble with QSAR (or How I Learned to Stop Worrying and Embrace Fallacy)," *J. Chem. Inf.*

*Model.*, 2008.

169 Nelson HD, Humphrey LL, Nygren P, Teutsch SM, Allan JD. Postmenopausal hormone replacement therapy: scientific review. *JAMA*. 2002;288:872-881.

170 The move to online surveys, which allows students who do not attend class to participate in the survey, does not augur well for the egos of professors.

171 Eagles, John, et al., "Season of birth in females with anorexia nervosa in Northeast Scotland," *International Journal of Eating Disorders*, 30, 2, September 2001.

172 *Thinking, Fast and Slow*, Daniel Kahneman, Farrar, Straus and Giroux, 2011, p.175.

# 23

## EXPLORING DATA WITH PANDAS

Most of the second half of this book is focused on building various kinds of computational models that can be used to extract useful information from data. In the chapters following this one, we will take a quick look at simple ways to use machine learning to build models from data.

Before doing so, however, we will look at a popular library that can be used to quickly get acquainted with a dataset before diving into more detailed analysis. **Pandas**<sup>173</sup> is built on top of numpy. Pandas provides mechanisms to facilitate

- Organizing data
- Calculating simple statistics about data
- Storing the data in formats that facilitates future analysis

---

### 23.1 DataFrames and CSV Files

Everything in Pandas is built around the type **DataFrame**. A DataFrame is a mutable two-dimensional tabular data structure with labeled axes (rows and columns). One way to think about it is as a spreadsheet on steroids.

While DataFrames can be built from scratch using Python code, a more common way to create a DataFrame is by reading in a **CSV file**. As we saw in Chapter 19, each line of a CSV file consists of one or more values, separated by commas.<sup>174</sup> CSV files are typically used to store tabular numbers in plain text. In such cases, it is common for lines to have the same number of fields. Because they are plain text, they are often used to move data from one application to

another. For example, most spreadsheet programs allow users to write the contents of spreadsheet into a CSV file.

[Figure 23-1](#) shows a DataFrame containing information about the late rounds of the 2019 FIFA Women's World Cup. Each column represents something called a **series**. An **index** is associated with each row. By default, the indices are consecutive numbers, but they needn't be. A **name** is associated with each column. As we will see, these names play a role similar to that of keys in dictionaries.

The diagram illustrates a Pandas DataFrame named `wwc`. It consists of 8 rows and 6 columns. The columns are labeled: Round, Winner, W Goals, Loser, and L Goals. The rows are indexed from 0 to 7. Row 0 is labeled 'Quarters' under 'Round'. Row 1 is labeled 'Quarters'. Row 2 is labeled 'Quarters'. Row 3 is labeled 'Quarters'. Row 4 is labeled 'Semis'. Row 5 is labeled 'Semis'. Row 6 is labeled '3rd Place'. Row 7 is labeled 'Championship'. Arrows point to specific labels: 'Index (row) label' points to the index number 1; 'column label' points to the 'Round' header; 'columns axis = 1' points to the 'L Goals' header; and 'rows axis = 0' points to the index number 0.

	Round	Winner	W Goals	Loser	L Goals
0	Quarters	England	3	Norway	0
1	Quarters	USA	2	France	1
2	Quarters	Netherlands	2	Italy	0
3	Quarters	Sweden	2	Germany	1
4	Semis	USA	2	England	1
5	Semis	Netherlands	1	Sweden	0
6	3rd Place	Sweden	2	England	1
7	Championship	USA	2	Netherlands	0

[Figure 23-1](#) A sample Pandas DataFrame bound to the variable `wwc`

The DataFrame pictured in [Figure 23-1](#) was produced using the code below and the CSV file depicted in [Figure 23-2](#).

```
import pandas as pd
wwc = pd.read_csv('wwc2019_q-f.csv')
print(wwc)
```

```
Round,Winner,W Goals,L Goals
Quarters,England,3,Norway,0
Quarters,USA,2,France,1
Quarters,Netherlands,2,Italy,0
Quarters,Sweden,2,Germany,1
Semis,USA,2,England,1
Semis,Netherlands,1,Sweden,0
3rd Place,Sweden,2,England,1
Championship,USA,2,Netherlands,0
```

[Figure 23-2](#) An example CSV file

After importing Pandas, the code uses the Pandas' function `read_csv` to read the CSV file, and then prints it in the tabular form shown in [Figure 23-1](#). If the DataFrame has a large number of rows or columns, `print` will replace columns and/or rows in the center of the DataFrame with ellipses. This can be avoided by first converting the DataFrame to a string using the DataFrame method `to_string`.

Together, a row index and a column label indicate a data cell (as in a spreadsheet). We discuss how to access individual cells and groups of cells in Section 23.3. Typically, but not always, the cells in a column are all of the same type. In the DataFrame in [Figure 23-1](#), each of the cells in the `Round`, `Winner`, and `Loser` columns is of type `str`. The cells in the `W Goals` and `L Goals` columns are of type `numpy.int64`. You won't have a problem if you think of them as Python ints.

We can directly access the three components of a DataFrame using the attributes `index`, `columns`, and `values`.

The `index` attribute is of type `RangeIndex`. For example, the value of `wwc.index` is `RangeIndex(start=0, stop=8, step=1)`. Therefore, the code

```
for i in wwc.index:  
    print(i)
```

will print the integers 0-7 in ascending order.

The `columns` attribute is of type `Index`. For example, the value `wwc.columns` is `Index(['Round', 'Winner', 'W Goals', 'Loser', 'L Goals'], dtype='object')`, and the code

```
for c in wwc.columns:  
    print(c)
```

prints

```
Round  
Winner  
W Goals  
Loser  
L Goals
```

The `values` attribute is of type `numpy.ndarray`. In Chapter 13 we introduced the type `numpy.array`. It turns out that `array` is a special case of `ndarray`. Whereas arrays are one-dimensional (like other sequence types), ndarrays can be multidimensional. The number of dimensions and items in an ndarray is called its **shape** and is represented by a tuple of non-negative integers that specify the size of each dimension. The value of `wwc.values` is the two-dimensional ndarray

```
[['Quarters' 'England' 3 'Norway' 0]  
 ['Quarters' 'USA' 2 'France' 1]  
 ['Quarters' 'Netherlands' 2 'Italy' 0]  
 ['Quarters' 'Sweden' 2 'Germany' 1]  
 ['Semis' 'USA' 2 'England' 1]  
 ['Semis' 'Netherlands' 1 'Sweden' 0]  
 ['3rd Place' 'Sweden' 2 'England' 1]  
 ['Championship' 'USA' 2 'Netherlands' 0]]
```

Since it has eight rows and five columns, its shape is `(8, 5)`.

---

## 23.2 Creating Series and DataFrames

In practice, Pandas' DataFrames are typically created by loading a dataset that has been stored as either an SQL database, a CSV file, or in a format associated with a spreadsheet application. However, it is sometimes useful to construct series and DataFrames using Python code.

The expression `pd.DataFrame()` produces an empty DataFrame, and the statement `print(pd.DataFrame())` produces the output

```
Empty DataFrame  
Columns: []  
Index: []
```

A simple way to create a non-empty DataFrame is to pass in a list. For example, the code

```
rounds = ['Semis', 'Semis', '3rd Place', 'Championship']  
print(pd.DataFrame(rounds))
```

prints

```
          0  
0      Semis  
1      Semis  
2    3rd Place  
3  Championship
```

Notice that Pandas has automatically generated a label, albeit not a particularly descriptive one, for the DataFrame's only column. To get a more descriptive label, we can pass in a dictionary rather than a list. For example, the code `print(pd.DataFrame({'Round': rounds}))` prints

```
        Round  
0      Semis  
1      Semis  
2    3rd Place  
3  Championship
```

To directly create a DataFrame with multiple columns, we need only pass in a dictionary with multiple entries, each consisting of a column label as a key and a list as the value associated with each key. Each of these lists must be of the same length. For example, the code

```
rounds = ['Semis', 'Semis', '3rd Place', 'Championship']  
teams = ['USA', 'Netherlands', 'Sweden', 'USA']  
df = pd.DataFrame({'Round': rounds, 'Winner': teams})  
print(df)
```

prints

```
        Round      Winner  
0      Semis      USA  
1      Semis  Netherlands
```

2	3rd Place	Sweden
3	Championship	USA

Once a DataFrame has been created, it is easy to add columns. For example, the statement `df['W Goals'] = [2, 1, 0, 0]` mutates `df` so that its value becomes

	Round	Winner	W Goals
0	Semis	USA	2
1	Semis	Netherlands	1
2	3rd Place	Sweden	0
3	Championship	USA	0

Just as the values associated with a key in dictionary can be replaced, the values associated with a column can be replaced. For example, after executing the statement `df['W Goals'] = [2, 1, 2, 2]`, the value of `df` becomes

	Round	Winner	W Goals
0	Semis	USA	2
1	Semis	Netherlands	1
2	3rd Place	Sweden	2
3	Championship	USA	2

It is also easy to drop columns from a DataFrame. The function call `print(df.drop('Winner', axis = 'columns'))` prints

	Round	W Goals
0	Semis	2
1	Semis	1
2	3rd Place	2
3	Championship	2

and leaves `df` unchanged. If we had not included `axis = 'columns'` (or equivalently `axis = 1`) in the call to `drop`, the axis would have defaulted to `'rows'` (equivalent to `axis = 0`), which would have led to generating the exception `KeyError: "['Winner'] not found in axis."`

If a DataFrame is large, using `drop` in this way is inefficient, since it requires copying the DataFrame. The copy can be avoided by setting the `inplace` keyword argument to `drop` to `True`. The call `df.drop('Winner', axis = 'columns', inplace = True)` mutates `df` and returns `None`.

Rows can be added to the beginning or end of a DataFrame using the `DataFrame` constructor to create a new DataFrame, and then using the `concat` function to combine the new DataFrame with an existing DataFrame. For example, the code

```
quarters_dict = {'Round': ['Quarters']*4,
                 'Winner': ['England', 'USA', 'Netherlands',
                            'Sweden'],
                 'W Goals': [3, 2, 2, 2]}
df = pd.concat([pd.DataFrame(quarters_dict), df], sort = False)
```

sets `df` to

	Round	Winner	W Goals
0	Quarters	England	3
1	Quarters	USA	2
2	Quarters	Netherlands	2
3	Quarters	Sweden	2
0	Semis	USA	2
1	Semis	Netherlands	1
2	3rd Place	Sweden	2
3	Championship	USA	2

Had the keyword argument `sort` been set to `True`, `concat` would have also changed the order of the columns based upon lexicographic ordering of their labels. That is

```
pd.concat([pd.DataFrame(quarters_dict), df], sort = True)
```

swaps the position of the last two columns and returns the DataFrame

	Round	W Goals	Winner
0	Quarters	3	England
1	Quarters	2	USA
2	Quarters	2	Netherlands
3	Quarters	2	Sweden
0	Semis	2	USA
1	Semis	1	Netherlands
2	3rd Place	2	Sweden
3	Championship	2	USA

If no value for `sort` is provided, it defaults to `False`.

Notice that the indices of each of the concatenated DataFrames are unchanged. Consequently, there are multiple rows with the same index. The indices can be reset using the `reset_index` method. For example, the expression `df.reset_index(drop = True)` evaluates to

	Round	Winner	W Goals
0	Quarters	England	3
1	Quarters	USA	2
2	Quarters	Netherlands	2
3	Quarters	Sweden	2
4	Semis	USA	2
5	Semis	Netherlands	1
6	3rd Place	Sweden	2
7	Championship	USA	2

If `reset_index` is invoked with `drop = False`, a new column containing the old indices is added to the DataFrame. The column is labeled `index`.

You might be wondering why Pandas even allows duplicate indices. The reason is that it is often helpful to use a semantically meaningful index to label rows. For example, `df.set_index('Round')` evaluates to

Round	Winner	W Goals
Quarters	England	3
Quarters	USA	2
Quarters	Netherlands	2
Quarters	Sweden	2
Semis	USA	2
Semis	Netherlands	1
3rd Place	Sweden	2
Championship	USA	2

---

### 23.3 Selecting Columns and Rows

As is the case for other composite types in Python, square brackets are the primary mechanism for selecting parts of a DataFrame. To select a single column of a DataFrame, we simply place the label of the column in between square brackets. For example, `wwc['Winner']` evaluates to

```
0      England
1      USA
2  Netherlands
3      Sweden
4      USA
5  Netherlands
6      Sweden
7      USA
```

The type of this object is `series`, i.e., it is not a DataFrame. A Series is a one-dimensional sequence of values, each of which is labeled by an index. To select a single item from a Series, we place an index within square brackets following the series. So, `wwc['Winner'][3]` evaluates to the string `Sweden`.

We can iterate over a series using a `for` loop. For example,

```
winners = ''
for w in wwc['Winner']:
    winners += w + ','
print(winners[:-1])
```

prints `England,USA,Netherlands,Sweden,USA,Netherlands,Sweden,USA.`

**Finger exercise:** Write a function that returns the sum of the goals scored by winners.

Square brackets can also be used to select multiple columns from a DataFrame. This is done by placing a list of column labels within the square brackets. This produces a DataFrame rather than series. For example, `wwc[['Winner', 'Loser']]` produces the DataFrame

	Winner	Loser
0	England	Norway
1	USA	France
2	Netherlands	Italy
3	Sweden	Germany
4	USA	England
5	Netherlands	Sweden
6	Sweden	England
7	USA	Netherlands

The column labels in the list within the selection square brackets don't have to be in the same order as the labels appear in the original DataFrame. This makes it convenient to use selection to reorganize

the DataFrame. For example, `wwc[['Round', 'Winner', 'Loser', 'W Goals', 'L Goals']]` returns the DataFrame

	Round	Winner	Loser	W Goals	L Goals
0	Quarters	England	Norway	3	0
1	Quarters	USA	France	2	1
2	Quarters	Netherlands	Italy	2	0
3	Quarters	Sweden	Germany	2	1
4	Semis	USA	England	2	1
5	Semis	Netherlands	Sweden	1	0
6	3rd Place	Sweden	England	2	1
7	Championship	USA	Netherlands	2	0

Note that attempting to select a row by putting its index inside of square brackets will not work. It will generate a `KeyError` exception. Curiously, however, we can select rows using slicing. So, while `wwc[1]` causes an exception, `wwc[1:2]` produces a DataFrame with a single row,

	Round	Winner	W Goals	Loser	L Goals
1	Quarters	USA	2	France	1

We discuss other ways of selecting rows in the next subsection.

### 23.3.1 Selection Using loc and iloc

The `loc` method can be used to select rows, columns, or combinations of rows and columns from a DataFrame. Importantly, all selection is done using labels. This is worth emphasizing, since some of the labels (e.g., the indices) can look suspiciously like numbers.

If `df` is a DataFrame, the expression `df.loc[label]` returns a series corresponding to the row associated with `label` in `df`. For example, `wwc.loc[3]` returns the Series

Round	Quarters
Winner	Sweden
W Goals	2
Loser	Germany
L Goals	1

Notice that the column labels of `wwc` are the index labels for the Series, and the values associated with those labels are the values for

the corresponding columns in the row labeled `3` in `wwc`.

To select multiple rows, we need only put a list of labels (rather than a single label) inside the square brackets following `.loc`. When this is done, the value of the expression is a DataFrame rather than a Series. For example, the expression `wwc.loc[[1, 3, 5]]` produces

	Round	Winner	W Goals	Loser	L Goals
1	Quarters	USA	2	France	1
3	Quarters	Sweden	2	Germany	1
5	Semis	Netherlands	1	Sweden	0

Notice that the index associated with each row of the new DataFrame is the index of that row in the old DataFrame.

Slicing provides another way to select multiple rows. The general form is `df.loc[first:last:step]`. If `first` is not supplied, it defaults to the first index in the DataFrame. If `last` is not supplied, it defaults to the last index in the DataFrame. If `step` is not supplied, it defaults to `1`. The expression `wwc.loc[3:7:2]` produces the DataFrame

	Round	Winner	W Goals	Loser	L Goals
3	Quarters	Sweden	2	Germany	1
5	Semis	Netherlands	1	Sweden	0
7	Championship	USA	2	Netherlands	0

As a Python programmer, you might be surprised that that the row labeled `7` is included. For other Python data containers (such as lists), the last value is excluded when slicing, but not for DataFrames.<sup>175</sup> The expression `wwc.loc[6:]` produces the DataFrame

	Round	Winner	W Goals	Loser	L Goals
6	3rd Place	Sweden	2	England	1
7	Championship	USA	2	Netherlands	0

And the expression `wwc.loc[:2]` produces

	Round	Winner	W Goals	Loser	L Goals
0	Quarters	England	3	Norway	0
1	Quarters	USA	2	France	1
2	Quarters	Netherlands	2	Italy	0

**Finger exercise:** Write an expression that selects all even numbered rows in `wwc`.

As we mentioned earlier, `loc` can be used to simultaneously select a combination of rows and columns. This is done with an expression of the form

```
df.loc[row_selector, column_selector]
```

The row and column selectors can be written using any of the mechanisms already discussed, i.e., a single label, a list of labels, or a slicing expression. For example, `wwc.loc[0:2, 'Round':'L Goals':2]` produces

	Round	W Goals	L Goals
0	Quarters	3	0
1	Quarters	2	1
2	Quarters	2	0

**Finger exercise:** Write an expression that generates the DataFrame

	Round	Winner	W Goals	Loser	L Goals
1	Quarters	USA	2	France	1
2	Quarters	Netherlands	2	Italy	0

Thus far, you wouldn't have gone wrong if you thought of the index labels as integers. Let's see how selection works when 1) the labels are not number-like, and 2) more than one row has the same label. Let `wwc_by_round` be the DataFrame

Round	Winner	W Goals	Loser	L Goals
Quarters	England	3	Norway	0
Quarters	USA	2	France	1
Quarters	Netherlands	2	Italy	0
Quarters	Sweden	2	Germany	1
Semis	USA	2	England	1
Semis	Netherlands	1	Sweden	0
3rd Place	Sweden	2	England	1
Championship	USA	2	Netherlands	0

What do you think the expression `wwc_by_round.loc['Semis']` evaluates to? It selects all rows with the label `Semis` to return

Round	Winner	W Goals	Loser	L Goals
-------	--------	---------	-------	---------

Semis	USA	2	England	1
Semis	Netherlands	1	Sweden	0

Similarly, `wwc_by_round.loc[['Semis', 'Championship']]` selects all rows with a label of either Semis or Championship:

Round	Winner	W Goals	Loser	L Goals
Semis	USA	2	England	1
Semis	Netherlands	1	Sweden	0
Championship	USA	2	Netherlands	0

Slicing also work with non-numeric indices. The expression

```
wwc_by_round.loc['Quarters':'Semis':2]
```

produces a DataFrame by selecting the first row labeled by Quarters and then selecting every other row until it has passed a row labeled Semis to generate

Round	Winner	W Goals	Loser	L Goals
Quarters	England	3	Norway	0
Quarters	Netherlands	2	Italy	0
Semis	USA	2	England	1

Now, suppose we want to select the second and third of the rows labeled Quarters. We can't simply write `wwc_by_round.loc['Quarters']` because that will select all four rows labeled Quarters. Enter the `iloc` method.

The `iloc` method is like `loc`, except rather than working with labels, it works with integers (hence the `i` in `iloc`). The first row of a DataFrame is `iloc 0`, the second at `iloc 1`, etc. So, to select the second and third of the rows labeled Quarters, we write `wwc_by_round.iloc[[1,2]]`.

### 23.3.2 Selection by Group

It is often convenient to split a DataFrame into subsets and apply some aggregation or transformation separately to each subset. The `groupby` method makes it easy to do this sort of thing.

Suppose, for example, we want to know the total number of goals scored by the winning and losing teams in each round. The code

```
grouped_by_round = wwc.groupby('Round')
```

binds `group_by_round` to an object of type `DataFrameGroupBy`. We can then apply the aggregator `sum` to that object to generate a `DataFrame`. The code

```
grouped_by_round = wwc.groupby('Round')
print(grouped_by_round.sum())
```

prints

Round	W Goals	L Goals
3rd Place	2	1
Championship	2	0
Quarters	9	2
Semis	3	1

The code `print(wwc.groupby('Winner').mean())` prints

Winner	W Goals	L Goals
England	3.0	0.000000
Netherlands	1.5	0.000000
Sweden	2.0	1.000000
USA	2.0	0.666667

From this we can easily see that England averaged three goals in the games it won, while shutting out its opponents.

The code `print(wwc.groupby(['Loser', 'Round']).mean())` prints

Loser	Round	W Goals	L Goals
England	3rd Place	2	1
	Semis	2	1
France	Quarters	2	1
Germany	Quarters	2	1
Italy	Quarters	2	0
Netherlands	Championship	2	0
Norway	Quarters	3	0
Sweden	Semis	1	0

From this we can easily see that England averaged one goal in the games it lost, while giving up two.

### 23.3.3 Selection by Content

Suppose we want to select all of the rows for games won by Sweden from the DataFrame in [Figure 23-1](#). Since this DataFrame is a small one, we could look at each row and find the indices of the rows corresponding to those games. Of course, that approach doesn't scale to large DataFrames. Fortunately, it is easy to select rows based on their contents using something called **Boolean indexing**.

The basic idea is to write a logical expression referring to the values contained in the DataFrame. That expression is then evaluated on each row of the DataFrame, and the rows for which it evaluates to `True` are selected. The expression `wwc.loc[wwc['Winner'] == 'Sweden']` evaluates to the DataFrame

	Round	Winner	W Goals	Loser	L Goals
3	Quarters	Sweden	2	Germany	1
6	3rd Place	Sweden	2	England	1

Retrieving all of the games involving Sweden is only a little more complicated. The logical operators `&` (corresponding to `and`), `|` (corresponding to `or`), and `-` (corresponding to `not`) can be used to form expressions. The expression `wwc.loc[(wwc['Winner'] == 'Sweden') | (wwc['Loser'] == 'Sweden')]` returns

	Round	Winner	W Goals	Loser	L Goals
3	Quarters	Sweden	2	Germany	1
5	Semis	Netherlands	1	Sweden	0
6	3rd Place	Sweden	2	England	1

Beware, the parentheses around the two subterms of the logical expression are necessary because in Pandas `|` has higher precedence than `==`.

**Finger exercise:** Write an expression that returns a DataFrame containing games in which the USA but not France played.

If we expect to do many queries selecting games in which a country participated, it might be convenient to define the function

```
def get_country(df, country):  
    """df a DataFrame with series labeled Winner and Loser  
    country a str  
    returns a DataFrame with all rows in which country
```

```

    appears
        in either the Winner or Loser column"""
    return df.loc[(df['Winner'] == country) | (df['Loser']
== country)]

```

Since `get_country` returns a DataFrame, it is easy to extract the games between pairs of teams by composing two calls of `get_country`. For example, evaluating `get_country(get_country(wwc, 'Sweden'), 'Germany')` extracts the one game (teams play each other at most once during a knockout round) between these two teams.

Suppose we want to generalize `get_country` so that it accepts a list of countries as an argument and returns all games in which any of the countries in the list played. We can do this using the `isin` method:

```

def get_games(df, countries):
    return df[(df['Winner'].isin(countries)) |
              (df['Loser'].isin(countries))]

```

The `isin` method filters a DataFrame by selecting only those rows with a specified value (or element of a specified collection of values) in a specified column. The expression `df['Winner'].isin(countries)` in the implementation of `get_games` selects those rows in `df` in which the column `Winner` contains an element in the list `countries`.

**Finger exercise:** Print a DataFrame containing only the games in which Sweden played either Germany or Netherlands.

## 23.4 Manipulating the Data in a DataFrame

We've now looked at some simple ways to create and select parts of DataFrames. One of the things that makes DataFrames worth creating is the ease of extracting aggregate information from them. Let's start by looking at some ways we might extract aggregate information from the DataFrame `wwc`, pictured in [Figure 23-1](#).

The columns of a DataFrame can be operated on in ways that are analogous to the ways we operate on numpy arrays. For example, analogous to the way the expression `2*np.array([1,2,3])` evaluates to the array `[2 4 6]`, the expression `2*wwc['W Goals']` evaluates to the series

```
0      6
1      4
2      4
3      4
4      4
5      2
6      4
7      4
```

The expression `wwc['W Goals'].sum()` sums the values in the `W Goals` column to produce the value 16. Similarly, the expression

```
(wwc[wwc['Winner'] == 'Sweden']['W Goals'].sum() +
wwc[wwc['Winner'] == 'Sweden']['L Goals'].sum())
```

computes the total number of goals scored by Sweden, 6, and the expression

```
(wwc['W Goals'].sum() - wwc['L Goals'].sum())/len(wwc['W Goals'])
```

computes the mean goal differential of the games in the DataFrame, 1.5.

**Finger exercise:** Write an expression that computes the total number of goals scored in all of the rounds.

**Finger exercise:** Write an expression that computes the total number of goals scored by the losing teams in the quarter finals.

Suppose we want to add a column containing the goal differential for all of the games and add a row summarizing the totals for all the columns containing numbers. Adding the column is simple. We merely execute `wwc['G Diff'] = wwc['W Goals'] - wwc['L Goals']`. Adding the row is more involved. We first create a dictionary with the contents of the desired row, and then use that dictionary to create a new DataFrame containing only the new row. We then use the `concat` function to concatenate `wwc` and the new DataFrame.

```
#Add new column to wwc
wwc['G Diff'] = wwc['W Goals'] - wwc['L Goals']
#create a dict with values for new row
new_row_dict = {'Round': ['Total'],
                'W Goals': [wwc['W Goals'].sum()],
```

```

'L Goals': [wwc['L Goals'].sum()],
'G Diff': [wwc['G Diff'].sum()]}
#Create DataFrame from dict, then pass it to concat
new_row = pd.DataFrame(new_row_dict)
wwc = pd.concat([wwc, new_row], sort =
False).reset_index(drop = True)

```

This code produces the DataFrame

G Diff	Round	Winner	W Goals	Loser	L Goals
0	Quarters	England	3	Norway	0
1	Quarters	USA	2	France	1
2	Quarters	Netherlands	2	Italy	0
3	Quarters	Sweden	2	Germany	1
4	Semis	USA	2	England	1
5	Semis	Netherlands	1	Sweden	0
6	3rd Place	Sweden	2	England	1
7	Championship	USA	2	Netherlands	0
8	Total	NaN	16	NaN	4
12					

Notice that when we tried to sum the values in columns that did not contain numbers, Pandas did not generate an exception. Instead it supplied the special value `NaN` (Not a Number).

In addition to providing simple arithmetic operations like `sum` and `mean`, Pandas provides methods for computing a variety of useful statistical functions. Among the most useful of these is `corr`, which is used to compute the **correlation** between two series.

A correlation is a number between `-1` and `1` that provides information about the relationship between two numeric values. A positive correlation indicates that as the value of one variable increases, so does the value of the other. A negative correlation indicates that as the value of one variable increases, the value of the other variable decreases. A correlation of zero indicates that there is no relation between the values of the variables.

The most commonly used measure of correlation is Pearson correlation. Pearson correlation measures the strength and direction of the linear relationship between two variables. In addition to Pearson correlation, Pandas supports two other measures of correlation, Spearman and Kendall. There are important differences among the three measures (e.g., Spearman is less sensitive to outliers than Pearson, but is useful only for discovering monotonic relationships), but a discussion of when to use which is beyond the scope of this book.

To print the Pearson pairwise correlations of `W Goals`, `L Goals`, and `G Diff` for all of the games (and exclude the row with the totals), we need only execute

```
print(wwc.loc[wwc['Round'] != 'Total'].corr(method =  
'pearson'))
```

which produces

	W Goals	L Goals	G Diff
W Goals	1.000000	0.000000	0.707107
L Goals	0.000000	1.000000	-0.707107
G Diff	0.707107	-0.707107	1.000000

The values along the diagonal are all 1, because each series is perfectly positively correlated with itself. Unsurprisingly, the goal differentials are strongly positively correlated with the number of goals scored by the winning team, and strongly negatively correlated with the number of goals scored by the loser. The weaker negative correlation between the goals scored by the winners and losers also makes sense for professional soccer.<sup>[176](#)</sup>

---

## 23.5 An Extended Example

In this section we will look at two datasets, one containing historical temperature data for 21 U.S. cities and the other historical data about the global use of fossil fuels.

### 23.5.1 Temperature Data

The code

```

pd.set_option('display.max_rows', 6)
pd.set_option('display.max_columns', 5)
temperatures = pd.read_csv('US_temperatures.csv')
print(temperatures)

```

prints

	Date	Albuquerque	...	St Louis	Tampa
0	19610101	-0.55	...	-0.55	15.00
1	19610102	-2.50	...	-0.55	13.60
2	19610103	-2.50	...	0.30	11.95
	...	...	...	...	...
20085	20151229	-2.15	...	1.40	26.10
20086	20151230	-2.75	...	0.60	25.55
20087	20151231	-0.75	...	-0.25	25.55
	[20088 rows x 22 columns]				

The first two lines of code set default options that limit the number of rows and columns shown when printing DataFrames. These options play a role similar to that played by the `rcParams` we used for setting various default values for plotting. The function `reset_option` can be used to set an option back to the system default value.

This DataFrame is organized in a way that makes it easy to see what the weather was like in different cities on specific dates. For example, the query

```
temperatures.loc[temperatures['Date']==19790812][['New
York','Tampa']]
```

tells us that on August 12, 1979, the temperature in New York was 15C and in Tampa 25.55C.

**Finger exercise:** Write an expression that evaluates to `True` if Phoenix was warmer than Tampa on October 31, 2000, and `False` otherwise.

**Finger exercise:** Write code to extract the date on which the temperature in Phoenix was 41.4C.<sup>[177](#)</sup>

Unfortunately, looking at data from 21 cities for 20,088 dates doesn't give us much direct insight into larger questions related to temperature trends. Let's start by adding columns that provide summary information about the temperatures each day. The code

```

temperatures['Max T'] = temperatures.max(axis = 'columns')
temperatures['Min T'] = temperatures.min(axis = 'columns')
temperatures['Mean T'] = round(temperatures.mean(axis =
'columns'), 2)
print(temperatures.loc[20000704:20000704])

```

prints

	Date	Albuquerque	...	Min T	Mean T
14429	20000704	26.65	...	15.25	1666747.37

Was the mean temperature of those 21 cities on July 4, 2000, really much higher than the temperature on the surface of the sun? Probably not. It seems more likely that there is a bug in our code. The problem is that our DataFrame encodes dates as numbers, and these numbers are used to compute the mean of each row. Conceptually, it might make more sense to think of the date as an index for a series of temperatures. So, let's change the DataFrame to make the dates indices. The code

```

temperatures.set_index('Date', drop = True, inplace = True)
temperatures['Max'] = temperatures.max(axis = 'columns')
temperatures['Min'] = temperatures.min(axis = 'columns')
temperatures['Mean T'] = round(temperatures.mean(axis =
'columns'), 2)
print(temperatures.loc[20000704:20000704])

```

prints the more plausible

	Albuquerque	Baltimore	...	Min T	Mean T
Date			...		
20000704	26.65	25.55	...	15.25	24.42

Notice, by the way, that since `Date` is no longer a column label, we had to use a different print statement. Why did we use slicing to select a single row? Because we wanted to create a DataFrame rather than a series.

We are now in a position to start producing some plots showing various trends. For example,

```

plt.figure(figsize = (14, 3)) #set aspect ratio for figure
plt.plot(list(temperatures['Mean T']))
plt.title('Mean Temp Across 21 US Cities')

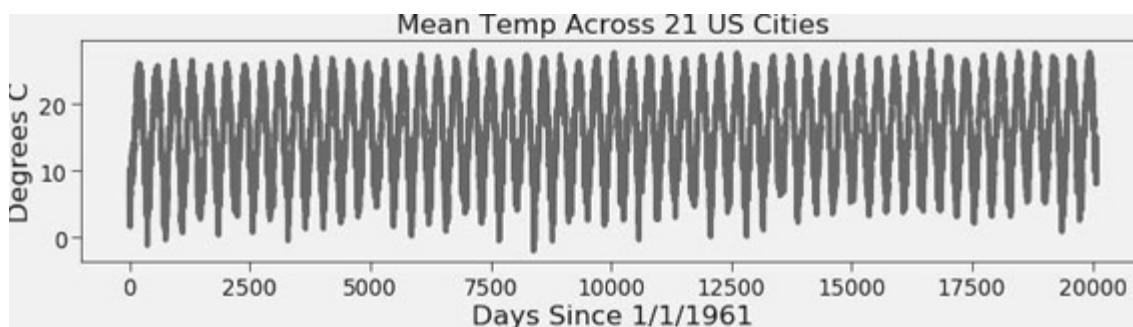
```

```

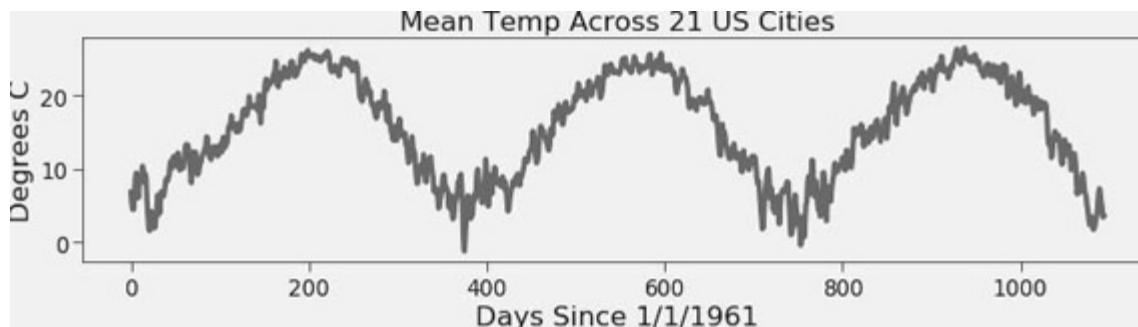
plt.xlabel('Days Since 1/1/1961')
plt.ylabel('Degrees C')

```

produces a plot that shows the seasonality of temperatures in the United States. Notice that before plotting the mean temperatures we cast the series into a list. Had we plotted the series directly, it would have used the indices of the series (integers representing dates) for the x-axis. This would have produced a rather odd-looking plot, since the points on the x-axis would have been strangely spaced. For example, the distance between December 30, 1961 and December 31, 1961 would have been 1, but the distance between December 31, 1961 and January 1, 1962 would have been 8870 ( $19620,101 - 19611231$ ).



We can see the seasonal pattern more clearly, by zooming in on a few years and producing a plot using the call plt.plot(list(temperatures['Mean T']) [0:3\*365]).



Over the last decades, a consensus that the Earth is warming has emerged. Let's see whether this data is consistent with that

consensus. Since we are investigating a hypothesis about a long-term trend, we should probably not be looking at daily or seasonal variations in temperature. Instead, let's look at annual data.

As a first step, let's use the data in `temperatures` to build a new DataFrame in which the rows represent years rather than days. Code that does this is contained in [Figure 23-3](#) and [Figure 23-4](#). Most of the work is done in the function `get_dict`, [Figure 23-3](#), which returns a dictionary mapping a year to a dictionary giving the values for that year associated with different labels. The implementation of `get_dict` iterates over the rows in `temperatures` using `iterrows`. That method returns an iterator that for each row returns a pair containing the index label and the contents of the row as a series. Elements of the yielded series can be selected using column labels.[178](#)

```
def get_dict(temperatures, labels):
    """temperatures a DataFrame. Its indices are ints
       representing dates of the form yyyymmdd
       labels a list of column labels
       returns a dict with strs representing years as keys,
          the values dicts with the columns as keys, and
          a list of the daily temperatures in that column for
          that year as values
    """
    year_dict = {}
    for index, row in temperatures.iterrows():
        year = str(index)[0:4]
        try:
            for col in labels:
                year_dict[year][col].append(row[col])
        except:
            year_dict[year] = {col:[] for col in labels}
            for col in labels:
                year_dict[year][col].append(row[col])
    return year_dict
```

[Figure 23-3](#) Building a dictionary mapping years to temperature data

If `test` were the DataFrame

	Max T	Min T	Mean T
Date			
19611230	24.70	-13.35	3.35

```

19611231  24.75 -10.25    5.10
19620101  25.55 -10.00    5.70
19620102  25.85  -4.45    6.05

```

the call `get_dict(test, ['Max', 'Min'])` would return the dictionary

```

{'1961': {'Max T': [24.7, 24.75], 'Min T': [-13.35, -10.25],
'Mean T': [3.35, 5.1]}, '1962': {'Max T': [25.55, 25.85],
'Min T': [-10.0, -4.45], 'Mean T': [5.7, 6.05]}}

```

```

temperatures = pd.read_csv('US_temperatures.csv')
temperatures.set_index('Date', drop = True, inplace = True)
temperatures['Mean T'] = round(temperatures.mean(axis = 'columns'), 2)
temperatures['Max T'] = temperatures.max(axis = 'columns')
temperatures['Min T'] = temperatures.min(axis = 'columns')
yearly_dict = get_dict(temperatures, ['Max T', 'Min T', 'Mean T'])
years, mins, maxes, means = [], [], [], []
for y in yearly_dict:
    years.append(y)
    mins.append(min(yearly_dict[y]['Min T']))
    maxes.append(max(yearly_dict[y]['Max T']))
    means.append(round(np.mean(yearly_dict[y]['Mean T']), 2))

yearly_temps = pd.DataFrame({'Year': years, 'Min T': mins,
                             'Max T': maxes, 'Mean T': means})
print(yearly_temps)

```

[Figure 23-4](#) Building a DataFrame organized around years

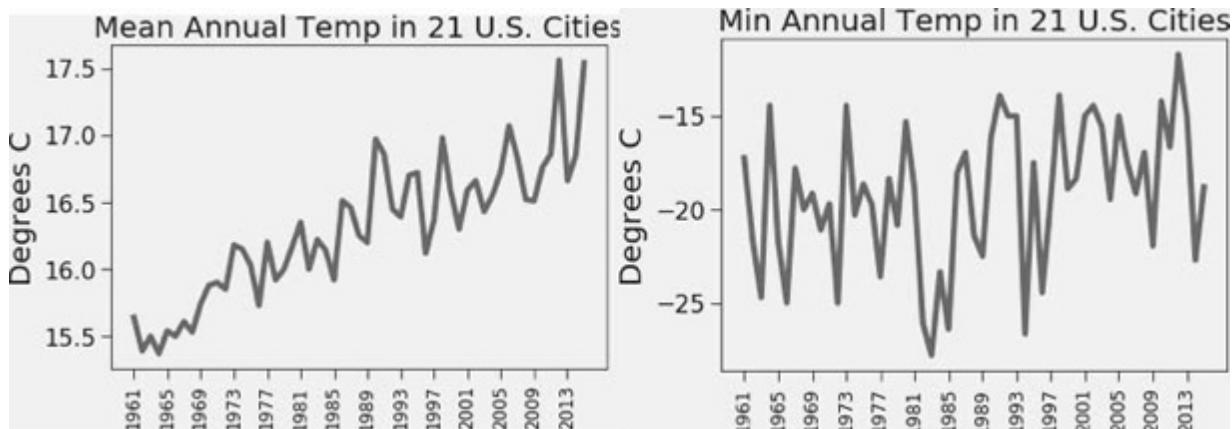
The code following the invocation of `get_dict` in [Figure 23-4](#) builds a list containing each year appearing in `temperatures`, and additional lists containing the minimum, maximum, and mean temperatures for those years. Finally it uses those lists to build the DataFrame `yearly_temps`:

	Year	Min T	Max T	Mean T
0	1961	-17.25	38.05	15.64
1	1962	-21.65	36.95	15.39
2	1963	-24.70	36.10	15.50
..	...	...	...	...
52	2013	-15.00	40.55	16.66
53	2014	-22.70	40.30	16.85
54	2015	-18.80	40.55	17.54

Now that we have the data in a convenient format, let's generate some plots to visualize how the temperatures change over time. The code in [Figure 23-5](#) produced the plots in [Figure 23-6](#).

```
plt.figure(0)
plt.plot(yearly_temps['Year'], yearly_temps['Mean T'])
plt.title('Mean Annual Temp in 21 U.S. Cities')
plt.figure(1)
plt.plot(yearly_temps['Year'], yearly_temps['Min T'])
plt.title('Min Annual Temp in 21 U.S. Cities')
for i in range(2):
    plt.figure(i)
    plt.xticks(range(0, len(yearly_temps), 4),
               rotation = 'vertical', size = 'large')
    plt.ylabel('Degrees C')
```

[Figure 23-5](#) Produce plots relating year to temperature measurements

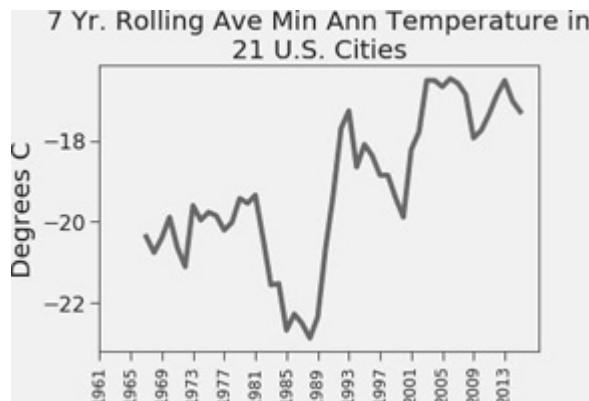


[Figure 23-6](#) Mean and minimum annual temperatures

The plot on the left in [Figure 23-6](#) shows an undeniable trend;<sup>179</sup> the mean temperatures in these 21 cities has risen over time. The plot on the right is less clear. The extreme annual fluctuations make it hard to see a trend. A more revealing plot can be produced by plotting a **moving average** of the temperatures.

The Pandas method `rolling` is used to perform an operation on multiple consecutive values of a series. Evaluating the expression `yearly_temps['Min T'].rolling(7).mean()` produces a series in

which the first 6 values are `NaN`, and for each  $i$  greater than 6, the  $i$ th value in the series is the mean of `yearly_temps['Min'][i-6:i+1]`. Plotting that series against the year produces the plot in [Figure 23-7](#), which does suggest a trend.



[Figure 23-7](#) Rolling average minimum temperatures

While visualizing the relationship between two series can be informative, it is often useful to look at those relationships more quantitatively. Let's start by looking at the correlations between years and the seven-year rolling averages of the minimum, maximum, and mean temperatures. Before computing the correlations, we first update the series in `yearly_temps` to contain rolling averages and then convert the year values from strings to integers. The code

```
num_years = 7
for label in ['Min T', 'Max T', 'Mean T']:
    yearly_temps[label] =
        yearly_temps[label].rolling(num_years).mean()
yearly_temps['Year'] = yearly_temps['Year'].apply(int)
print(yearly_temps.corr())
```

prints

	Year	Min T	Max T	Mean T
Year	1.000000	0.713382	0.918975	0.969475
Min T	0.713382	1.000000	0.629268	0.680766
Max T	0.918975	0.629268	1.000000	0.942378
Mean T	0.969475	0.680766	0.942378	1.000000

All of the summary temperature values are positively correlated with the year, with the mean temperatures the most strongly correlated. That raises the question of how much of the variance in the rolling average of the mean temperatures is explained by the year. The following code prints the coefficient of determination (Section 20.2.1).

```
indices = np.isfinite(yearly_temps['Mean T'])
model = np.polyfit(list(yearly_temps['Year'][indices]),
                   list(yearly_temps['Mean T'][indices]), 1)
print(r_squared(yearly_temps['Mean T'][indices],
                np.polyval(model, yearly_temps['Year']
[indices])))
```

Since some of the values in the `Mean` series are `NaN`, we first use the function `np.isfinite` to get the indices of the non-`NaN` values in `yearly_temps['Mean']`. We then build a linear model and finally use the `r_squared` function (see Figure 20-13) to compare the results predicted by the model to the actual temperatures. The linear model relating years to the seven-year rolling average mean temperature explains nearly 94% of the variance.

**Finger exercise:** Find the coefficient of determination ( $r^2$ ) for the mean annual temperature rather than for the rolling average and for a ten-year rolling average.

If you happen to live in the U.S. or plan to travel to the U.S., you might be more interested in looking at the data by city rather than year. Let's start by producing a new DataFrame that provides summary data for each city. In deference to our American readers, we convert all temperatures to Fahrenheit by applying a conversion function to all values in `city_temps`. The penultimate line adds a column showing how extreme the temperature variation is. Executing this code produces the DataFrame in [Figure 23-8](#).<sup>180</sup>

```
temperatures = pd.read_csv('US_temperatures.csv')
temperatures.drop('Date', axis = 'columns', inplace = True)
means = round(temperatures.mean(), 2)
maxes = temperatures.max()
mins = temperatures.min()
city_temps = pd.DataFrame({'Min T':mins, 'Max T':maxes,
                           'Mean T':means})
```

```

city_temps = city_temps.apply(lambda x: 1.8*x + 32)
city_temps['Max-Min'] = city_temps['Max T'] -
city_temps['Min T']
print(city_temps.sort_values('Mean T', ascending =
False).to_string())

```

	Min T	Max T	Mean T	Max-Min
San Juan	68.99	88.97	80.492	19.98
Miami	37.94	90.05	76.604	52.11
Phoenix	32.45	106.52	73.904	74.07
Tampa	28.94	89.06	72.878	60.12
New Orleans	18.95	90.95	68.882	72.00
Las Vegas	19.49	105.98	67.964	86.49
Dallas	8.51	97.52	66.092	89.01
San Diego	43.07	92.03	64.130	48.96
Los Angeles	42.98	94.01	63.158	51.03
Charlotte	9.50	90.50	60.512	81.00
San Francisco	30.56	86.00	57.632	55.44
Albuquerque	-3.46	89.96	57.110	93.42
St Louis	-8.50	96.98	56.408	105.48
Baltimore	-0.04	93.47	55.562	93.51
Philadelphia	0.50	92.48	55.364	91.98
New York	3.56	91.04	54.194	87.48
Portland	11.03	89.96	54.068	78.93
Seattle	12.02	86.99	52.376	74.97
Boston	1.04	92.48	51.620	91.44
Chicago	-18.04	92.48	49.622	110.52
Detroit	-12.01	89.51	49.550	101.52

[Figure 23-8](#) Average temperatures for select cities

To visualize differences among cities, we generated the plot in [Figure 23-9](#) using the code

```

plt.plot(city_temps.sort_values('Max-Min', ascending=False)
         ['Max-Min'], 'o')
plt.figure()
plt.plot(city_temps.sort_values('Max-Min', ascending=False)
         ['Min T'],
         'b^', label = 'Min T')
plt.plot(city_temps.sort_values('Max-Min', ascending=False)
         ['Max T'],
         'kx', label = 'Max T')
plt.plot(city_temps.sort_values('Max-Min', ascending=False)
         ['Mean T'],

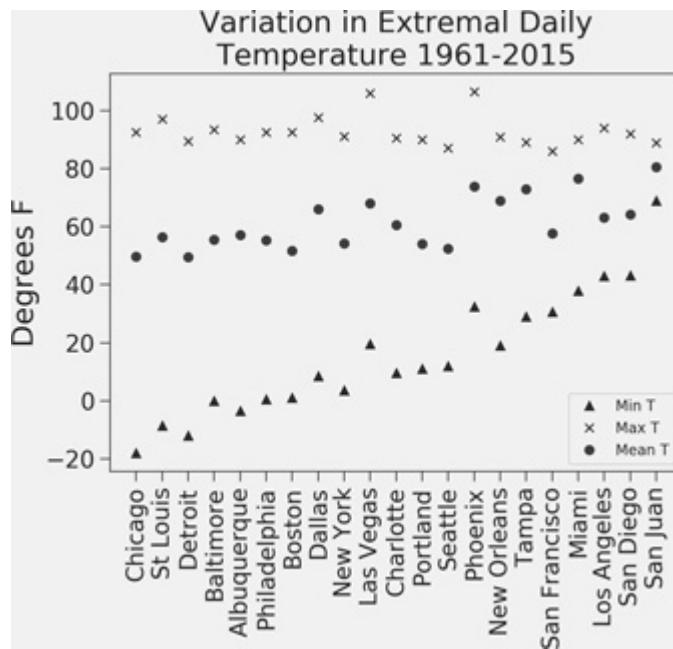
```

```

    'ro', label = 'Mean T')
plt.xticks(rotation = 'vertical')
plt.legend()
plt.title('Variation in Extremal Daily\nTemperature 1961-
2015')
plt.ylabel('Degrees F')

```

Notice that we used the sort order `Max - Min` for all three series. The use of `ascending = False` reverses the default sorting order.



[Figure 23-9](#) Variation in temperature extremes

Looking at this plot we can see, among other things, that

- Across cities, the minimum temperature differs much more than the maximum temperature. Because of this,  $\text{Max} - \text{Min}$  (the sort order) is strongly positively correlated with the minimum temperature.
- It never gets very hot in San Francisco or Seattle.
- The temperature in San Juan is close to constant.
- The temperature in Chicago is not close to constant. It gets both quite hot and frighteningly cold in the windy city.

- It gets uncomfortably hot in both Phoenix and Las Vegas.
- San Francisco and Albuquerque have about the same mean temperature, but radically different minima and maxima.

### 23.5.2 Fossil Fuel Consumption

The file `global-fossil-fuel-consumption.csv` contains data about the yearly consumption of fossil fuels on Earth from 1965 and 2015. The code

```
emissions = pd.read_csv('global-fossil-fuel-
consumption.csv')
print(emissions)
```

prints

	Year	Coal	Crude Oil	Natural Gas
0	1965	16151.96017	18054.69004	6306.370076
1	1966	16332.01679	19442.23715	6871.686791
2	1967	16071.18119	20830.13575	7377.525476
..	..	..	..	..
50	2015	43786.84580	52053.27008	34741.883490
51	2016	43101.23216	53001.86598	35741.829870
52	2017	43397.13549	53752.27638	36703.965870

Now, let's replace the columns showing the consumption of each kind of fuel by two columns, one showing the sum of the three, and the other the five-year rolling average of the sum.

```
emissions['Fuels'] = emissions.sum(axis = 'columns')
emissions.drop(['Coal', 'Crude Oil', 'Natural Gas'], axis =
'columns',
               inplace = True)
num_years = 5
emissions['Roll F'] =\
    emissions['Fuels'].rolling(num_years).mean()
emissions = emissions.round()
```

We can plot this data using

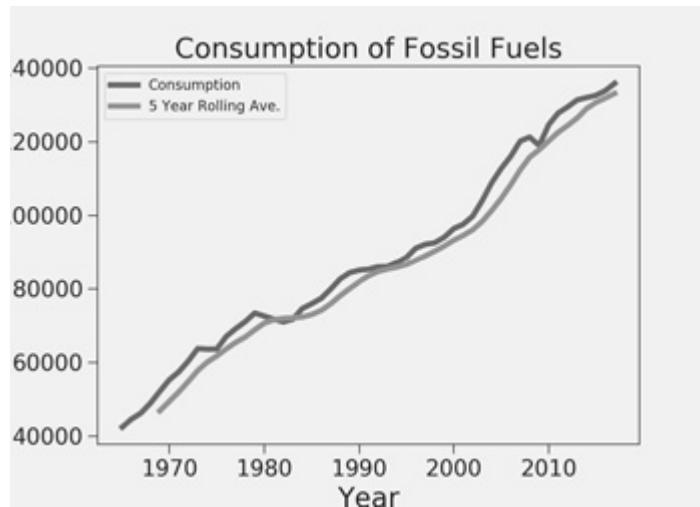
```
plt.plot(emissions['Year'], emissions['Fuels'],
         label = 'Consumption')
plt.plot(emissions['Year'], emissions['Roll F'],
         label = str(num_years) + ' Year Rolling Ave.')
plt.legend()
```

```

plt.title('Consumption of Fossil Fuels')
plt.xlabel('Year')
plt.ylabel('Consumption')

```

to get the plot in [Figure 23-10](#).



[Figure 23-10](#) Global consumption of fossil fuels

While there are a few small dips in consumption (e.g., around the 2008 financial crisis), the upward trend is unmistakable.

The scientific community has reached consensus that there is an association between this rise in fuel consumption and the rise in the average temperature on the planet. Let's see how it relates to the temperatures in the 21 U.S. cities we looked at in Section 23.5.1.

Recall that `yearly_temps` was bound to the DataFrame

	Year	Min T	Max T	Mean T
0	1961	-17.25	38.05	15.64
1	1962	-21.65	36.95	15.39
2	1963	-24.70	36.10	15.50
..	...	...	...	...
52	2013	-15.00	40.55	16.66
53	2014	-22.70	40.30	16.85
54	2015	-18.80	40.55	17.54

Wouldn't it be nice if there were an easy way to combine `yearly_temps` and `emissions`? Pandas' `merge` function does just that. The code

```

yearly_temps['Year'] = yearly_temps['Year'].astype(int)
merged_df = pd.merge(yearly_temps, emissions,
                     left_on = 'Year', right_on = 'Year')
print(merged_df)

```

prints the DataFrame

	Year	Min T	...	Fuels	Roll F
0	1965	-21.7	...	42478.0	NaN
1	1966	-25.0	...	44612.0	NaN
2	1967	-17.8	...	46246.0	NaN
..	...	...	...	...	...
48	2013	-15.0	...	131379.0	126466.0
49	2014	-22.7	...	132028.0	129072.0
50	2015	-18.8	...	132597.0	130662.0

The DataFrame contains the union of the columns appearing in `yearly_temps` and `emissions` but includes only rows built from the rows in `yearly_temps` and `emissions` that contain the same value in the `Year` column.

Now that we have the emissions and temperature information in the same DataFrame, it is easy to look at how things are correlated with each other. The code

```
print(merged_df.corr().round(2).to_string())
```

prints

	Year	Min T	Max T	Mean T	Fuels	Roll F
Year	1.00	0.37	0.72	0.85	0.99	0.98
Min T	0.37	1.00	0.22	0.49	0.37	0.33
Max T	0.72	0.22	1.00	0.70	0.75	0.66
Mean T	0.85	0.49	0.70	1.00	0.85	0.81
Fuels	0.99	0.37	0.75	0.85	1.00	1.00
Roll F	0.98	0.33	0.66	0.81	1.00	1.00

We see that global fuel consumption in previous years is indeed highly correlated with both the mean and maximum temperature in these U.S. cities. Does this imply that increased fuel consumption is causing the rise in temperature? It does not. Notice that both are highly correlated with year. Perhaps some lurking variable is also correlated with year and is the causal factor. What we can say from a statistical perspective, is that the data does not contradict the widely accepted scientific hypothesis that the increased use of fossil fuels generates greenhouse gasses that have caused temperatures to rise.

This concludes our brief look at Pandas. We have only scratched the surface of what it offers. We will use it later in the book and introduce a few more features. If you want to learn more, there are many online resources and some excellent inexpensive books. The website <https://www.dataschool.io/best-python-pandas-resources/> lists some of these.

---

## 23.6 Terms Introduced in Chapter

DataFrame

row

series

index

name

CSV file

shape (of ndarray)

Boolean indexing

correlation of series

moving (rolling) average

---

**173** Disappointingly, the name Pandas has nothing to do with the cute-looking animal. The name was derived from the term “panel data,” an econometrics term for data that includes observations over multiple time periods.

**174** CSV is an acronym for comma-separated values.

**175** As Ralph Waldo Emerson said, “foolish consistency is the hobgoblin of small minds.” Unfortunately, the difference between foolish consistency and sensible consistency is not always clear.

176 This relationship does not hold for all sports. For example, the number of points scored by the winner and the loser in NBA games are positively correlated.

177 For those among you who don't think in degrees C, that's a temperature of almost 106F. And that was the average temperature that day. The high that day was 122F (50C)! Once you have retrieved the date, you might enjoy reading about it online.

178 The method `iteruples` can also be used to iterate over the rows of the DataFrame. It yields tuples rather than series. It is considerably faster than `iterrows`. It is also less intuitive to use since elements of the yielded tuple are selected by position in the tuple rather than column name.

179 Perhaps "undeniable" is too strong a word since there do seem to be some deniers.

180 I confess to being surprised that the minimum temperature for Boston was over 0, since I recall being out too many times in below 0F weather. Then I remembered that the temperatures in the original csv file were the average temperature over a day, so the minimum does not capture the actual low point of the day.

## 24

# A QUICK LOOK AT MACHINE LEARNING

The amount of digital data in the world has been growing at a rate that defies human comprehension. The world's data storage capacity has doubled about every three years since the 1980s. During the time it will take you to read this chapter, approximately  $10^{18}$  bits of data will be added to the world's store. It's not easy to relate to a number that large. One way to think about it is that  $10^{18}$  Canadian pennies would have a surface area roughly twice that of the earth.

Of course, more data does not always lead to more useful information. Evolution is a slow process, and the ability of the human mind to assimilate data does not, alas, double every three years. One approach that the world is using to attempt to wring more useful information from “big data” is **statistical machine learning**.

Machine learning is hard to define. In some sense, every useful program learns something. For example, an implementation of Newton's method learns the roots of a polynomial. One of the earliest definitions was proposed by the American electrical engineer and computer scientist Arthur Samuel,<sup>181</sup> who defined it as a “field of study that gives computers the ability to learn without being explicitly programmed.”

Humans learn in two ways—memorization and generalization. We use memorization to accumulate individual facts. In England, for example, primary school students might learn a list of English monarchs. Humans use **generalization** to deduce new facts from old facts. A student of political science, for example, might observe the behavior of a large number of politicians, and generalize from those observations to conclude that all politicians lie when campaigning.

When computer scientists speak about machine learning, they most often mean the discipline of writing programs that automatically learn to make useful inferences from implicit patterns in data. For example, linear regression (see Chapter 20) learns a curve that is a model of a collection of examples. That model can then be used to make predictions about previously unseen examples. The basic paradigm is

1. Observe a set of examples, frequently called the **training data**, that represents incomplete information about some statistical phenomenon.
2. Use inference techniques to create a model of a process that could have generated the observed examples.
3. Use that model to make predictions about previously unseen examples.

Suppose, for example, you were given the two sets of names in [Figure 24-1](#) and the **feature vectors** in [Figure 24-2](#).

```
A: {Abraham Lincoln, George Washington, Charles de Gaulle}  
B: {Benjamin Harrison, James Madison, Louis Napoleon}
```

[Figure 24-1](#) Two sets of names

```
Abraham Lincoln: [American, President, 193 cm tall]  
George Washington: [American, President, 189 cm tall]  
Charles de Gaulle: [French, President, 196 cm tall]  
Benjamin Harrison: [American, President, 168 cm tall]  
James Madison: [American, President, 163 cm tall]  
Louis Napoleon: [French, President, 169 cm tall]
```

[Figure 24-2](#) Associating a feature vector with each name

Each element of a vector corresponds to some aspect (i.e., feature) of the person. Based on this limited information about these historical figures, you might infer that the process assigning either the label A

or the label  $B$  to each example was intended to separate tall presidents from shorter ones.

There are many approaches to machine learning, but all try to learn a model that is a generalization of the provided examples. All have three components:

- A representation of the model
- An objective function for assessing the goodness of the model
- An optimization method for learning a model that minimizes or maximizes the value of the objective function

Broadly speaking, machine learning algorithms can be thought of as either supervised or unsupervised.

In **supervised learning**, we start with a set of feature vector/value pairs. The goal is to derive from these pairs a rule that predicts the value associated with a previously unseen feature vector. **Regression models** associate a real number with each feature vector. **Classification models** associate one of a finite number of **labels** with each feature vector.<sup>[182](#)</sup>

In Chapter 20, we looked at one kind of regression model, linear regression. Each feature vector was an  $x$ -coordinate, and the value associated with it was the corresponding  $y$ -coordinate. From the set of feature vector/value pairs we learned a model that could be used to predict the  $y$ -coordinate associated with any  $x$ -coordinate.

Now, let's look at a simple classification model. Given the sets of presidents we labeled  $A$  and  $B$  in [Figure 24-1](#) and the feature vectors in [Figure 24-2](#), we can generate the feature vector/label pairs in [Figure 24-3](#).

```
[American, President, 193 cm tall], A  
[American, President, 189 cm tall], A  
[French, President, 196 cm tall], A  
[American, President, 168 cm tall], B  
[American, President, 163 cm tall], B  
[French, President, 169 cm tall], B
```

[Figure 24-3](#) Feature vector/label pairs for presidents

From these labeled examples, a learning algorithm might infer that all tall presidents should be labeled  $A$  and all short presidents labeled  $B$ . When asked to assign a label to

[American, President, 189 cm.]<sup>183</sup>

it would use the rule it had learned to choose label  $A$ .

Supervised machine learning is broadly used for such tasks as detecting fraudulent use of credit cards and recommending movies to people.

In **unsupervised learning**, we are given a set of feature vectors but no labels. The goal of unsupervised learning is to uncover latent structure in the set of feature vectors. For example, given the set of presidential feature vectors, an unsupervised learning algorithm might separate the presidents into tall and short, or perhaps into American and French. Approaches to unsupervised machine learning can be categorized as either methods for clustering or methods for learning latent variable models.

A **latent variable** is a variable whose value is not directly observed but can be inferred from the values of variables that are observed. Admissions officers at universities, for example, try to infer the probability of an applicant being a successful student (the latent variable), based on a set of observable values such as secondary school grades and performance on standardized tests. There is a rich set of methods for learning latent variable models, but we do not cover them in this book.

**Clustering** partitions a set of examples into groups (called clusters) such that examples in the same group are more similar to each other than they are to examples in other groups. Geneticists, for example, use clustering to find groups of related genes. Many popular clustering methods are surprisingly simple.

We present a widely used clustering algorithm in Chapter 25, and several approaches to supervised learning in Chapter 26. In the remainder of this chapter, we discuss the process of building feature vectors and different ways of calculating the similarity between two feature vectors.

---

## 24.1 Feature Vectors

The concept of **signal-to-noise ratio (SNR)** is used in many branches of engineering and science. The precise definition varies across applications, but the basic idea is simple. Think of it as the ratio of useful input to irrelevant input. In a restaurant, the signal might be the voice of your dinner date, and the noise the voices of the other diners.<sup>184</sup> If we were trying to predict which students would do well in a programming course, previous programming experience and mathematical aptitude would be part of the signal, but hair color merely noise. Separating the signal from the noise is not always easy. When it is done poorly, the noise can be a distraction that obscures the truth in the signal.

The purpose of **feature engineering** is to separate those features in the available data that contribute to the signal from those that are merely noise. Failure to do an adequate job of this can lead to a bad model. The danger is particularly high when the **dimensionality** of the data (i.e., the number of different features) is large relative to the number of samples.

Successful feature engineering reduces the vast amount of information that might be available to information from which it will be productive to generalize. Imagine, for example, that your goal is to learn a model that will predict whether a person is likely to suffer a heart attack. Some features, such as their age, are likely to be highly relevant. Other features, such as whether they are left-handed, are less likely to be relevant.

**Feature selection** techniques can be used to automatically identify which features in a given set of features are most likely to be helpful. For example, in the context of supervised learning, we can select those features that are most strongly correlated with the labels of the examples.<sup>185</sup> However, these feature selection techniques are of little help if relevant features are not there to start with. Suppose that our original feature set for the heart attack example includes height and weight. It might be the case that while neither height nor weight is highly predictive of a heart attack, body mass index (BMI) is. While BMI can be computed from height and weight, the relationship (weight in kilograms divided by the square of height in meters) is too complicated to be automatically found by typical

machine learning techniques. Successful machine learning often involves the design of features by those with domain expertise.

In unsupervised learning, the problem is even harder. Typically, we choose features based upon our intuition about which features might be relevant to the kinds of structure we would like to find. However, relying on intuition about the potential relevance of features is problematic. How good is your intuition about whether someone's dental history is a useful predictor of a future heart attack?

Consider [Figure 24-4](#), which contains a table of feature vectors and the label (reptile or not) with which each vector is associated.

Name	Egg-laying	Scales	Poisonous	Cold-blooded	# Legs	Reptile
Cobra	True	True	True	True	0	Yes
Rattlesnake	True	True	True	True	0	Yes
Boa constrictor	False	True	False	True	0	Yes
Alligator	True	True	False	True	4	Yes
Dart frog	True	False	True	False	4	No
Salmon	True	True	False	True	0	No
Python	True	True	False	True	0	Yes

[Figure 24-4](#). Name, features, and labels for assorted animals

A supervised machine learning algorithm (or a human) given only the information about cobras—i.e., only the first row of the table—cannot do much more than to remember the fact that a cobra is a reptile. Now, let's add the information about rattlesnakes. We can begin to generalize and might infer the rule that an animal is a reptile if it lays eggs, has scales, is poisonous, is cold-blooded, and has no legs.

Now, suppose we are asked to decide if a boa constrictor is a reptile. We might answer “no,” because a boa constrictor is neither poisonous nor egg-laying. But this would be the wrong answer. Of course, it is hardly surprising that attempting to generalize from two examples might lead us astray. Once we include the boa constrictor in our training data, we might formulate the new rule that an animal

is a reptile if it has scales, is cold-blooded, and is legless. In doing so, we are discarding the features `egg-laying` and `poisonous` as irrelevant to the classification problem.

If we use the new rule to classify the alligator, we conclude incorrectly that since it has legs it is not a reptile. Once we include the alligator in the training data, we reformulate the rule to allow reptiles to have either none or four legs. When we look at the dart frog, we correctly conclude that it is not a reptile, since it is not cold-blooded. However, when we use our current rule to classify the salmon, we incorrectly conclude that a salmon is a reptile. We can add yet more complexity to our rule to separate salmon from alligators, but it's a losing battle. There is no way to modify our rule so that it will correctly classify both salmon and pythons, since the feature vectors of these two species are identical.

This kind of problem is more common than not in machine learning. It is rare to have feature vectors that contain enough information to classify things perfectly. In this case, the problem is that we don't have enough features.

If we had included the fact that reptile eggs have amnios,<sup>186</sup> we could devise a rule that separates reptiles from fish. Unfortunately, in most practical applications of machine learning it is not possible to construct feature vectors that allow for perfect discrimination.

Does this mean that we should give up because all of the available features are mere noise? No. In this case, the features `scales` and `cold-blooded` are necessary conditions for being a reptile, but not sufficient conditions. The rule that an animal is a reptile if it has scales and is cold-blooded will not yield any false negatives, i.e., any animal classified as a non-reptile will indeed not be a reptile. However, the rule will yield some false positives, i.e., some of the animals classified as reptiles will not be reptiles.

---

## 24.2 Distance Metrics

In [Figure 24-4](#) we described animals using four binary features and one integer feature. Suppose we want to use these features to evaluate the similarity of two animals, for example, to ask whether a rattlesnake is more similar to a boa constrictor or to a dart frog.<sup>187</sup>

The first step in doing this kind of comparison is converting the features for each animal into a sequence of numbers. If we say `True` = 1 and `False` = 0, we get the following feature vectors:

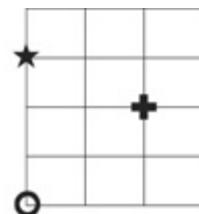
```
Rattlesnake: [1,1,1,1,0]
Boa constrictor: [0,1,0,1,0]
Dart frog: [1,0,1,0,4]
```

There are many ways to compare the similarity of vectors of numbers. The most commonly used metrics for comparing equal-length vectors are based on the **Minkowski distance**:[188](#)

$$\text{distance}(V, W, p) = \left( \sum_{i=1}^{\text{len}} \text{abs}(V_i - W_i)^p \right)^{1/p}$$

where *len* is the length of the vectors.

The parameter *p*, which must be at least 1, defines the kinds of paths that can be followed in traversing the distance between the vectors *V* and *W*.[189](#) This can be easily visualized if the vectors are of length two, and can therefore be represented using Cartesian coordinates. Consider the picture in [Figure 24-5](#).



[Figure 24-5](#) Visualizing distance metrics

Is the circle in the bottom-left corner closer to the cross or closer to the star? It depends. If we can travel in a straight line, the cross is closer. The Pythagorean Theorem tells us that the cross is the square root of 8 units from the circle, about 2.8 units, whereas we can easily see that the star is 3 units from the circle. These distances are called **Euclidean distances**, and correspond to using the Minkowski

distance with  $p = 2$ . But imagine that the lines in the picture correspond to streets, and that we have to stay on the streets to get from one place to another. The star remains 3 units from the circle, but the cross is now 4 units away. These distances are called **Manhattan distances**,<sup>190</sup> and they correspond to using the Minkowski distance with  $p = 1$ . [Figure 24-6](#) contains a function implementing the Minkowski distance.

```
def minkowski_dist(v1, v2, p):
    """Assumes v1 and v2 are equal-length arrays of numbers
       Returns Minkowski distance of order p between v1 and v2"""
    dist = 0.0
    for i in range(len(v1)):
        dist += abs(v1[i] - v2[i])**p
    return dist**(1/p)
```

[Figure 24-6](#) Minkowski distance

[Figure 24-7](#) contains class `Animal`. It defines the distance between two animals as the Euclidean distance between the feature vectors associated with the animals.

```

class Animal(object):
    def __init__(self, name, features):
        """Assumes name a string; features a list of numbers"""
        self.name = name
        self.features = np.array(features)

    def get_name(self):
        return self.name

    def get_features(self):
        return self.features

    def distance(self, other):
        """Assumes other is an Animal
           Returns the Euclidean distance between feature vectors
           of self and other"""
        return minkowski_dist(self.get_features(),
                              other.get_features(), 2)

```

[Figure 24-7](#) Class Animal

[Figure 24-8](#) contains a function that compares a list of animals to each other and produces a table showing the pairwise distances. The code uses a Matplotlib plotting facility that we have not previously used: `table`.

The `table` function produces a plot that (surprise!) looks like a table. The keyword arguments `rowLabels` and `colLabels` are used to supply the labels (in this example the names of the animals) for the rows and columns. The keyword argument `cellText` is used to supply the values appearing in the cells of the table. In the example, `cellText` is bound to `table_vals`, which is a list of lists of strings. Each element in `table_vals` is a list of the values for the cells in one row of the table. The keyword argument `cellLoc` is used to specify where in each cell the text should appear, and the keyword argument `loc` is used to specify where in the figure the table itself should appear. The last keyword parameter used in the example is `colWidths`. It is bound to a list of floats giving the width (in inches) of each column in the table. The code `table.scale(1, 2.5)` instructs Matplotlib to leave the horizontal width of the cells unchanged, but to increase the height of the cells by a factor of 2.5 (so the tables look prettier).

```

def compare_animals(animals, precision):
    """Assumes animals is a list of animals, precision an int >= 0
       Builds a table of distances between each animal"""
    # Get labels for columns and rows
    column_labels = [a.get_name() for a in animals]
    row_labels = column_labels[:]
    table_vals = []
    # Get distances between pairs of animals
    # For each row
    for a1 in animals:
        row = []
        #For each column
        for a2 in animals:
            distance = a1.distance(a2)
            row.append(str(round(distance, precision)))
        table_vals.append(row)
    # Produce table
    table = plt.table(rowLabels=row_labels,
                       colLabels=column_labels,
                       cellText=table_vals,
                       cellLoc='center',
                       loc='center',
                       colWidths=[0.2]*len(animals))
    plt.axis('off')
    table.scale(1, 2.5)

```

[Figure 24-8](#) Build table of distances between pairs of animals

If we run the code

```

rattlesnake = Animal('rattlesnake', [1,1,1,1,0])
boa = Animal('boa', [0,1,0,1,0])
dart_frog = Animal('dart frog', [1,0,1,0,4])
animals = [rattlesnake, boa, dart_frog]
compare_animals(animals, 3)

```

it produces the table in [Figure 24-9](#).

As you probably expected, the distance between the rattlesnake and the boa constrictor is less than that between either of the snakes and the dart frog. Notice, by the way, that the dart frog is a bit closer to the rattlesnake than to the boa constrictor.

	rattlesnake	boa	dart frog
rattlesnake	0.0	1.414	4.243
boa	1.414	0.0	4.472
dart frog	4.243	4.472	0.0

[Figure 24-9](#). Distances between three animals

Now, let's insert before the last line of the above code the lines

```
alligator = Animal('alligator', [1,1,0,1,4])
animals.append(alligator)
```

It produces the table in [Figure 24-10](#).

	rattlesnake	boa	dart frog	alligator
rattlesnake	0.0	1.414	4.243	4.123
boa	1.414	0.0	4.472	4.123
dart frog	4.243	4.472	0.0	1.732
alligator	4.123	4.123	1.732	0.0

[Figure 24-10](#). Distances between four animals

Perhaps you're surprised that the alligator is considerably closer to the dart frog than to either the rattlesnake or the boa constrictor. Take a minute to think about why.

The feature vector for the alligator differs from that of the rattlesnake in two places: whether it is poisonous and the number of legs. The feature vector for the alligator differs from that of the dart frog in three places: whether it is poisonous, whether it has scales, and whether it is cold-blooded. Yet, according to our Euclidean

distance metric, the alligator is more like the dart frog than like the rattlesnake. What's going on?

The root of the problem is that the different features have different ranges of values. All but one of the features range between 0 and 1, but the number of legs ranges from 0 to 4. This means that when we calculate the Euclidean distance, the number of legs gets disproportionate weight. Let's see what happens if we turn the feature into a binary feature, with a value of 0 if the animal is legless and 1 otherwise.

	rattlesnake	boa	dart frog	alligator
rattlesnake	0.0	1.414	1.732	1.414
boa	1.414	0.0	2.236	1.414
dart frog	1.732	2.236	0.0	1.732
alligator	1.414	1.414	1.732	0.0

Figure 24-11 Distances using a different feature representation

This looks a lot more plausible.

Of course, it is not always convenient to use only binary features. In Section 25.4, we will present a more general approach to dealing with differences in scale among features.

---

## 24.3 Terms Introduced in Chapter

statistical machine learning

generalization

training data

feature vector

supervised learning

regression models  
classification models  
label  
unsupervised learning  
latent variable  
clustering  
signal-to-noise ratio (SNR)  
feature engineering  
dimensionality (of data)  
feature selection  
Minkowski distance  
triangle inequality  
Euclidean distance  
Manhattan distance

---

[181](#) Samuel is probably best known as the author of a program that played checkers. The program, which he started working on in the 1950s and continued to work on into the 1970s, was impressive for its time, though not particularly good by modern standards. However, while working on it Samuel invented several techniques that are still used today. Among other things, Samuel's checker-playing program was possibly the first program ever written that improved based upon “experience.”

[182](#) Much of the machine learning literature uses the word “class” rather than “label.” Since we have used the word “class” for something else in this book, we will stick to using “label” for this concept.

[183](#) In case you are curious, Thomas Jefferson was 189 cm. tall.

184 Unless your dinner date is exceedingly boring. In that case, your dinner date's conversation becomes the noise, and the conversation at the next table the signal.

185 Since features are often strongly correlated with each other, this can lead to a large number of redundant features. There are more sophisticated feature selection techniques, but we do not cover them in this book.

186 Amnios are protective outer layers that allow eggs to be laid on land rather than in the water.

187 This question is not quite as silly as it sounds. A naturalist and a toxicologist (or someone looking to enhance the effectiveness of a blow dart) might give different answers to this question.

188 Another popular distance metric is cosine similarity. This captures the difference in the angle of the two vectors. It is often useful for high-dimensional vectors.

189 When  $p < 1$ , peculiar things happen. Consider, for example  $p = 0.5$  and the points  $A = (0, 0)$ ,  $B = (1, 1)$ , and  $C = (0, 1)$ . If you compute the pairwise distances between these points, you will discover that the distance from  $A$  to  $B$  is 4, the distance from  $A$  to  $C$  is 1, and the distance from  $C$  to  $B$  is 1. Common sense dictates that the distance from  $A$  to  $B$  via  $C$  cannot be less than the distance from  $A$  to  $B$ . (Mathematicians refer to this as the **triangle inequality**, which states that for any triangle the sum of the lengths of any two sides must not be less than the length of the third side.)

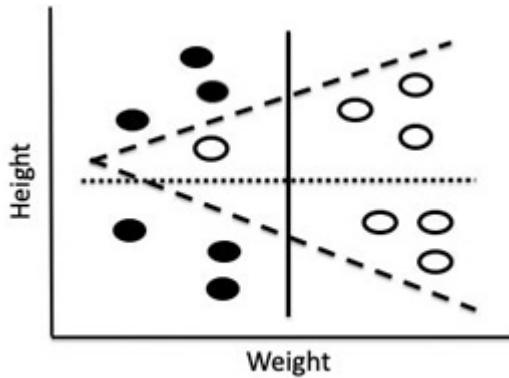
190 Manhattan Island is the most densely populated borough of New York City. On most of the island, the streets are laid out in a rectangular grid, so using the Minkowski distance with  $p = 1$  provides a good approximation of the distance pedestrians travel walking from one place to another. Driving or taking public transit in Manhattan is a totally different story.

# 25

## CLUSTERING

Unsupervised learning involves finding hidden structure in unlabeled data. The most commonly used unsupervised machine learning technique is clustering.

Clustering can be defined as the process of organizing objects into groups whose members are similar in some way. A key issue is defining the meaning of “similar.” Consider the plot in [Figure 25-1](#), which shows the height, weight, and shirt color for 13 people.



[Figure 25-1](#) Height, weight, and shirt color

If we cluster people by height, there are two obvious clusters—delimited by the dotted horizontal line. If we cluster people by weight, there are two different obvious clusters—delimited by the solid vertical line. If we cluster people based on their shirts, there is yet a third clustering—delimited by the angled dashed lines. Notice, by the way, that this last division is not linear since we cannot separate the people by shirt color using a single straight line.

Clustering is an optimization problem. The goal is to find a set of clusters that optimizes an objective function, subject to some set of constraints. Given a distance metric that can be used to decide how close two examples are to each other, we need to define an objective function that minimizes the dissimilarity of the examples within a cluster.

One measure, which we call variability (often called inertia in the literature), of how different the examples within a single cluster,  $c$ , are from each other is

$$\text{variability}(c) = \sum_{e \in c} \text{distance}(\text{mean}(c), e)^2$$

where  $\text{mean}(c)$  is the mean of the feature vectors of all the examples in the cluster. The mean of a set of vectors is computed component-wise. The corresponding elements are added, and the result divided by the number of vectors. If  $v_1$  and  $v_2$  are arrays of numbers, the value of the expression  $(v_1 + v_2) / 2$  is their **Euclidean mean**.

What we are calling variability is similar to the notion of variance presented in Chapter 17. The difference is that variability is not normalized by the size of the cluster, so clusters with more points are likely to look less cohesive according to this measure. If we want to compare the coherence of two clusters of different sizes, we need to divide the variability of each cluster by the size of the cluster.

The definition of variability within a single cluster,  $c$ , can be extended to define a dissimilarity metric for a set of clusters,  $C$ :

$$\text{dissimilarity}(C) = \sum_{c \in C} \text{variability}(c)$$

Notice that since we don't divide the variability by the size of the cluster, a large incoherent cluster increases the value of  $\text{dissimilarity}(C)$  more than a small incoherent cluster does. This is by design.

So, is the optimization problem to find a set of clusters,  $C$ , such that  $dissimilarity(C)$  is minimized? Not exactly. It can easily be minimized by putting each example in its own cluster. We need to add some constraint. For example, we could put a constraint on the minimum distance between clusters or require that the maximum number of clusters be some constant  $k$ .

In general, solving this optimization problem is computationally prohibitive for most interesting problems. Consequently, people rely on greedy algorithms that provide approximate solutions. In Section 25.2, we present one such algorithm, k-means clustering. But first we will introduce some abstractions that are useful for implementing that algorithm (and other clustering algorithms as well).

---

## 25.1 Class Cluster

Class `Example` ([Figure 25-2](#)) will be used to build the samples to be clustered. Associated with each example is a name, a feature vector, and an optional label. The `distance` method returns the Euclidean distance between two examples.

```

class Example(object):

    def __init__(self, name, features, label = None):
        #Assumes features is an array of floats
        self.name = name
        self.features = features
        self.label = label

    def dimensionality(self):
        return len(self.features)

    def set_label(self, label):
        self.label = label

    def get_features(self):
        return self.features[:]

    def get_label(self):
        return self.label

    def get_name(self):
        return self.name

    def distance(self, other):
        return minkowski_dist(self.features, other.get_features(), 2)

    def __str__(self):
        return '{}:{}:{}'.format(self.name, self.features, self.label)

```

[Figure 25-2](#) Class Example

Class `Cluster` ([Figure 25-3](#)) is slightly more complex. A cluster is a set of examples. The two interesting methods in `cluster` are `compute_centroid` and `variability`. Think of the **centroid** of a cluster as its center of mass. The method `compute_centroid` returns an example with a feature vector equal to the Euclidean mean of the feature vectors of the examples in the cluster. The method `variability` provides a measure of the coherence of the cluster.

```

class Cluster(object):
    def __init__(self, examples):
        """Assumes examples a non-empty list of Examples"""
        self.examples = examples
        self.centroid = self.compute_centroid()

    def update(self, examples):
        """Assume examples is a non-empty list of Examples
           Replace examples; return amount centroid has changed"""
        old_centroid = self.centroid
        self.examples = examples
        self.centroid = self.compute_centroid()
        return old_centroid.distance(self.centroid)

    def compute_centroid(self):
        vals = np.array([0.0]*self.examples[0].dimensionality())
        for e in self.examples: #compute mean
            vals += e.get_features()
        centroid = Example('centroid', vals/len(self.examples))
        return centroid

    def get_centroid(self):
        return self.centroid

    def variability(self):
        tot_dist = 0.0
        for e in self.examples:
            tot_dist += (e.distance(self.centroid))**2
        return tot_dist

    def members(self):
        for e in self.examples:
            yield e

    def __str__(self):
        names = []
        for e in self.examples:
            names.append(e.get_name())
        names.sort()
        result = ('Cluster with centroid '
                  + str(self.centroid.get_features()) + ' contains:\n  ')
        for e in names:
            result = result + e + ', '
        return result[:-2] #remove trailing comma and space

```

[Figure 25-3](#) Class Cluster

**Finger exercise:** Is the centroid of a cluster always one of the examples in the cluster?

---

## 25.2 K-means Clustering

**K-means clustering** is probably the most widely used clustering method.<sup>191</sup> Its goal is to partition a set of examples into  $k$  clusters such that

- Each example is in the cluster whose centroid is the closest centroid to that example.
- The dissimilarity of the set of clusters is minimized.

Unfortunately, finding an optimal solution to this problem on a large data set is computationally intractable. Fortunately, there is an efficient greedy algorithm<sup>192</sup> that can be used to find a useful approximation. It is described by the pseudocode

```
randomly choose k examples as initial centroids of clusters
while true:
    1. Create k clusters by assigning each example to closest
       centroid
    2. Compute k new centroids by averaging the examples in each
       cluster
    3. If none of the centroids differ from the previous
       iteration:
        return the current set of clusters
```

The complexity of step 1 is order  $\theta(k \cdot n \cdot d)$ , where  $k$  is the number of clusters,  $n$  is the number of examples, and  $d$  the time required to compute the distance between a pair of examples. The complexity of step 2 is  $\theta(n)$ , and the complexity of step 3 is  $\theta(k)$ . Hence, the complexity of a single iteration is  $\theta(k \cdot n \cdot d)$ . If the examples are compared using the Minkowski distance,  $d$  is linear in the length of the feature vector.<sup>193</sup> Of course, the complexity of the entire algorithm depends upon the number of iterations. That is not easy to characterize, but suffice it to say that it is usually small.

[Figure 25-4](#) contains a translation into Python of the pseudocode describing k-means. The only wrinkle is that it raises an exception if any iteration creates a cluster with no members. Generating an empty cluster is rare. It can't occur on the first iteration, but it can occur on subsequent iterations. It usually results from choosing too

large a  $k$  or an unlucky choice of initial centroids. Treating an empty cluster as an error is one of the options used by MATLAB. Another is creating a new cluster containing a single point—the point furthest from the centroid in the other clusters. We chose to treat it as an error to simplify the implementation.

One problem with the k-means algorithm is that the value returned depends upon the initial set of randomly chosen centroids. If a particularly unfortunate set of initial centroids is chosen, the algorithm might settle into a local optimum that is far from the global optimum. In practice, this problem is typically addressed by running k-means multiple times with randomly chosen initial centroids. We then choose the solution with the minimum dissimilarity of clusters.

[Figure 25-5](#) contains a function, `try_k_means`, that calls `k_means` ([Figure 25-4](#)) multiple times and selects the result with the lowest dissimilarity. If a trial fails because `k_means` generated an empty cluster and therefore raised an exception, `try_k_means` merely tries again—assuming that eventually `k_means` will choose an initial set of centroids that successfully converges.

```

def k_means(examples, k, verbose = False):
    #Get k randomly chosen initial centroids, create cluster for each
    initial_centroids = random.sample(examples, k)
    clusters = []
    for e in initial_centroids:
        clusters.append(Cluster([e]))

    #Iterate until centroids do not change
    converged = False
    num_iterations = 0
    while not converged:
        num_iterations += 1
        #Create a list containing k distinct empty lists
        new_clusters = []
        for i in range(k):
            new_clusters.append([])

        #Associate each example with closest centroid
        for e in examples:
            #Find the centroid closest to e
            smallest_distance = e.distance(clusters[0].get_centroid())
            index = 0
            for i in range(1, k):
                distance = e.distance(clusters[i].get_centroid())
                if distance < smallest_distance:
                    smallest_distance = distance
                    index = i
            #Add e to the list of examples for appropriate cluster
            new_clusters[index].append(e)

        for c in new_clusters: #Avoid having empty clusters
            if len(c) == 0:
                raise ValueError('Empty Cluster')

        #Update each cluster; check if a centroid has changed
        converged = True
        for i in range(k):
            if clusters[i].update(new_clusters[i]) > 0.0:
                converged = False
        if verbose:
            print('Iteration #' + str(num_iterations))
            for c in clusters:
                print(c)
            print('') #add blank line
    return clusters

```

[Figure 25-4.](#) K-means clustering

```

def dissimilarity(clusters):
    tot_dist = 0.0
    for c in clusters:
        tot_dist += c.variability()
    return tot_dist

def try_k_means(examples, num_clusters, num_trials, verbose = False):
    """Calls k_means num_trials times and returns the result with the
       lowest dissimilarity"""
    best = k_means(examples, num_clusters, verbose)
    min_dissimilarity = dissimilarity(best)
    trial = 1
    while trial < num_trials:
        try:
            clusters = k_means(examples, num_clusters, verbose)
        except ValueError:
            continue #If failed, try again
        curr_dissimilarity = dissimilarity(clusters)
        if curr_dissimilarity < min_dissimilarity:
            best = clusters
            min_dissimilarity = curr_dissimilarity
        trial += 1
    return best

```

[Figure 25-5](#) Finding the best k-means clustering

---

## 25.3 A Contrived Example

[Figure 25-6](#) contains code that generates, plots, and clusters examples drawn from two distributions.

The function `gen_distributions` generates a list of `n` examples with two-dimensional feature vectors. The values of the elements of these feature vectors are drawn from normal distributions.

The function `plot_samples` plots the feature vectors of a set of examples. It uses `plt.annotate` to place text next to points on the plot. The first argument is the text, the second argument the point with which the text is associated, and the third argument the location of the text relative to the point with which it is associated.

The function `contrived_test` uses `gen_distributions` to create two distributions of 10 examples (each with the same standard deviation but different means), plots the examples using `plot_samples`, and then clusters them using `try_k_means`.

```

def gen_distribution(x_mean, x_sd, y_mean, y_sd, n, name_prefix):
    samples = []
    for s in range(n):
        x = random.gauss(x_mean, x_sd)
        y = random.gauss(y_mean, y_sd)
        samples.append(Example(name_prefix+str(s), [x, y]))
    return samples

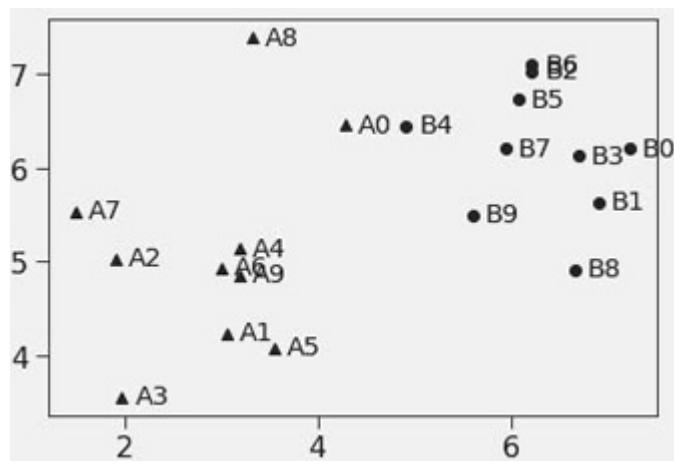
def plot_samples(samples, marker):
    x_vals, y_vals = [], []
    for s in samples:
        x = s.get_features()[0]
        y = s.get_features()[1]
        plt.annotate(s.get_name(), xy = (x, y),
                     xytext = (x+0.13, y-0.07),
                     fontsize = 'x-large')
        x_vals.append(x)
        y_vals.append(y)
    plt.plot(x_vals, y_vals, marker)

def contrived_test(num_trials, k, verbose = False):
    x_mean = 3
    x_sd = 1
    y_mean = 5
    y_sd = 1
    n = 10
    d1_samples = gen_distribution(x_mean, x_sd, y_mean, y_sd, n, 'A')
    plot_samples(d1_samples, 'k^')
    d2_samples = gen_distribution(x_mean+3, x_sd, y_mean+1,
                                  y_sd, n, 'B')
    plot_samples(d2_samples, 'ko')
    clusters = try_k_means(d1_samples+d2_samples, k, num_trials,
                           verbose)
    print('Final result')
    for c in clusters:
        print('', c)

```

[Figure 25-6](#) A test of k-means

The call `contrived_test(1, 2, True)` produced the plot in [Figure 25-7](#) and printed the lines in [Figure 25-8](#). Notice that the initial (randomly chosen) centroids led to a highly skewed clustering in which a single cluster contained all but one of the points. By the fourth iteration, however, the centroids had moved to places such that the points from the two distributions were reasonably well separated into two clusters. The only “mistakes” were made on `A0` and `A8`.



[Figure 25-7](#) Examples from two distributions

```

Iteration #1
Cluster with centroid [4.71113345 5.76359152] contains:
A0, A1, A2, A4, A5, A6, A7, A8, A9, B0, B1, B2, B3, B4, B5, B6,
B7, B8, B9
Cluster with centroid [1.97789683 3.56317055] contains:
A3

Iteration #2
Cluster with centroid [5.46369488 6.12015454] contains:
A0, A4, A8, A9, B0, B1, B2, B3, B4, B5, B6, B7, B8, B9
Cluster with centroid [2.49961733 4.56487432] contains:
A1, A2, A3, A5, A6, A7

Iteration #3
Cluster with centroid [5.84078727 6.30779094] contains:
A0, A8, B0, B1, B2, B3, B4, B5, B6, B7, B8, B9
Cluster with centroid [2.67499815 4.67223977] contains:
A1, A2, A3, A4, A5, A6, A7, A9

Iteration #4
Cluster with centroid [5.84078727 6.30779094] contains:
A0, A8, B0, B1, B2, B3, B4, B5, B6, B7, B8, B9
Cluster with centroid [2.67499815 4.67223977] contains:
A1, A2, A3, A4, A5, A6, A7, A9

Final result
Cluster with centroid [5.84078727 6.30779094] contains:
A0, A8, B0, B1, B2, B3, B4, B5, B6, B7, B8, B9
Cluster with centroid [2.67499815 4.67223977] contains:
A1, A2, A3, A4, A5, A6, A7, A9

```

[Figure 25-8](#) Lines printed by a call to contrived\_test(1, 2, True)

When we tried 50 trials rather than 1, by calling `contrived_test(50, 2, False)`, it printed

```
Final result
Cluster with centroid [2.74674403 4.97411447] contains:
A1, A2, A3, A4, A5, A6, A7, A8, A9
Cluster with centroid [6.0698851 6.20948902] contains:
A0, B0, B1, B2, B3, B4, B5, B6, B7, B8, B9
```

`A0` is still mixed in with the `B`'s, but `A8` is not. If we try 1000 trials, we get the same result. That might surprise you, since a glance at [Figure 25-7](#) reveals that if `A0` and `B0` are chosen as the initial centroids (which would probably happen with 1000 trials), the first iteration will yield clusters that perfectly separate the `A`'s and `B`'s. However, in the second iteration new centroids will be computed, and `A0` will be assigned to a cluster with the `B`'s. Is this bad? Recall that clustering is a form of unsupervised learning that looks for structure in unlabeled data. Grouping `A0` with the `B`'s is not unreasonable.

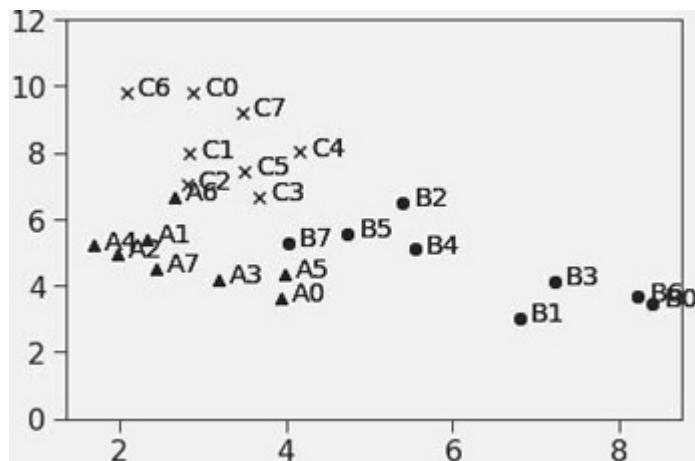
One of the key issues in using k-means clustering is choosing `k`. The function `contrived_test_2` in [Figure 25-9](#) generates, plots, and clusters points from three overlapping Gaussian distributions. We will use it to look at the results of clustering this data for various values of `k`. The data points are shown in [Figure 25-10](#).

```

def contrived_test2(num_trials, k, verbose = False):
    x_mean = 3
    x_sd = 1
    y_mean = 5
    y_sd = 1
    n = 8
    d1_samples = gen_distribution(x_mean,x_sd, y_mean, y_sd, n, 'A')
    plot_samples(d1_samples, 'k^')
    d2_samples = gen_distribution(x_mean+3,x_sd,y_mean, y_sd, n, 'B')
    plot_samples(d2_samples, 'ko')
    d3Samples = gen_distribution(x_mean, x_sd, y_mean+3, y_sd, n, 'C')
    plot_samples(d3Samples, 'kx')
    clusters = try_k_means(d1_samples + d2_samples + d3Samples,
                           k, num_trials, verbose)
    plt.ylim(0, 12)
    print('Final result has dissimilarity',
          round(dissimilarity(clusters), 3))
    for c in clusters:
        print('', c)

```

[Figure 25-9](#) Generating points from three distributions



[Figure 25-10](#) Points from three overlapping Gaussians

The invocation `contrived_test2(40, 2)` prints

```

Final result has dissimilarity 90.128
Cluster with centroid [5.5884966 4.43260236] contains:
A0, A3, A5, B0, B1, B2, B3, B4, B5, B6, B7
Cluster with centroid [2.80949911 7.11735738] contains:
A1, A2, A4, A6, A7, C0, C1, C2, C3, C4, C5, C6, C7

```

The invocation `contrived_test2(40, 3)` prints

```
Final result has dissimilarity 42.757
Cluster with centroid [7.66239972 3.55222681] contains:
B0, B1, B3, B6
Cluster with centroid [3.56907939 4.95707576] contains:
A0, A1, A2, A3, A4, A5, A7, B2, B4, B5, B7
Cluster with centroid [3.12083099 8.06083681] contains:
A6, C0, C1, C2, C3, C4, C5, C6, C7
```

And the invocation `contrived_test2(40, 6)` prints

```
Final result has dissimilarity 11.441
Cluster with centroid [2.10900238 4.99452866] contains:
A1, A2, A4, A7
Cluster with centroid [4.92742554 5.60609442] contains:
B2, B4, B5, B7
Cluster with centroid [2.80974427 9.60386549] contains:
C0, C6, C7
Cluster with centroid [3.27637435 7.28932247] contains:
A6, C1, C2, C3, C4, C5
Cluster with centroid [3.70472053 4.04178035] contains:
A0, A3, A5
Cluster with centroid [7.66239972 3.55222681] contains:
B0, B1, B3, B6
```

The last clustering is the tightest fit, i.e., the clustering has the lowest dissimilarity (11.441). Does this mean that it is the “best” clustering? Not necessarily. Recall that when we looked at linear regression in Section 20.1.1, we observed that by increasing the degree of the polynomial we got a more complex model that provided a tighter fit to the data. We also observed that when we increased the degree of the polynomial, we ran the risk of finding a model with poor predictive value—because it overfit the data.

Choosing the right value for  $k$  is exactly analogous to choosing the right degree polynomial for a linear regression. By increasing  $k$ , we can decrease dissimilarity, at the risk of overfitting. (When  $k$  is equal to the number of examples to be clustered, the dissimilarity is 0!) If we have information about how the examples to be clustered were generated, e.g., chosen from  $m$  distributions, we can use that information to choose  $k$ . Absent such information, there are a variety of heuristic procedures for choosing  $k$ . Going into them is beyond the scope of this book.

---

## 25.4 A Less Contrived Example

Different species of mammals have different eating habits. Some species (e.g., elephants and beavers) eat only plants, others (e.g., lions and tigers) eat only meat, and some (e.g., pigs and humans) eat anything they can get into their mouths. The vegetarian species are called herbivores, the meat eaters are called carnivores, and those species that eat both plants and animals are called omnivores.

Over the millennia, evolution (or, if you prefer, some other mysterious process) has equipped species with teeth suitable for consumption of their preferred foods.<sup>194</sup> That raises the question of whether clustering mammals based on their dentition produces clusters that have some relation to their diets.

[Figure 25-11](#) shows the contents of a file listing some species of mammals, their dental formulas (the first 8 numbers), and their average adult weight in pounds.<sup>195</sup> The comments at the top describe the items associated with each mammal, e.g., the first item following the name is the number of top incisors.

```

#Meaning of columns given by next line (t for top and b or bottom)
#Name,t incisors,t canines,t premolars,t molars,b incisors,b canines,
#b premolars,b molars,weight
Name,ti,tc,tpm,tm,bi,bc,bpm,bm,weight
Badger,3,1,3,1,3,1,3,2,10
Bear,3,1,4,2,3,1,4,3,278
Cougar,3,1,3,1,3,1,2,1,63
Cow,0,0,3,3,3,1,2,1,400
Deer,0,0,3,3,4,0,3,3,200
Dog,3,1,4,2,3,1,4,3,20
Elk,0,1,3,3,3,1,3,3,500
Fox,3,1,4,2,3,1,4,3,5
Fur seal,3,1,4,1,2,1,4,1,200
Grey seal,3,1,3,2,2,1,3,2,268
Guinea pig,1,0,1,3,1,0,1,3,1
Human,2,1,2,3,2,1,2,3,150
Jaguar,3,1,3,1,3,1,2,1,81
Kangaroo,3,1,2,4,1,0,2,4,55
Lion,3,1,3,1,3,1,2,1,175
Mink,3,1,3,1,3,1,3,2,1
Mole,3,1,4,3,3,1,4,3,0.75
Moose,0,0,3,3,4,0,3,3,900
Mouse,1,0,0,3,1,0,0,3,0.3
Pig,3,1,4,3,3,1,4,3,50
Porcupine,1,0,1,3,1,0,1,3,3
Rabbit,2,0,3,3,1,0,2,3,1
Raccoon,3,1,4,2,3,1,4,2,40
Rat,1,0,0,3,1,0,0,3,.75
Red bat,1,1,2,3,3,1,2,3,1
Sea lion,3,1,4,1,2,1,4,1,415
Skunk,3,1,3,1,3,1,3,2,2
Squirrel,1,0,2,3,1,0,1,3,2
Wolf,3,1,4,2,3,1,4,3,27
Woodchuck,1,0,2,3,1,0,1,3,4

```

[Figure 25-11](#) Mammal dentition in `dentalFormulas.csv`

[Figure 25-12](#) contains three functions. The function `read_mammal_data` first reads a CSV file, formatted like the one in [Figure 25-11](#), to create a DataFrame. The keyword argument `comment` is used to instruct `read_csv` to ignore lines starting with `#`. If the parameter `scale_method` is not equal to `None`, it then scales each column in the DataFrame using `scale_method`. Finally, it creates and returns a dictionary mapping species names to feature vectors. The function `build_mammal_examples` uses the dictionary returned by

`read_mammal_data` to produce and return a set of examples. The function `test_teeth` produces and prints a clustering.

```
def read_mammal_data(fName, scale_method = None):
    """fName a CSV file describing dentition of mammals
       returns a dict mapping species to feature vectors
    """
    df = pd.read_csv('dentalFormulas.csv', comment = '#')
    df = df.set_index('Name')
    if scale_method != None:
        for c in df.columns:
            df[c] = scale_method(df[c])
    feature_vector_list = [np.array(df.loc[i].values)
                           for i in df.index]
    species_names = list(df.index)
    return {species_names[i]: feature_vector_list[i]
            for i in range(len(species_names))}

def build_mammal_examples(species_dict):
    examples = []
    for i in species_dict:
        example = Example(i, species_dict[i])
        examples.append(example)
    return examples

def test_teeth(file_name, num_clusters, num_trials,
              scale_method = None):
    def print_clustering(clustering):
        for c in clustering:
            names = ''
            for p in c.members():
                names += p.get_name() + ', '
            print('\n' + names[:-2]) #remove trailing comma and space
    species_dict = read_mammal_data(file_name, scale_method)
    examples = build_mammal_examples(species_dict)
    print_clustering(try_k_means(examples, num_clusters, num_trials))
```

[Figure 25-12](#) Read and process CSV file

The call `test_teeth('dentalFormulas.csv', 3, 40)` prints

Bear, Cow, Deer, Elk, Fur seal, Grey seal, Lion, Sea lion

Badger, Cougar, Dog, Fox, Guinea pig, Human, Jaguar,  
Kangaroo, Mink, Mole, Mouse, Pig, Porcupine, Rabbit,

Raccoon, Rat, Red bat, Skunk, Squirrel, Wolf, Woodchuck

Moose

A cursory inspection suggests that we have a clustering totally dominated by the weights of the animals. The problem is that the range of weights is much larger than the range of any of the other features. Therefore, when the Euclidean distance between examples is computed, the only feature that truly matters is weight.

We encountered a similar problem in Section 24.2 when we found that the distance between animals was dominated by the number of legs. We solved the problem there by turning the number of legs into a binary feature (legged or legless). That was fine for that data set, because all of the animals happened to have either zero or four legs. Here, however, there is no obvious way to turn weight into a single binary feature without losing a great deal of information.

This is a common problem, which is often addressed by scaling the features so that each feature has a mean of 0 and a standard deviation of 1,<sup>196</sup> as done by the function `z_scale` in [Figure 25-13](#). It's easy to see why the statement `result = result - mean` ensures that the mean of the returned array will always be close to 0.<sup>197</sup> That the standard deviation will always be 1 is not obvious. It can be shown by a long and tedious chain of algebraic manipulations, which we will not bore you with. This kind of scaling is often called **z-scaling** because the standard normal distribution is sometimes referred to as the Z-distribution.

Another common approach to scaling is to map the minimum feature value to 0, map the maximum feature value to 1, and use **linear scaling** in between, as done by the function `linear_scale` in [Figure 25-13](#). This is often called **min-max scaling**.

```

def z_scale(vals):
    """Assumes vals is a sequence of floats"""
    result = np.array(vals) - np.array(vals).mean()
    return (result/np.std(result)).round(4)

def linear_scale(vals):
    """Assumes vals is a sequence of floats"""
    vals = np.array(vals)
    vals -= vals.min()
    return (vals/vals.max()).round(4)

```

[Figure 25-13](#) Scaling attributes

The call `test_teeth('dentalFormulas.csv', 3, 40, z_scale)` prints

Badger, Bear, Cougar, Dog, Fox, Fur seal, Grey seal, Human,  
Jaguar, Lion, Mink, Mole, Pig, Raccoon, Red bat, Sea lion,  
Skunk, Wolf

Guinea pig, Kangaroo, Mouse, Porcupine, Rabbit, Rat,  
Squirrel, Woodchuck

Cow, Deer, Elk, Moose

It's not immediately obvious how this clustering relates to the features associated with each of these mammals, but at least it is not merely grouping the mammals by weight.

Recall that we started this section by hypothesizing that there was a relationship between a mammal's dentition and its diet. [Figure 25-14](#) contains an excerpt of a CSV file, `diet.csv`, that associates mammals with their dietary preference.

```
#diet: 0=herbivore, 1=carnivore, 2=omnivore
Name,Diet
Badger,1
Bear,2
Cougar,1
Cow,0
Deer,0
Dog,1
Elk,0
Fox,1
Fur seal,1
Grey seal,1
Guinea pig,0
Human,2
...
Wolf,1
Woodchuck,2
```

[Figure 25-14](#). Start of CSV file classifying mammals by diet

We can use information in `diet.csv` to see to what extent the clusterings we have produced are related to diet. The code in [Figure 25-15](#) does exactly that.

```

def add_labels(examples, label_file):
    df = pd.read_csv(label_file, comment = '#')
    df = df.set_index('Name')
    for e in examples:
        if e.get_name() in df.index:
            e.set_label(df.loc[e.get_name()]['Diet'])

def check_diet(cluster):
    herbivores, carnivores, omnivores = 0, 0, 0
    for m in cluster.members():
        if m.get_label() == 0:
            herbivores += 1
        elif m.get_label() == 1:
            carnivores += 1
        else:
            omnivores += 1
    print(' ', herbivores, 'herbivores,', carnivores, 'carnivores,', omnivores, 'omnivores\n')

def test_teeth_diet(features_file, labels_file, num_clusters,
                     num_trials, scale_method = None):
    def print_clustering(clustering):
        for c in clustering:
            names = ''
            for p in c.members():
                names += p.get_name() + ', '
            print(names[:-2])
            check_diet(c)
    species_dict = read_mammal_data(features_file, scale_method)
    examples = build_mammal_examples(species_dict)
    add_labels(examples, labels_file)
    print_clustering(try_k_means(examples, num_clusters, num_trials))

```

[Figure 25-15](#) Relating clustering to labels

**When `test_teeth_diet('dentalFormulas.csv', 'diet.csv', 3, 40, z_scale)` was run, it printed**

Badger, Bear, Cougar, Dog, Fox, Fur seal, Grey seal, Human,  
Jaguar, Lion, Mink, Mole, Pig, Raccoon, Red bat, Sea lion,  
Skunk, Wolf  
0 herbivores, 13 carnivores, 5 omnivores

Guinea pig, Kangaroo, Mouse, Porcupine, Rabbit, Rat,  
Squirrel, Woodchuck  
3 herbivores, 0 carnivores, 5 omnivores

Cow, Deer, Elk, Moose  
4 herbivores, 0 carnivores, 0 omnivores

The clustering with z-scaling (linear scaling yields the same clusters) does not perfectly partition the animals based upon their eating habits, but it is certainly correlated with what they eat. It does a good job of separating the carnivores from the herbivores, but there is no obvious pattern in where the omnivores appear. This suggests that perhaps features other than dentition and weight might be needed to separate omnivores from herbivores and carnivores.

---

## 25.5 Terms Introduced in Chapter

Euclidean mean

dissimilarity

centroid

k-means clustering

standard normal distribution

z-scaling

linear scaling

min-max scaling

linear interpolation

---

**191** Though k-means clustering is probably the most commonly used clustering method, it is not the most appropriate method in all situations. Two other widely used methods, not covered in this book, are hierarchical clustering and EM-clustering.

**192** The most widely used k-means algorithm is attributed to James MacQueen, and was first published in 1967. However, other approaches to k-means clustering were used as early as the 1950s.

**193** Unfortunately, in many applications we need to use a distance metric, e.g., earth-movers distance or dynamic-time-warping distance that has a higher computational complexity.

**194** Or, perhaps, species have chosen food based on their dentition. As we pointed out in Section 22.4, correlation does not imply causation.

**195** We included the information about weight because the author has been told, on more than one occasion, that there is a causal relationship between weight and eating habits.

**196** A normal distribution with a mean of  $0$  and a standard deviation of  $1$  is called a **standard normal distribution**.

**197** We say “close,” because floating-point numbers are only an approximation to the reals.

# 26

## CLASSIFICATION METHODS

The most common application of supervised machine learning is building classification models. A **classification model**, or classifier, is used to label an example as belonging to one of a finite set of categories. Deciding whether an email message is spam, for example, is a classification problem. In the literature, these categories are typically called **classes** (hence the name classification). Equivalently, we can describe an example as belonging to a class or as having a **label**.

In **one-class learning**, the training set contains examples drawn from only one class. The goal is to learn a model that predicts whether an example belongs to that class. One-class learning is useful when it is difficult to find training examples that lie outside the class. One-class learning is frequently used for building anomaly detectors, e.g., detecting previously unseen kinds of attacks on a computer network.

In **two-class learning** (often called **binary classification**), the training set contains examples drawn from exactly two classes (typically called positive and negative), and the objective is to find a boundary that separates the two classes. **Multi-class learning** involves finding boundaries that separate more than two classes from each other.

In this chapter, we look at two widely used supervised learning methods for solving classification problems: k-nearest neighbors and regression. Before we do, we address the question of how to evaluate the classifiers produced by these methods.

The code in this chapter assumes the import statements

```
import pandas as pd  
import numpy as np
```

```
import matplotlib.pyplot as plt
import random
import sklearn.linear_model as sklm
import sklearn.metrics as skm
```

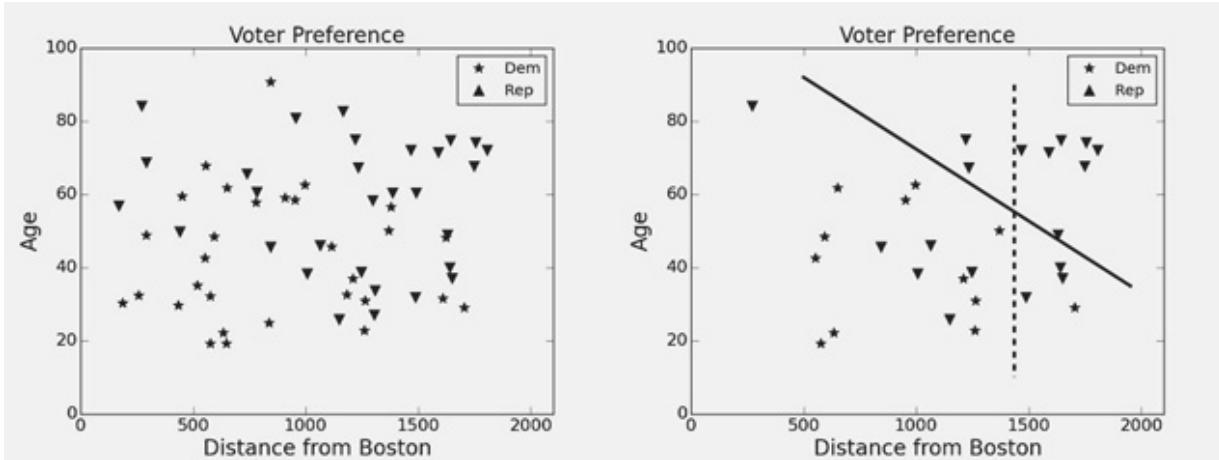
---

## 26.1 Evaluating Classifiers

Those of you who read Chapter 20 might recall that part of that chapter addressed the question of choosing a degree for a linear regression that would 1) provide a reasonably good fit for the available data, and 2) have a reasonable chance of making good predictions about as yet unseen data. The same issues arise when using supervised machine learning to train a classifier.

We start by dividing our data into two sets, a training set and a **test set**. The training set is used to learn a model, and the test set is used to evaluate that model. When we train the classifier, we attempt to minimize **training error**, i.e., errors in classifying the examples in the training set, subject to certain constraints. The constraints are designed to increase the probability that the model will perform reasonably well on as yet unseen data. Let's look at this pictorially.

The chart on the left of [Figure 26-1](#) shows a representation of voting patterns for 60 (simulated) American citizens. The x-axis is the distance of the voter's home from Boston, Massachusetts. The y-axis is the age of the voter. The stars indicate voters who usually vote Democratic, and the triangles voters who usually vote Republican. The chart on the right in [Figure 26-1](#) shows a training set containing a randomly chosen sample of 30 of those voters. The solid and dashed lines show two possible boundaries between the two populations. For the model based on the solid line, points below the line are classified as Democratic voters. For the model based on the dotted line, points to the left of the line are classified as Democratic voters.



[Figure 26-1](#) Plots of voter preferences

Neither boundary separates the training data perfectly. The training errors for the two models are shown in the **confusion matrices** in [Figure 26-2](#). The top-left corner of each shows the number of examples classified as Democratic that are actually Democratic, i.e., the true positives. The bottom-left corner shows the number of examples classified as Democratic that are actually Republican, i.e., the false positives. The righthand column shows the number of false negatives on the top and the number of true negatives on the bottom.

		Predicted Democratic	
		Pos	Neg
Actually Dem.	Pos	12	0
	Neg	9	9
Actually Rep.	Pos	11	1
	Neg	8	10

Solid Line                            Dashed Line

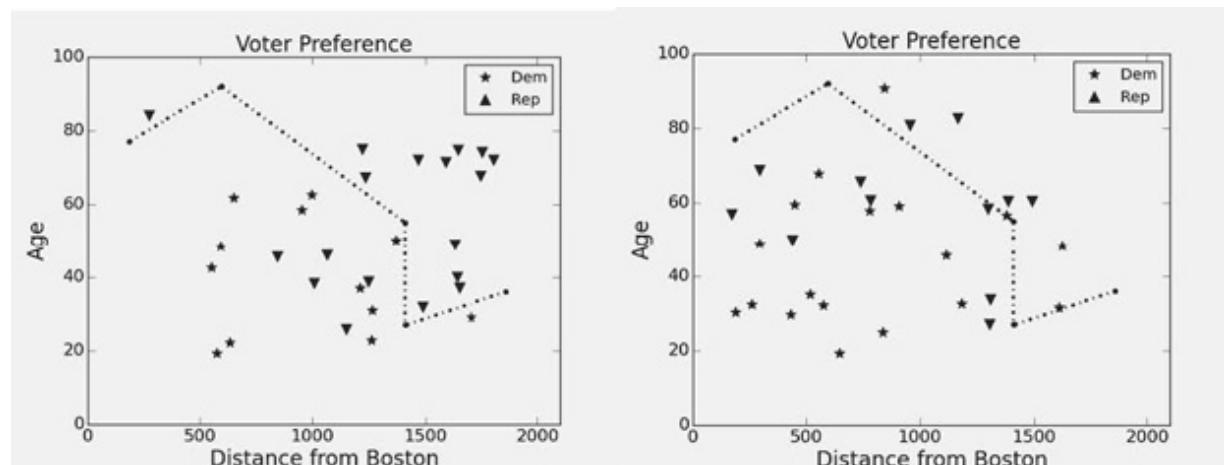
[Figure 26-2](#) Confusion matrices

The **accuracy** of each classifier on the training data can be calculated as

$$accuracy = \frac{true\ positive + true\ negative}{true\ positive + true\ negative + false\ positive + false\ negative}$$

In this case, each classifier has an accuracy of 0.7. Which does a better job of fitting the training data? It depends upon whether we are more concerned about misclassifying Republicans as Democrats, or vice versa.

If we are willing to draw a more complex boundary, we can get a classifier that does a more accurate job of classifying the training data. The classifier pictured in [Figure 26-3](#), for example, has an accuracy of about 0.83 on the training data, as depicted in the left plot of the figure. However, as we saw in our discussion of linear regression in Chapter 20, the more complicated the model, the higher the probability that it has been overfit to the training data. The righthand plot in [Figure 26-3](#) depicts what happens if we apply the complex model to the holdout set—the accuracy drops to 0.6.



[Figure 26-3](#) A more complex model

Accuracy is a reasonable way to evaluate a classifier when the two classes are of roughly equal size. It is a terrible way to evaluate a classifier when there is a large **class imbalance**. Imagine that you are charged with evaluating a classifier that predicts whether a person has a potentially fatal disease occurring in about 0.1% of the population to be tested. Accuracy is not a particularly useful statistic,

since 99.9% accuracy can be attained by merely declaring all patients disease-free. That classifier might seem great to those charged with paying for the treatment (nobody would get treated!), but it might not seem so great to those worried that they might have the disease.

Fortunately, there are statistics about classifiers that shed light when classes are imbalanced:

$$\text{sensitivity} = \frac{\text{true positive}}{\text{true positive} + \text{false negative}}$$

$$\text{specificity} = \frac{\text{true negative}}{\text{true negative} + \text{false positive}}$$

$$\text{positive predictive value} = \frac{\text{true positive}}{\text{true positive} + \text{false positive}}$$

$$\text{negative predictive value} = \frac{\text{true negative}}{\text{true negative} + \text{false negative}}$$

**Sensitivity** (called **recall** in some fields) is the true positive rate, i.e., the proportion of positives that are correctly identified as such. **Specificity** is the true negative rate, i.e., the proportion of negatives that are correctly identified as such. **Positive predictive value** is the probability that an example classified as positive is truly positive. **Negative predictive value** is the probability that an example classified as negative is truly negative.

Implementations of these statistical measures and a function that uses them to generate some statistics are in [Figure 26-4](#). We will use these functions later in this chapter.

```

def accuracy(true_pos, false_pos, true_neg, false_neg):
    numerator = true_pos + true_neg
    denominator = true_pos + true_neg + false_pos + false_neg
    return numerator/denominator

def sensitivity(true_pos, false_neg):
    try:
        return true_pos/(true_pos + false_neg)
    except ZeroDivisionError:
        return float('nan')

def specificity(true_neg, false_pos):
    try:
        return true_neg/(true_neg + false_pos)
    except ZeroDivisionError:
        return float('nan')

def pos_pred_val(true_pos, false_pos):
    try:
        return true_pos/(true_pos + false_pos)
    except ZeroDivisionError:
        return float('nan')

def neg_pred_val(true_neg, false_neg):
    try:
        return true_neg/(true_neg + false_neg)
    except ZeroDivisionError:
        return float('nan')

def get_stats(true_pos, false_pos, true_neg, false_neg,
             toPrint = True):
    accur = accuracy(true_pos, false_pos, true_neg, false_neg)
    sens = sensitivity(true_pos, false_neg)
    spec = specificity(true_neg, false_pos)
    ppv = pos_pred_val(true_pos, false_pos)
    if toPrint:
        print(' Accuracy =', round(accur, 3))
        print(' Sensitivity =', round(sens, 3))
        print(' Specificity =', round(spec, 3))
        print(' Pos. Pred. Val. =', round(ppv, 3))
    return (accur, sens, spec, ppv)

```

[Figure 26-4](#). Functions for evaluating classifiers

---

## 26.2 Predicting the Gender of Runners

Earlier in this book, we used data from the Boston Marathon to illustrate a number of statistical concepts. We will now use the same data to illustrate the application of various classification methods. The task is to predict the gender of a runner given the runner's age and finishing time.

The function `build_marathon_examples` in [Figure 26-6](#) reads in the data from a CSV file of the form shown in [Figure 26-5](#), and then builds a set of examples. Each example is an instance of class `Runner`. Each runner has a label (gender) and a feature vector (age and finishing time). The only interesting method in `Runner` is `feature_dist`. It returns the Euclidean distance between the feature vectors of two runners.

Name,Gender,Age,Div,Ctry,Time
Gebremariam Gebregziabher,M,27,14,ETH,142.93
Matebo Levy,M,22,2,KEN,133.10
Cherop Sharon,F,28,1,KEN,151.83

[Figure 26-5](#) First few lines of `bm_results2012.csv`

The next step is to split the examples into a training set and a held-out test set. As is frequently done, we use 80% of the data for training and test on the remaining 20%. This is done using the function `divide_80_20` at the bottom of [Figure 26-6](#). Notice that we select the training data at random. It would have taken less code to simply select the first 80% of the data, but that runs the risk of not being representative of the set as a whole. If the file had been sorted by finishing time, for example, we would get a training set biased towards the better runners.

We are now ready to look at different ways of using the training set to build a classifier that predicts the gender of a runner. Inspection reveals that 58% of the runners in the training set are male. So, if we guess male all the time, we should expect an accuracy of 58%. Keep this baseline in mind when looking at the performance of more sophisticated classification algorithms.

```

class Runner(object):
    def __init__(self, name, gender, age, time):
        self._name = name
        self._feature_vec = np.array([age, time])
        self._label = gender

    def feature_dist(self, other):
        return ((self._feature_vec-other._feature_vec)**2).sum()**0.5

    def get_time(self):
        return self._feature_vec[1]

    def get_age(self):
        return self._feature_vec[0]

    def get_label(self):
        return self._label

    def get_features(self):
        return self._feature_vec

    def __str__(self):
        return (f'{self._name}: {self.get_age()}, ' +
               f'{self.get_time()}, {self._label}')

```

[Figure 26-6](#) Build examples and divide data into training and test sets

---

## 26.3 K-nearest Neighbors

**K-nearest neighbors** (KNN) is probably the simplest of all classification algorithms. The “learned” model is simply the training examples themselves. New examples are assigned a label based on how similar they are to examples in the training data.

Imagine that you and a friend are strolling through the park and spot a bird. You believe that it is a yellow-throated woodpecker, but your friend is pretty sure that it is a golden-green woodpecker. You rush home and dig out your cache of bird books (or, if you are under 35, go to your favorite search engine) and start looking at labeled pictures of birds. Think of these labeled pictures as the training set. None of the pictures is an exact match for the bird you saw, so you settle for selecting the five that look the most like the bird you saw (the five “nearest neighbors”). The majority of them are photos of a yellow-throated woodpecker—you declare victory.

A weakness of KNN (and other) classifiers is that they often give poor results when the distribution of examples in the training data is different from that in the test data. If the frequency of pictures of bird species in the book is the same as the frequency of that species in your neighborhood, KNN will probably work well. Suppose, however, that despite the species being equally common in your neighborhood, your books contain 30 pictures of yellow-throated woodpeckers and only one of a golden-green woodpecker. If a simple majority vote is used to determine the classification, the yellow-throated woodpecker will be chosen even if the photos don't look much like the bird you saw. This problem can be partially mitigated by using a more complicated voting scheme in which the k-nearest neighbors are weighted based on their similarity to the example being classified.

The functions in [Figure 26-7](#) implement a k-nearest neighbors classifier that predicts the gender of a runner based on the runner's age and finishing time. The implementation is brute force. The function `find_k_nearest` is linear in the number of examples in `example_set`, since it computes the feature distance between `example` and each element in `example_set`. The function `k_nearest_classify` uses a simple majority-voting scheme to do the classification. The complexity of `k_nearest_classify` is  $O(\text{len}(\text{training}) * \text{len}(\text{test\_set}))$ , since it calls the function `find_k_nearest` a total of `len(test_set)` times.

```

def find_k_nearest(example, example_set, k):
    k_nearest, distances = [], []
    # Build lists containing first k examples and their distances
    for i in range(k):
        k_nearest.append(example_set[i])
        distances.append(example.feature_dist(example_set[i])))
    max_dist = max(distances) #Get maximum distance
    # Look at examples not yet considered
    for e in example_set[k:]:
        dist = example.feature_dist(e)
        if dist < max_dist:
            #replace farther neighbor by this one
            max_index = distances.index(max_dist)
            k_nearest[max_index] = e
            distances[max_index] = dist
            max_dist = max(distances)
    return k_nearest, distances

def k_nearest_classify(training_set, test_set, label, k):
    """Assumes training_set & test_set lists of examples, k an int
       Uses a k-nearest neighbor classifier to predict
           whether each example in test_set has the given label
       Returns number of true positives, false positives,
           true negatives, and false negatives"""
    true_pos, false_pos, true_neg, false_neg = 0, 0, 0, 0
    for e in test_set:
        print('Classifying', e)
        nearest, distances = find_k_nearest(e, training_set, k)
        # conduct vote
        num_match = 0
        for i in range(len(nearest)):
            if nearest[i].get_label() == label:
                num_match += 1
        if num_match > k//2: # guess label
            if e.get_label() == label:
                true_pos += 1
            else:
                false_pos += 1
        else: # guess not label
            if e.get_label() != label:
                true_neg += 1
            else:
                false_neg += 1
    return true_pos, false_pos, true_neg, false_neg

```

[Figure 26-7](#) Finding the k-nearest neighbors

## When the code

```

examples = build_marathon_examples('bm_results2012.csv')
training, test_set = divide_80_20(examples)

```

```
true_pos, false_pos, true_neg, false_neg =\  
    k_nearest_classify(training, test_set, 'M', 9)  
get_stats(true_pos, false_pos, true_neg, false_neg)
```

was run, it printed

```
Accuracy = 0.65  
Sensitivity = 0.715  
Specificity = 0.563  
Pos. Pred. Val. = 0.684
```

Should we be pleased that we can predict gender with 65% accuracy given age and finishing time? One way to evaluate a classifier is to compare it to a classifier that doesn't even look at age and finishing time. The classifier in [Figure 26-8](#) first uses the examples in `training` to estimate the probability of a randomly chosen example in `test_set` being from class `label`. Using this prior probability, it then randomly assigns a label to each example in `test_set`.

```

def prevalence_classify(training_set, test_set, label):
    """Assumes training_set & test_set lists of examples
    Uses a prevalence-based classifier to predict
        whether each example in test_set is of class label
    Returns number of true positives, false positives,
        true negatives, and false negatives"""
    num_with_label = 0
    for e in training:
        if e.get_label() == label:
            num_with_label += 1
    prob_label = num_with_label / len(training_set)
    true_pos, false_pos, true_neg, false_neg = 0, 0, 0, 0
    for e in test_set:
        if random.random() < prob_label: #guess label
            if e.get_label() == label:
                true_pos += 1
            else:
                false_pos += 1
        else: #guess not label
            if e.get_label() != label:
                true_neg += 1
            else:
                false_neg += 1
    return true_pos, false_pos, true_neg, false_neg

```

[Figure 26-8](#) Prevalence-based classifier

When we test `prevalence_classify` on the same Boston Marathon data on which we tested KNN, it prints

```

Accuracy = 0.514
Sensitivity = 0.593
Specificity = 0.41
Pos. Pred. Val. = 0.57

```

indicating that we are reaping a considerable advantage from considering age and finishing time.

That advantage has a cost. If you run the code in [Figure 26-7](#), you will notice that it takes a rather long time to finish. There are 17,233 training examples and 4,308 test examples, so there are nearly 75 million distances calculated. This raises the question of whether we really need to use all of the training examples. Let's see what happens if we simply **downsample** the training data by a factor of 10.

If we run

```
reduced_training = random.sample(training,  
len(training)//10)  
true_pos, false_pos, true_neg, false_neg =\  
    k_nearest_classify(reduced_training, test_set, 'M',  
9)  
get_stats(true_pos, false_pos, true_neg, false_neg)
```

it completes in one-tenth the time, with little change in classification performance:

```
Accuracy = 0.638  
Sensitivity = 0.667  
Specificity = 0.599  
Pos. Pred. Val. = 0.687
```

In practice, when people apply KNN to large data sets, they often downsample the training data. An even more common alternative is to use some sort of fast approximate-KNN algorithm.

In the above experiments, we set  $k$  to 9. We did not choose this number for its role in science (the number of planets in our solar system),<sup>198</sup> its religious significance (the number of forms of the Hindu goddess Durga), or its sociological importance (the number of hitters in a baseball lineup). Instead, we learned  $k$  from the training data by using the code in [Figure 26-9](#) to search for a good  $k$ .

The outer loop tests a sequence of values for  $k$ . We test only odd values to ensure that when the vote is taken in `k_nearest_classify`, there will always be a majority for one gender or the other.

The inner loop tests each value of  $k$  using **n-fold cross validation**. In each of the `num_folds` iterations of the loop, the original training set is split into a new training set/test set pair. We then compute the accuracy of classifying the new test set using  $k$ -nearest neighbors and the new training set. When we exit the inner loop, we calculate the average accuracy of the `num_folds` folds.

When we ran the code, it produced the plot in [Figure 26-10](#). As we can see, 17 was the value of  $k$  that led to the best accuracy across 5 folds. Of course, there is no guarantee that some value larger than 21 might not have been even better. However, once  $k$  reached 9, the accuracy fluctuated over a reasonably narrow range, so we chose to use 9.

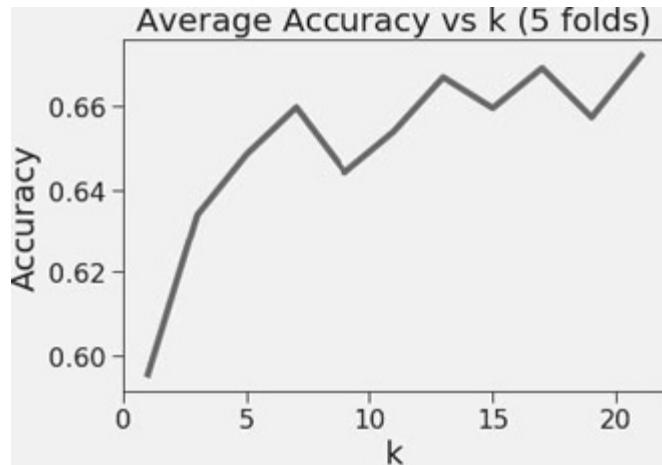
```

def find_k(training_set, min_k, max_k, num_folds, label):
    # Find average accuracy for range of odd values of k
    accuracies = []
    for k in range(min_k, max_k + 1, 2):
        score = 0.0
        for i in range(num_folds):
            # downsample to reduce computation time
            fold = random.sample(training_set,
                                  min(5000, len(training_set)))
            examples, test_set = divide_80_20(fold)
            true_pos, false_pos, true_neg, false_neg =\
                k_nearest_classify(examples, test_set, label, k)
            score += accuracy(true_pos, false_pos, true_neg, false_neg)
        accuracies.append(score/num_folds)
    plt.plot(range(min_k, max_k + 1, 2), accuracies)
    plt.title('Average Accuracy vs k (' + str(num_folds) +
              ' folds)')
    plt.xlabel('k')
    plt.ylabel('Accuracy')

find_k(training, 1, 21, 5, 'M')

```

[Figure 26-9](#) Searching for a good k

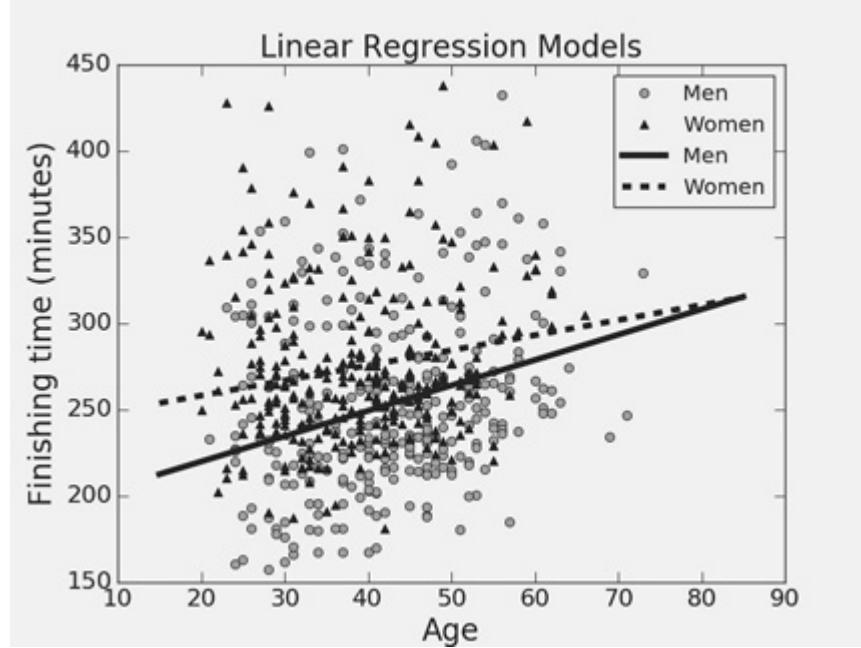


[Figure 26-10](#) Choosing a value for k

## 26.4 Regression-based Classifiers

In Chapter 20 we used linear regression to build models of data. We can try the same thing here and use the training data to build

separate models for the men and the women. The plot in [Figure 25-11](#) was produced by the code in [Figure 26-12](#).



[Figure 26-11](#) Linear regression models for men and women

```

# Build training sets for men and women
age_m, age_w, time_m, time_w = [], [], [], []
for e in training:
    if e.get_label() == 'M':
        age_m.append(e.get_age())
        time_m.append(e.get_time())
    else:
        age_w.append(e.get_age())
        time_w.append(e.get_time())
# downsample to make plot of examples readable
ages, times = [], []
for i in random.sample(range(len(age_m)), 300):
    ages.append(age_m[i])
    times.append(time_m[i])
# Produce scatter plot of examples
plt.plot(ages, times, 'yo', markersize = 6, label = 'Men')
ages, times = [], []
for i in random.sample(range(len(age_w)), 300):
    ages.append(age_w[i])
    times.append(time_w[i])
plt.plot(ages, times, 'k^', markersize = 6, label = 'Women')
# Learn two first-degree linear regression models
m_model = np.polyfit(age_m, time_m, 1)
f_model = np.polyfit(age_w, time_w, 1)
#Plot lines corresponding to models
xmin, xmax = 15, 85
plt.plot((xmin, xmax), (np.polyval(m_model,(xmin, xmax))),
          'k', label = 'Men')
plt.plot((xmin, xmax), (np.polyval(f_model,(xmin, xmax))),
          'k--', label = 'Women')
plt.title('Linear Regression Models')
plt.xlabel('Age')
plt.ylabel('Finishing time (minutes)')
plt.legend()

```

[Figure 26-12](#) Produce and plot linear regression models

A quick glance at [Figure 26-11](#) is enough to see that the linear regression models explain only a small amount of the variance in the data.<sup>199</sup> Nevertheless, it is possible to use these models to build a classifier. Each model attempts to capture the relationship between age and finishing time. This relationship is different for men and women, a fact we can exploit in building a classifier. Given an example, we ask whether the relationship between age and finishing time is closer to the relationship predicted by the model for male

runners (the solid line) or to the model for female runners (the dashed line). This idea is implemented in [Figure 26-13](#).

When the code is run, it prints

```
Accuracy = 0.614
Sensitivity = 0.684
Specificity = 0.523
Pos. Pred. Val. = 0.654
```

The results are better than random, but worse than for KNN.

```
true_pos, false_pos, true_neg, false_neg = 0, 0, 0, 0
for e in test_set:
    age = e.get_age()
    time = e.get_time()
    if (abs(time - np.polyval(m_model,age)) <
        abs(time - np.polyval(f_model, age))):
        if e.get_label() == 'M':
            true_pos += 1
        else:
            false_pos += 1
    else:
        if e.get_label() == 'F':
            true_neg += 1
        else:
            false_neg += 1
get_stats(true_pos, false_pos, true_neg, false_neg)
```

[Figure 26-13](#) Using linear regression to build a classifier

You might be wondering why we took this indirect approach to using linear regression, rather than explicitly building a model using some function of age and time as independent variable and real numbers (say `0` for female and `1` for male) as the dependent variable.

We could easily build such a model using `polyfit` to map a function of age and time to a real number. However, what would it mean to predict that some runner is halfway between male and female? Were there some hermaphrodites in the race? Perhaps we can interpret the y-axis as the probability that a runner is male. Not really. There is not even a guarantee that applying `polyval` to the model will return a value between `0` and `1`.

Fortunately, there is a form of regression, **logistic regression**,<sup>200</sup> designed explicitly for predicting the probability of an event. The Python library `sklearn`<sup>201</sup> provides a good implementation of logistic regression—and many other useful functions and classes related to machine learning.

The module `sklearn.linear_model` contains the class `LogisticRegression`. The `_init_` method of this class has a large number of parameters that control things such as the optimization algorithm used to solve the regression equation. They all have default values, and on most occasions, it is fine to stick with those.

The central method of class `LogisticRegression` is `fit`. The method takes as arguments two sequences (tuples, lists, or arrays) of the same length. The first is a sequence of feature vectors and the second a sequence of the corresponding labels. In the literature, these labels are typically called **outcomes**.

The `fit` method returns an object of type `LogisticRegression` for which coefficients have been learned for each feature in the feature vector. These coefficients, often called **feature weights**, capture the relationship between the feature and the outcome. A positive feature weight suggests a positive correlation between the feature and the outcome, and a negative feature weight suggests a negative correlation. The absolute magnitude of the weight is related to the strength of the correlation.<sup>202</sup> The values of these weights can be accessed using the `coef_` attribute of `LogisticRegression`. Since it is possible to train a `LogisticRegression` object on multiple outcomes (called classes in the documentation for the package), the value of `coef_` is a sequence in which each element contains the sequence of weights associated with a single outcome. So, for example, the expression `model.coef_[1][0]` denotes the value of the coefficient of the first feature for the second outcome.

Once the coefficients have been learned, the method `predict_proba` of the `LogisticRegression` class can be used to predict the outcome associated with a feature vector. The method `predict_proba` takes a single argument (in addition to `self`), a sequence of feature vectors. It returns an array of arrays, one per feature vector. Each element in the returned array contains a prediction for the corresponding feature vector. The reason that the

prediction is an array is that it contains a probability for each label used in building model.

The code in [Figure 26-14](#) contains a simple illustration of how this all works. It first creates a list of 100,000 examples, each of which has a feature vector of length 3 and is labeled either 'A', 'B', 'C', or 'D'. The first two feature values for each example are drawn from a Gaussian with a standard deviation of 0.5, but the means vary depending upon the label. The value of the third feature is chosen at random, and therefore should not be useful in predicting the label. After creating the examples, the code generates a logistic regression model, prints the feature weights, and finally the probabilities associated with four examples.

```
feature_vecs, labels = [], []
for i in range(25000): # create 4 examples in each iteration
    feature_vecs.append([random.gauss(0, 0.5), random.gauss(0, 0.5),
                         random.random()])
    labels.append('A')
    feature_vecs.append([random.gauss(0, 0.5), random.gauss(2, 0.),
                         random.random()])
    labels.append('B')
    feature_vecs.append([random.gauss(2, 0.5), random.gauss(0, 0.5),
                         random.random()])
    labels.append('C')
    feature_vecs.append([random.gauss(2, 0.5), random.gauss(2, 0.5),
                         random.random()])
    labels.append('D')

model = sklm.LogisticRegression().fit(feature_vecs, labels)
print('model.classes_ =', model.classes_)

for i in range(len(model.coef_)):
    print('For label', model.classes_[i],
          'feature weights =', model.coef_[i].round(4))

print('[0, 0] probs =', model.predict_proba([[0, 0, 1]])[0].round(4))
print('[0, 2] probs =', model.predict_proba([[0, 2, 2]])[0].round(4))
print('[2, 0] probs =', model.predict_proba([[2, 0, 3]])[0].round(4))
print('[2, 2] probs =', model.predict_proba([[2, 2, 4]])[0].round(4))
```

[Figure 26-14](#) Using `sklearn` to do multi-class logistic regression

When we ran the code in [Figure 26-14](#), it printed

```
model.classes_ = ['A' 'B' 'C' 'D']
For label A feature weights = [-4.7229 -4.3618  0.0595]
For label B feature weights = [-3.3346  4.7875  0.0149]
For label C feature weights = [ 3.7026 -4.4966 -0.0176]
For label D feature weights = [ 4.3548  4.0709 -0.0568]
[0, 0] probs = [9.998e-01 0.000e+00 2.000e-04 0.000e+00]
[0, 2] probs = [2.60e-03 9.97e-01 0.00e+00 4.00e-04]
[2, 0] probs = [3.000e-04 0.000e+00 9.996e-01 2.000e-04]
[2, 2] probs = [0.000e+00 5.000e-04 2.000e-04 9.992e-01]
```

Let's look first at the feature weights. The first line tells us that the first two features have roughly the same weight and are negatively correlated with the probability of an example having label '`A`'.<sup>203</sup> That is, the larger the value of the first two features, the less likely that the example is of type '`A`'. The third feature, which we expect to have little value in predicting the label, has a small value relative to the other two values, indicating that it is relatively unimportant. The second line tells us that the probability of an example having the label '`B`' is negatively correlated with value of the first feature, but positively with the second feature. Again, the third feature has a relatively small value. The third and fourth lines are mirror images of the first two lines.

Now, let's look at the probabilities associated with the four examples. The order of the probabilities corresponds to the order of the outcomes in the attribute `model.classes_`. As you would hope, when we predict the label associated with the feature vector `[0, 0]`, '`A`' has a very high probability and '`D`' a very low probability. Similarly, `[2, 2]` has a very high probability for '`D`' and a very low one for '`A`'. The probabilities associated with the middle two examples are also as expected.

The example in [Figure 26-15](#) is similar to the one in [Figure 26-14](#), except that we create examples of only two classes, '`A`' and '`D`', and don't include the irrelevant third feature.

```

feature_vecs, labels = [], []
for i in range(20000):
    feature_vecs.append([random.gauss(0, 0.5), random.gauss(0, 0.5)])
    labels.append('A')
    feature_vecs.append([random.gauss(2, 0.5), random.gauss(2, 0.5)])
    labels.append('D')

model = sklm.LogisticRegression().fit(feature_vecs, labels)
print('model.coef =', model.coef_.round(4))
print('[0, 0] probs =', model.predict_proba([[0, 0]])[0].round(4))
print('[0, 2] probs =', model.predict_proba([[0, 2]])[0].round(4))
print('[2, 0] probs =', model.predict_proba([[2, 0]])[0].round(4))
print('[2, 2] probs =', model.predict_proba([[2, 2]])[0].round(4))

```

[Figure 26-15](#) Example of two-class logistic regression

When we run the code in [Figure 26-15](#), it prints

```

model.coef = [[6.7081 6.5737]]
[0, 0] probs = [1. 0.]
[0, 2] probs = [0.5354 0.4646]
[2, 0] probs = [0.4683 0.5317]
[2, 2] probs = [0. 1.]

```

Notice that there is only one set of weights in `coef_`. When `fit` is used to produce a model for a binary classifier, it only produces weights for one label. This is sufficient because once `proba` has calculated the probability of an example being in either of the classes, the probability of it being in the other class is determined—since the probabilities must add up to 1. To which of the two labels do the weights in `coef_` correspond? Since the weights are positive, they must correspond to '`D`', since we know that the larger the values in the feature vector, the more likely the example is of class '`D`'. Traditionally, binary classification uses the labels `0` and `1`, and the classifier uses the weights for `1`. In this case, `coef_` contains the weights associated with the largest label, as defined by the `>` operator for type `str`.

Let's return to the Boston Marathon example. The code in [Figure 26-16](#) uses the `LogisticRegression` class to build and test a model for our Boston Marathon data. The function `apply_model` takes four arguments:

- `model`: an object of type `LogisticRegression` for which a fit has been constructed
- `test_set`: a sequence of examples. The examples have the same kinds of features and labels used in constructing the fit for `model`.
- `label`: The label of the positive class. The confusion matrix information returned by `apply_model` is relative to this label.
- `prob`: the probability threshold to use in deciding which label to assign to an example in `test_set`. The default value is `0.5`. Because it is not a constant, `apply_model` can be used to investigate the tradeoff between false positives and false negatives.

The implementation of `apply_model` first uses a list comprehension (Section 5.3.2) to build a list whose elements are the feature vectors of the examples in `test_set`. It then calls `model.predict_proba` to get an array of pairs corresponding to the prediction for each feature vector. Finally, it compares the prediction against the label associated with the example with that feature vector, and keeps track of and returns the number of true positives, false positives, true negatives, and false negatives.

When we ran the code, it printed

```
Feature weights for label M: age = 0.055, time = -0.011
Accuracy = 0.636
Sensitivity = 0.831
Specificity = 0.377
Pos. Pred. Val. = 0.638
```

Let's compare these results to what we got when we used KNN:

```
Accuracy = 0.65
Sensitivity = 0.715
Specificity = 0.563
Pos. Pred. Val. = 0.684
```

The accuracies and positive predictive values are similar, but logistic regression has a much higher sensitivity and a much lower specificity. That makes the two methods hard to compare. We can address this problem by adjusting the probability threshold used by `apply_model` so that it has approximately the same sensitivity as

KNN. We can find that probability by iterating over values of `prob` until we get a sensitivity close to that we got using KNN.

If we call `apply_model` with `prob = 0.578` instead of `0.5`, we get the results

```
Accuracy = 0.659
Sensitivity = 0.715
Specificity = 0.586
Pos. Pred. Val. = 0.695
```

In other words, the models have similar performance.

```
def apply_model(model, test_set, label, prob = 0.5):
    # Create vector containing feature vectors for all test examples
    test_feature_vecs = [e.get_features() for e in test_set]
    probs = model.predict_proba(test_feature_vecs)
    true_pos, false_pos, true_neg, false_neg = 0, 0, 0, 0
    for i in range(len(probs)):
        if probs[i][1] > prob:
            if test_set[i].get_label() == label:
                true_pos += 1
            else:
                false_pos += 1
        else:
            if test_set[i].get_label() != label:
                true_neg += 1
            else:
                false_neg += 1
    return true_pos, false_pos, true_neg, false_neg

examples = build_marathon_examples('bm_results2012.csv')
training, test = divide_80_20(examples)

feature_vecs, labels = [], []
for e in training:
    feature_vecs.append([e.get_age(), e.get_time()])
    labels.append(e.get_label())
model = sklm.LogisticRegression().fit(feature_vecs, labels)
print('Feature weights for label M:',
      'age =', str(round(model.coef_[0][0], 3)) + ',',
      'time =', round(model.coef_[0][1], 3))
true_pos, false_pos, true_neg, false_neg = \
    apply_model(model, test, 'M', 0.5)
get_stats(true_pos, false_pos, true_neg, false_neg)
```

[Figure 26-16](#) Use logistic regression to predict gender

Since it can be complicated to explore the ramifications of changing the decision threshold for a logistic regression model, people often use something called the **receiver operating characteristic curve**,<sup>204</sup> or **ROC curve**, to visualize the tradeoff between sensitivity and specificity. The curve plots the true positive rate (sensitivity) against the false positive rate ( $1 - \text{specificity}$ ) for multiple decision thresholds.

ROC curves are often compared to one another by computing the area under the curve (**AUROC**, often abbreviated as **AUC**). This area is equal to the probability that the model will assign a higher probability of being positive to a randomly chosen positive example than to a randomly chosen negative example. This is known as the **discrimination** of the model. Keep in mind that discrimination says nothing about the accuracy, often called the **calibration**, of the probabilities. We could, for example, divide all of the estimated probabilities by 2 without changing the discrimination—but it would certainly change the accuracy of the estimates.

The code in [Figure 26-17](#) plots the ROC curve for the logistic regression classifier as a solid line, [Figure 26-18](#). The dotted line is the ROC for a random classifier—a classifier that chooses the label randomly. We could have computed the AUROC by first interpolating (because we have only a discrete number of points) and then integrating the ROC curve, but we got lazy and simply called the function `sklearn.metrics.auc`.

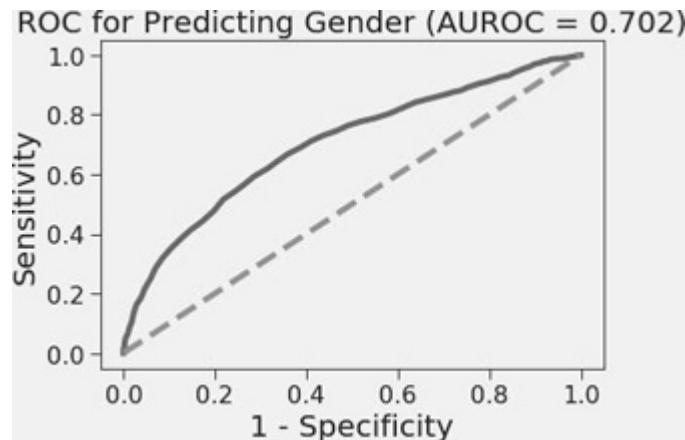
```

def build_ROC(model, test_set, label, title, plot = True):
    xVals, yVals = [], []
    for p in np.arange(0, 1, 0.01):
        true_pos, false_pos, true_neg, false_neg = \
            apply_model(model, test_set, label, p)
        xVals.append(1.0 - specificity(true_neg, false_pos))
        yVals.append(sensitivity(true_pos, false_neg))
    auroc = skm.auc(xVals, yVals)
    if plot:
        plt.plot(xVals, yVals)
        plt.plot([0,1], [0,1], '--')
        plt.title(title + ' (AUROC = ' +
                  str(round(auroc, 3)) + ')')
        plt.xlabel('1 - Specificity')
        plt.ylabel('Sensitivity')
    return auroc

build_ROC(model, test, 'M', 'ROC for Predicting Gender')

```

[Figure 26-17](#) Construct ROC curve and find AUROC



[Figure 26-18](#) ROC curve and AUROC

**Finger exercise:** Write code to plot the ROC curve and compute the AUROC when the model built in [Figure 26-16](#) is tested on 200 randomly chosen competitors. Use that code to investigate the impact of the number of training examples (try varying it from  $10$  to  $10^{10}$  in increments of  $50$ ) on the AUROC.

---

## 26.5 Surviving the *Titanic*

On the morning of April 15, 1912, the RMS *Titanic* hit an iceberg and sank in the North Atlantic. Of the roughly 1,300 passengers on board, 832 perished in the disaster. Many factors contributed to the disaster, including navigational error, inadequate lifeboats, and the slow response of a nearby ship. Whether individual passengers survived had an element of randomness, but was far from completely random. One interesting question is whether it is possible to build a reasonably good model for predicting survival using only information from the ship's passenger manifest.

In this section, we build a classification model from a CSV file containing information for 1046 passengers.<sup>205</sup> Each line of the file contains information about a single passenger: cabin class (1st, 2nd, or 3rd), age, gender, whether the passenger survived the disaster, and the passenger's name. The first few lines of the CSV file are

```
Class,Age,Gender,Survived,Last Name,Other Names
1,29.0,F,1,Allen, Miss. Elisabeth Walton
1,0.92,M,1,Allison, Master. Hudson Trevor
1,2.0,F,0,Allison, Miss. Helen Loraine
```

Before building a model, it's probably a good idea take a quick look at the data with Pandas. Doing this often provides useful insights into the role various features might play in a model. Executing the code

```
manifest = pd.read_csv('TitanicPassengers.csv')
print(manifest.corr().round(2))
```

produces the correlation table

	Class	Age	Survived
Class	1.00	-0.41	-0.32
Age	-0.41	1.00	-0.06
Survived	-0.32	-0.06	1.00

Why doesn't `Gender` appear in this table? Because it is not encoded as a number in the CSV file. Let's deal with that and see what the correlations look like.

```

manifest['Gender'] = (manifest['Gender'].
                      apply(lambda g: 1 if g == 'M' else 0))
print(manifest.corr().round(2))

```

produces

	Class	Age	Gender	Survived
Class	1.00	-0.41	0.14	-0.32
Age	-0.41	1.00	0.06	-0.06
Gender	0.14	0.06	1.00	-0.54
Survived	-0.32	-0.06	-0.54	1.00

The negative correlations of `Class` and `Gender` with `Survived` suggest that it might indeed be possible to build a predictive model using information in the manifest. (Because we have coded males as 1 and females as 0, the negative correlation of `Survived` and `Gender` tells us that women are more likely to have survived than men. Similarly, the negative correlation with `Class` tells us that it was safer to have been in first class.)

Now, let's build a model using logistic regression. We chose to use logistic regression because

- It is the most commonly used classification method.
- By examining the weights produced by logistic regression, we can gain some insight into why some passengers were more likely to have survived than others.

[Figure 26-19](#) defines class `Passenger`. The only thing of interest in this code is the encoding of cabin class. Though the CSV file encodes the cabin class as an integer, it is really shorthand for a category. Cabin classes do not behave like numbers, e.g., a first-class cabin plus a second-class cabin does not equal a third-class cabin. We encode cabin class using three binary features (one per possible cabin class). For each passenger, exactly one of these variables is set to 1, and the other two are set to 0.

This is an example of an issue that frequently arises in machine learning. **Categorical** (sometimes called nominal) features are the natural way to describe many things, e.g., the home country of a runner. It's easy to replace these by integers, e.g., we could choose a representation for countries based on their ISO 3166-1 numeric

code,[206](#) e.g., 076 for Brazil, 826 for the United Kingdom, and 862 for Venezuela. The problem with doing this is that the regression will treat these as numerical variables, thus using a nonsensical ordering on the countries in which Venezuela would be closer to the UK than it is to Brazil.

This problem can be avoided by converting categorical variables to binary variables, as we did with cabin class. One potential problem with doing this is that it can lead to very long and sparse feature vectors. For example, if a hospital dispenses 2000 different drugs, we would convert one categorical variable into 2000 binary variables, one for each drug.

[Figure 26-20](#) contains code that uses Pandas to read the data from a file and build a set of examples from the data about the *Titanic*.

Now that we have the data, we can build a logistic regression model using the same code we used to build a model of the Boston Marathon data. However, because the data set has a relatively small number of examples, we need to be concerned about using the evaluation method we employed earlier. It is entirely possible to get an unrepresentative 80-20 split of the data, and then generate misleading results.

To ameliorate the risk, we create many 80-20 splits (each split is created using the `divide_80_20` function defined in [Figure 26-6](#)), build and evaluate a classifier for each, and then report mean values and 95% confidence intervals, using the code in [Figure 26-21](#) and [Figure 26-22](#).

```

class Passenger(object):
    features = ('1st Class', '2nd Class', '3rd Class',
                'age', 'male')
    def __init__(self, pClass, age, gender, survived, name):
        self.name = name
        self.feature_vec = [0, 0, 0, age, gender]
        self.feature_vec[pClass - 1] = 1
        self.label = survived
        self.cabinClass = pClass
    def distance(self, other):
        return minkowski_dist(self.veatureVec, other.feature_vec, 2)
    def get_class(self):
        return self.cabinClass
    def get_age(self):
        return self.feature_vec[3]
    def get_gender(self):
        return self.feature_vec[4]
    def get_name(self):
        return self.name
    def get_features(self):
        return self.feature_vec[:]
    def get_label(self):
        return self.label

```

[Figure 26-19](#) Class Passenger

```

def build_Titanic_examples():
    manifest = pd.read_csv('TitanicPassengers.csv')
    examples = []
    for index, row in manifest.iterrows():
        p = Passenger(row['Class'], row['Age'],
                      1 if row['Gender'] == 'M' else 0,
                      row['Survived'],
                      row['Last Name'] + row['Other Names'])
        examples.append(p)
    return examples

```

[Figure 26-20](#) Read *Titanic* data and build list of examples[207](#)

```

def test_models(examples, num_trials, print_stats, print_weights):
    stats, weights = [], [[], [], [], [], []]
    for i in range(num_trials):
        training, test_set = divide_80_20(examples)
        xVals, yVals = [], []
        for e in training:
            xVals.append(e.get_features())
            yVals.append(e.get_label())
        xVals = np.array(xVals)
        yVals = np.array(yVals)
        model = sklm.LogisticRegression().fit(xVals, yVals)
        for i in range(len(Passenger.features)):
            weights[i].append(model.coef_[0][i])
        true_pos, false_pos, true_neg, false_neg = \
            apply_model(model, test_set, 1, 0.5)
        auroc = build_ROC(model, test_set, 1, None, False)
        tmp = get_stats(true_pos, false_pos, true_neg, false_neg, False)
        stats.append(tmp + (auroc,))
    print('Averages for', num_trials, 'trials')
    if print_weights:
        for feature in range(len(weights)):
            feature_mean = round(sum(weights[feature])/num_trials, 3)
            feature_std = np.std(weights[feature])
            print(' Mean weight', Passenger.features[feature],
                  '=', str(feature_mean) + ', 95% conf. int. =',
                  round(feature_mean - 1.96*feature_std, 3), 'to',
                  round(feature_mean + 1.96*feature_std, 3))
    if print_stats:
        summarize_stats(stats)

```

[Figure 26-21](#) Test models for *Titanic* survival

```

def summarize_stats(stats):
    """assumes stats a list of 5 floats: accuracy, sensitivity,
    specificity, pos. pred. val, ROC"""
    def print_stat(X, name):
        mean = round(sum(X)/len(X), 3)
        std = np.std(X)
        print(' Mean', name, '=', str(mean) + ',',
              '95% conf. int. =',
              round(mean - 1.96*std, 3), 'to',
              round(mean + 1.96*std, 3))
    accs, sens, specs, ppvs, aurocs = [], [], [], [], []
    for stat in stats:
        accs.append(stat[0])
        sens.append(stat[1])
        specs.append(stat[2])
        ppvs.append(stat[3])
        aurocs.append(stat[4])
    print_stat(accs, 'accuracy')
    print_stat(sens, 'sensitivity')
    print_stat(specs, 'specificity')

```

[Figure 26-22](#) Print statistics about classifiers

The call `test_models(build_Titanic_examples(), 100, True, False)` printed

```

Averages for 100 trials
Mean accuracy = 0.783, 95% conf. int. = 0.736 to 0.83
Mean sensitivity = 0.702, 95% conf. int. = 0.603 to 0.801
Mean specificity = 0.783, 95% conf. int. = 0.736 to 0.83
Mean pos. pred. val. = 0.702, 95% conf. int. = 0.603 to
0.801
Mean AUROC = 0.839, 95% conf. int. = 0.789 to 0.889

```

It appears that this small set of features is sufficient to do a reasonably good job of predicting survival. To see why, let's take a look at the weights of the various features. We can do that with the call `test_models(build_Titanic_examples(), 100, False, True)`, which printed

```

Averages for 100 trials
Mean weight 1st Class = 1.145, 95% conf. int. = 1.02 to
1.27
Mean weight 2nd Class = -0.083, 95% conf. int. = -0.185 to
0.019

```

Mean weight 3rd Class = -1.062, 95% conf. int. = -1.179 to -0.945

Mean weight age = -0.034, 95% conf. int. = -0.04 to -0.028

Mean weight male = -2.404, 95% conf. int. = -2.542 to -2.266

When it comes to surviving a shipwreck, it seems useful to be rich (a first-class cabin on the *Titanic* cost the equivalent of more than \$70,000 in today's U.S. dollars), young, and female.

---

## 26.6 Wrapping Up

In the last three chapters, we've barely scratched the surface of machine learning.

The same could be said about many of the other topics presented in the second part of this book. I've tried to give you a taste of the kind of thinking involved in using computation to better understand the world—in the hope that you will find ways to pursue the topic on your own.

---

## 26.7 Terms Introduced in Chapter

classification model

class

label

one-class learning

two-class learning

binary classification

multi-class learning

test set

training error

confusion matrix

accuracy

class imbalance  
sensitivity (recall)  
specificity (precision)  
positive predictive value (PPV)  
k-nearest neighbors (KNN)  
downsample  
negative predictive value  
n-fold cross validation  
logistic regression  
outcome  
feature weight  
ROC curve  
AUROC  
model discrimination  
calibration  
categorical feature

---

[198](#) Some of us still believe in planet Pluto.

[199](#). Though we fit the models to the entire training set, we choose to plot only a small subset of the training points. When we plotted all of them, the result was a blob in which it was hard to see any useful detail.

[200](#) It's called logistic regression because the optimization problem being solved involves an objective function based on the log of an odds ratio. Such functions are called logit functions, and their inverses are call logistic functions.

[201](#) This toolkit comes preinstalled with some Python IDEs, e.g., Anaconda. To learn more about this library and find out how to

install it, go to <http://scikit-learn.org>.

**202** This relationship is complicated by the fact that features are often correlated with each other. For example, age and finishing time are positively correlated. When features are correlated, the magnitudes of the weights are not independent of each other.

**203** The slight difference in the absolute values of the weights is attributable to the fact that our sample size is finite.

**204** It is called the “receiver operating characteristic” for historical reasons. It was first developed during World War II as way to evaluate the operating characteristics of devices receiving radar signals.

**205** The data was extracted from a data set constructed by R.J. Dawson, and used in “The ‘Unusual Episode’ Data Revisited,” *Journal of Statistics Education*, v. 3, n. 3, 1995.

**206** ISO 3166-1 numeric is part of the ISO 3166 standard published by the International Organization for Standardization. ISO 3166 defines unique codes for the names of countries and their subdivisions (e.g., states and provinces).

**207** PEP-8 recommends using lowercase letters, even for proper nouns. But the name `build_titanic_examples` seems to suggest a function used to build enormously large examples rather than a function used to build examples related to the ship.

# PYTHON 3.8 QUICK REFERENCE

## Common operations on numerical types

`i+j` is the sum of `i` and `j`.

`i-j` is `i` minus `j`.

`i*j` is the product of `i` and `j`.

`i//j` is floor division.

`i/j` is floating-point division.

`i%j` is the remainder when the `int i` is divided by the `int j`.

`i**j` is `i` raised to the power `j`.

`x += y` is equivalent to `x = x + y`. `*=` and `-=` work the same way.

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

## Boolean operators

`x == y` returns `True` if `x` and `y` are equal.

`x != y` returns `True` if `x` and `y` are not equal.

`<, >, <=, >=` have their usual meanings.

`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`.

## Common operations on sequence types

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

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

`seq1 + seq2` concatenates 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 new sequence that is a slice of `seq`.

`e in seq` tests whether `e` is contained in the sequence.

`e not in seq` tests whether `e` is not contained in the sequence.

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

## Common string methods

`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`; returns `-1` if `s1` is not in `s`.

`s.rfind(s1)` same as `find`, but starts from the end of `s`.

`s.index(s1)` same as `find`, but raises an exception 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 to lowercase.

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

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

`s.split(d)` Splits `s` using `d` as a delimiter. Returns a list of substrings of `s`.

## Common list methods

`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)` appends 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 `ValueError` if `e` not in `L`.

`L.pop(i)` removes and returns the item at index `i`; `i` defaults to `-1`. Raises `IndexError` if `L` is empty.

`L.sort()` has the side effect of sorting the elements of `L`.

`L.reverse()` has the side effect of reversing the order of the elements in `L`.

`L.copy()` returns a shallow copy of `L`.

`L.deepcopy()` returns a deep copy of `L`.

## Common operations on dictionaries

`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`.

`d.items()` returns a view of the (key, value) pairs in `d`.

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

`d[k]` returns the item in `d` with key `k`. Raises `KeyError` if `k` is not in `d`.

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

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

`del d[k]` removes element with key `k` from `d`. Raises `KeyError` if `k` is not in `d`.

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

## Common input/output mechanisms

`input(msg)` prints `msg` and then returns the value entered as a string.

`print(s1, ..., sn)` prints strings `s1, ..., sn` separated by spaces.

`open('file_name', 'w')` creates a file for writing.

`open('file_name', 'r')` opens an existing file for reading.

`open('file_name', 'a')` opens an existing file for appending.

`file_handle.read()` returns a string containing contents of the file.

`file_handle.readline()` returns the next line in the file.

`file_handle.readlines()` returns a list containing lines of the file.

`file_handle.write(s)` writes the string `s` to the end of the file.

`file_handle.writelines(L)` writes each element of `L` to the file as a separate line.

`file_handle.close()` closes the file.

# INDEX

`__init__`, 179  
`__lt__`, 190  
`__name__`, 328  
`__str__`, 184

: slicing operator, 29, 107

$\alpha$  (alpha), threshold for significance, 461  
 $\mu$  (mu), mean, 372  
 $\sigma$  (sigma), (standard deviation), 372

""" (docstring delimiter), 80

\* repetition operator, 107

`\n`, newline character, 142

# start comment character, 21

+ concatenation operator, 27  
+ sequence method, 107

68-95-99.7 rule (empirical rule), 373

abs built-in function, 36

abstract data type. *See* data abstraction  
abstraction, 78

abstraction barrier, 178

acceleration due to gravity, 432

accuracy, of a classifier, 588

algorithm, 2

aliasing, 98

testing for, 151

al-Khwarizmi, Muhammad ibn Musa, 2

alpha ( $\alpha$ ), threshold for significance, 461

alternative hypothesis, 460

Anaconda, 138

Anaconda IDE, 13

annotate, Matplotlib plotting, 569

Anscombe, F.J., 497

append, list method, 97, 100, 620

approximate solution, 50

arange function, 450

arc of graph, 291

Archimedes, 404

arguments of function, 67

arm of a study, 475

array type, 265

operators on, 446

ASCII, 32  
assert statement, 176  
assertions, 176  
assignment statement, 19  
    multiple, 21, 91  
    mutation versus, 94  
    unpacking multiple returned values, 91  
attribute, 179  
AUC, 610  
AUROC, 610  
axhline, Matplotlib plotting, 349

Babbage, Charles, 488  
Bachelier, Louis, 322  
backtracking, 300  
bar chart, 491  
baseball, 389  
Bayes' Theorem, 484  
Bayesian statistics, 479  
bell curve, 372  
Bellman, Richard, 305  
Benford's law, 385  
Bernoulli trial, 380  
Bernoulli, Jacob, 348  
Bernoulli's theorem, 348  
BFS (breadth-first search), 302  
*Bible*, 404

big 6, 401

big O notation. *See* computational complexity

binary classification, 585

binary feature, 558

binary number, 57, 228, 344

binary search, 238

binary tree, 310

binding, of names to objects, 19

binomial coefficient, 380

binomial distribution, 379

bisection search algorithm, 54, 55

bisection search debugging technique, 162

bit, 57

black-box testing, 149–51, 151

block of code, 23

Boesky, Ivan, 290

Bonferroni correction, 479, 506

bool type, 17

Boolean expression, 18

- compound, 24
- short-circuit evaluation, 129

boolean operators quick reference, 619

Boston Marathon, 414, 476, 478, 591

Box, George E.P., 322

branching program, 21

breadth-first search (BFS), 302

break statement, 36, 37

Brown, Rita Mae, 160  
Brown, Robert, 322  
Brownian motion, 322  
Buffon, Comte de, 404  
bug, 156  
    covert, 157  
    intermittent, 157  
    origin of word, 156  
    overt, 157  
    persistent, 157  
built-in functions  
    abs, 36  
    help, 80  
    id, 96  
    input, 31  
    isinstance, 195  
    len, 28  
    map, 106  
    max, 66  
    range, 37  
    round, 59  
    sorted, 242, 248, 285  
    sum, 209  
    type, 17  
built-in functions on sequences, 93  
byte, 1

C++, 177  
calibration of a model, 610  
Canopy IDE, 13  
Cartesian coordinates, 323, 553  
case sensitivity in Python, 20  
case-fatality rate, 500  
categorical variable, 379, 613  
causal nondeterminism, 341  
Central Limit Theorem, 422–26, 430  
centroid, 564  
character, 27  
character encoding, 32  
cherry-picking, 478  
child node, 291  
Church, Alonzo, 68  
Church-Turing thesis, 5  
Chutes and Ladders, 338  
cipher, 118  
class  
    attribute, 179  
    class variable, 191  
    class, machine learning, 585, 603  
        class imbalance, 588  
classes  
    magic methods, 179  
    super, 191  
classes in Python, 177–212

`__init__` method, 179  
`__lt__` method, 190  
`__name__` method, 328  
`__str__` method, 184  
abstract, 207  
attribute, 180  
class variable, 180, 191  
data attribute, 180  
defining, 179  
definition, 179  
dot notation, 181  
inheritance, 190  
instance, 180  
instance variable, 180  
instantiation, 180  
`isinstance` function, 195  
`isinstance` vs. `type`, 195  
overriding attributes, 191  
printing instances, 184  
`self`, 181  
subclass, 191  
superclass, 191  
type hierarchy, 191  
type vs. `isinstance`, 195  
classification model, machine learning, 547, 585  
classifier, machine learning, 585  
client, of class or function, 78, 200

clique in graph theory, 297  
cloning, 101  
cloning a list, 101  
close method for files, 142  
CLT. *See* Central Limit Theorem  
CLU, 177  
clustering, 549, 561–83  
coding conventions, 41  
coefficient of determination ( $R^2$ ), 445–47  
coefficient of polynomial, 60  
coefficient of variation, 361–65, 364  
command. *See* statement  
comment in program, 21  
compiler, 11  
complexity. *See* computational complexity  
complexity classes, 221  
comprehension, dictionary, 118  
comprehension, list, 103, 607  
computation, 3  
computational complexity, 27, 213–31  
    amortized analysis, 241  
    asymptotic notation, 218  
    average-case, 215  
    best-case, 215  
    big O notation, 220  
    constant, 26, 221  
    expected-case, 215

- exponential, 221, 226
- inherently exponential, 290
- linear, 221, 223
- logarithmic, 221
- log-linear, 221, 224
- lower bound, 220
- polynomial, 221, 224
- pseudo-polynomial, 318
- quadratic, 224
- rules of thumb for expressing, 219
- tight bound, 220
- time-space tradeoff, 254, 402
- upper bound, 215, 220
- worst-case, 215

concatenation (+)

- append vs., 99
- lists, 99
- sequence types, 27
- tuples, 90

conceptual complexity, 213

conditional expressions, 26

conditional probability, 481

confidence interval, 365, 376, 400, 419, 424

confidence level, 419, 424

confusion matrix, 587

conjunct, 129

continuous probability distribution, 369, 370

continuous random variable, 370  
control c, 46  
control group, 457  
convenience sampling, 499  
Copenhagen Doctrine, 341  
copy standard library module, 102  
    copy.deepcopy, 102  
correlation, 495, 529, 550  
cosine similarity, 553  
count, list method, 100, 620  
count, str method, 109, 620  
craps (dice game), 396  
cross validation, 455  
crypto text, 118  
CSV file, 511  
curve fitting, 435  
data abstraction, 179, 323  
DataFrame (in Pandas), 511  
datetime standard library module, 187  
debuggers, 159  
debugging, 69, 137, 147, 156–66, 176  
    stochastic programs, 353  
declarative knowledge, 2  
decomposition, 78  
decrementing function, 46, 240  
deepcopy function, 102  
default parameter, 98

default parameter values, 71  
defensive programming, 158, 171, 176  
degree of belief, 484  
degree of polynomial, 60  
degrees of freedom, 462  
del, dict method, 621  
dental formula, 576  
depth-first search (DFS), 300  
destination node, 291  
deterministic program, 321, 342  
DFS (depth-first search), 300  
dict methods quick reference, 621  
dict type  
    comprehension, 118  
    deleting an element, 134  
    keys method, 116, 134  
    values method, 134  
    view, 116  
dict\_values, 116  
dictionary. *See* dict type  
dictionary comprehension, 118  
Dijkstra, Edsger, 148  
dimensionality, of data, 549  
discrete probability distribution, 369, 370  
discrete random variable, 369  
discrete uniform distribution, 379, *See also* distributions, uniform  
discrimination of a model, 610

disjunct, 129  
dispersion, 363  
dissimilarity metric, 562  
distributions, 355  
    Benford's, 385  
    empirical rule for normal, 425  
    exponential, 381  
    Gaussian (normal), 373  
    geometric, 384  
    normal, 406, *See also* normal distribution  
    rectangular, 378  
    uniform, 251, 379  
divide-and-conquer algorithms, 243, 319  
divide-and-conquer problem solving, 130  
docstring, 80  
*Don Quixote*, 119  
don't pass line, craps, 396  
dot notation, 86, 136, 183  
downsample, 597  
Dr. Pangloss, 147  
driver, in testing, 155  
dynamic programming, 305–19, 402  
dynamic-time-warping distance, 567  
e (Euler's number), 371  
earth-movers distance, 567  
edge of a graph, 291  
efficient programs. *See also* computational complexity

Einstein, Albert, 148, 322  
elastic limit of springs, 441  
elif, 24  
else, 23  
empirical rule, 373  
encapsulation, 200  
ENIAC, 393  
equality test  
    for objects, 96  
    value vs. object, 164  
error bars, 377  
escape character (\), 142  
Euclid, 384  
Euclidean distance, 554  
Euclidean mean, 562  
Euler, Leonhard, 291  
Euler's number ( $e$ ), 371  
exceptions, 167–75  
    built-in  
        AssertionError, 176  
        IndexError, 167  
        NameError, 167  
        TypeError, 167  
        ValueError, 167  
    built-in class, 174  
    except block, 169  
    raising, 167

try-except, 169  
unhandled, 168

exhaustive enumeration algorithms, 45, 47, 51, 281, 309  
square root algorithm, 51, 217

exponential decay, 382

exponential distribution, 381

exponential growth, 383

expression, 17

extend, list method, 99, 100, 620

factorial, 124, 216, 223  
iterative implementation, 124, 216  
recursive implementation, 124

false negative, 489, 552

false positive, 483, 485, 489, 552

family of hypotheses, 479

family-wise error rate, 479

feature vector, machine learning, 546

feature, machine learning  
engineering, 549  
scaling, 579  
selection, 550  
weight, 602

Fibonacci sequence, 125  
recursive implementation, 127

file system, 142

files, 134, 144

appending, 143  
close method, 142  
file handle, 142  
open function, 142  
reading, 143  
with statement, 142  
write method, 142  
writing, 142

find, str method, 109, 620  
first-class values, 84, 173  
Fisher, Ronald, 459  
fit method, logistic regression, 602  
fitting a curve to data, 435–43, 443  
    coefficient of determination ( $R^2$ ), 445–47  
    exponential with polyfit, 449  
    least-squares objective function, 436  
    linear regression, 437  
    objective function,, 435  
    overfitting, 441  
    polyfit, 436

fixed-program computers, 3  
flattening the curve, 279  
float type. *See* floating point  
floating point, 16, 18, 57, 56–60  
    conversion to int, 29  
    exponent, 57  
    internal representation, 57

precision, 58  
reals vs., 56  
rounded value, 58  
rounding errors, 60  
significant digits, 57  
test for equality (==), 59  
truncation of, 29

flow of control, 4  
flowchart, 4  
for loop, 37, 143  
formatted string literal, 30  
Franklin, Benjamin, 130  
frequency distribution, 369  
frequentist statistics, 479  
f-string, 30  
fully qualified names, 137  
function, 66, *See also* built-in functions  
    actual parameter, 67  
    argument, 67  
    as parameter, 247  
    call, 67  
    class as parameter, 327  
    default parameter values, 71  
    defining, 66  
    invocation, 67  
    keyword argument, 70, 71  
    positional parameter binding, 70

gambler's fallacy, 348

Gaussian distribution. *See* distributions, normal

generalization, 545, 547

generators, 203

geometric distribution, 384

geometric progression, 384

get, dict method, 621

glass-box testing, 152–53

global optimum, 290

global statement, 132

global variable, 132, 155

Gosset, William, 462

graph, 291

- adjacency list representation, 295
- adjacency matrix representation, 295
- breadth-first search (BFS), 302
- depth-first search (DFS), 300
- directed graph (digraph), 291
- edge, 291
- graph theory, 292
- node, 291

problems

- cliques, 297
- maximum clique, 297
- min cut, 297
- shortest path, 297
- shortest weighted path, 297

weighted, 292

Graunt, John, 487

gravity, acceleration due to, 432

greedy algorithm, 283–87

guess-and-check algorithms, 3, 47

Guinness, 462

halting problem, 5

hand simulation, 35

hand, in craps, 398

hashable, 111

hashable types, 184

hashing, 115, 250–54, 402

- collision, 251
- hash buckets, 251
- hash function, 250
- hash tables, 250
- probability of collisions, 386

help built-in function, 80

helper functions, 128, 239

Heron of Alexandria, 2

higher-order functions, 105

higher-order programming, 84, 105

histogram, 366

Hoare, C.A.R., 246

holdout set, 454, *See also* test set

Holmes, Sherlock, 165

Hooke's law, 431, 441  
Hopper, Grace Murray, 156  
hormone replacement therapy, 496  
housing prices, 490  
Huff, Darrell, 487, 510  
hypothesis testing, 459  
multiple hypotheses, 476–86

id built-in function, 96  
IDE (integrated development environment), 13  
IDLE IDE, 13  
if statement, 24  
image processing, 373  
immutable type, 94  
imperative knowledge, 2  
import \*, 137  
import statement, 136  
in operator, 40, 107  
in, dict method, 621  
indentation of code, 23  
independent events, 344  
index (of Pandas DataFrame), 512  
index, list method, 100, 620  
index, str method, 109, 620  
indexing for sequence types, 28  
indirection, 237  
induction, loop invariants, 242

inductive definition, 124  
infection-fatality rate, 500  
inferential statistics, 413  
information hiding, 200, 203  
    leading \_\_ in Python, 200  
input built-in function, 31  
input/output  
    quick reference, 621  
insert, list method, 100, 620  
instance, 180  
int type, 18  
int64, 513  
integrated development environment (IDE), 13  
integration, numeric, 373  
interface, 178  
interpreter, 4, 11  
*Introduction to Algorithms*, 233  
iPython console, 15  
iPython shell, 15  
is operator, 96  
isinstance built-in function, 195  
iterable, 92  
iteration, 33, 34  
    for loop, 143  
    generators, 203  
    over files, 143  
    tuples, 91

iterator type, 92  
Java, 177  
Julius Caesar, 130  
Kahneman, Daniel, 508  
Kennedy, Joseph, 164  
key, in dict, 112  
keys, dict method, 621  
keyword (reserved words) in Python, 20  
keyword argument, 70  
khemric, 494  
k-means clustering, 566–83  
knapsack problem, 282–90  
    0/1, 287  
    brute-force solution, 288  
    dynamic programming solution, 309–19  
    fractional (or continuous), 290  
k-nearest neighbors (KNN), 593–99  
Knight Capital Group, 158  
KNN (k-nearest neighbors), 593–99  
knowledge, declarative vs. imperative, 2  
Knuth, Donald, 220  
Königsberg bridges problem, 292  
label, machine learning, 585  
lambda abstraction, 68  
lambda expression, 85, 246  
Lampson, Butler, 237  
Laplace, Pierre-Simon, 404

latent variable, 548  
law of large numbers, 348, 352, 419  
leaf, of tree, 310  
least squares fit, 436, 439  
legend on plot, 269  
len built-in function, 107  
len, dict method, 621  
length (len), for sequence types, 28  
Leonardo of Pisa, 125  
lexical scoping, 74  
library, standard Python, 138, *See also* standard library modules  
LIFO, 76  
linear regression, 437, 546  
linear scaling, 580  
Liskov, Barbara, 197  
list comprehension, 103, 607  
list methods quick reference, 620  
list type, 93–100

- + (concatenation) operator, 99
- cloning, 101
- comprehension, 103
- copying, 101
- indexing, 235
- internal representation, 236

  
literal, in Python, 6, 16  
local optimum, 290  
local variable, 73

log function, 452  
logarithm, base of, 221  
logarithmic axis, 230  
logistic regression, 602–11, 614  
LogisticRegression class, 602  
loop invariant, 242  
lower, str method, 109, 620  
lurking variable, 495

machine code, 11  
machine learning  
    binary classification, 585  
    definition, 545  
    multi-class learning, 585  
    one-class learning, 585  
    supervised, 547  
    two-class learning, 585  
    unsupervised, 548

magic methods, 179  
Manhattan distance, 554  
Manhattan Project, 394  
many-to-one mapping, 251  
map built-in function, 106  
markeredgewith, 450  
math standard library module, 452  
MATLAB, 257, 567  
Matplotlib, 257, 511, *See also* plotting

max built-in function, 66  
maximum clique, 297  
memoization, 307  
memoryless property, 381  
merge sort, 224, 243–46, 305  
method, 86  
method invocation, 183  
min cut, 297  
Minkowski distance, 553, 558, 567  
min-max scaling, 580  
model, 321  
modules, 135–41, 135–41, 154  
modulus, 18  
Moksha-patamu, 338  
Molière, 178  
monotonic function, 86  
Monte Carlo simulation, 393–409, 393  
Monty Python, 13  
mortgages, 206, 263  
moving average, 536  
multi-class learning, 585  
multiple assignment, 21, 91  
    return values from functions, 92  
multiple hypotheses, 476–86, 478  
multiple hypothesis, 506  
multiplicative law for probabilities, 345  
mutable type, 94

mutation versus assignment, 94

n choose k, 380

name space, 73

names in Python, 20

nan (not a number), 170

nanosecond, 47

National Rifle Association, 500

natural number, 124

negative predictive value, 589

nested statements, 24

newline character (\n), 142

Newton's method. *See* Newton–Raphson method

Newtonian mechanics, 341

Newton–Raphson method, 60, 61, 235, 436

n-fold cross validation, 597

Nixon, Richard, 90

node, of a graph, 291

nominal variable. *See* categorical variable

nondeterminism, causal vs. predictive, 341

None, 17, 68, 173, 209

non-scalar type, 16, 89

normal distribution, 371–78, 406

standard, 579

not in, operator, 107

null hypothesis, 460

numeric operators, 18

quick reference, 619  
numeric types, 16  
numpy module, 265

O notation. *See* computational complexity  
O(1). *See* computational complexity, constant  
Obama, Barack, 123  
object, 16–19  
    class, 191  
    equality, 96  
    equality, vs. value equality, 164  
    first-class, 84  
    mutation, 94  
objective function, 435, 547, 562  
object-oriented programming, 177  
one-class learning, 585  
one-sample t-test, 469  
one-tailed t-test, 469  
open function, for files, 142  
operator precedence, 18  
operators, 17  
    %, on numbers, 18  
    \*\*, on numbers, 18  
    \*, on arrays, 265  
    \*, on numbers, 18  
    \*, on sequences, 107  
    -, on arrays, 265

- , on numbers, 18
- / , on numbers, 18
- // , on numbers, 18
- : slicing sequences, 29
- + , on numbers, 18
- + , on sequences, 107
- Boolean, 18
- floating point, 18
- in, on sequences, 107
- infix, 6
- integer, 18
- not in, on sequences, 107
- overloading, 27
  - optimal solution, 287
  - optimal substructure, 305, 314
  - optimization problem, 281, 435, 547, 562
    - constraints, 281
    - objective function, 281
  - order of growth, 220
  - outcomes, 602
  - overfitting, 441, 576, 588
  - overlapping subproblems, 305, 315
  - overloading of operators, 27
  - overriding, 191, *See also* classes in Python: class attributes
- packages, 138
- palindrome, 128

## Pandas

- ascending keyword argument, 540
- axis, 516
- Boolean indexing, 525
- comments in CSV files, 577
- concat, 517
- corr, 529
- correlation, 529
- DataFrame, 511
- iloc, 523
- isin, 526
- iterrows, 533
- itertuples, 534
- loc, 520
- logical operators, 525
- mean, 527
- NaN, 529
- read\_csv, 513
- read\_csv with comments, 577
- reset\_index, 518
- reset\_option, 531
- select column, 518
- series, 512
- Series, 519
- set\_index, 518
- set\_option, 530
- sort, ascending vs. descending, 540

sum, 527  
to\_string, 513  
parallel random access machine, 214  
parent node, 291  
Pascal, Blaise, 394  
pass line, craps, 396  
pass statement, 194  
path, 291  
paths through specification, 150  
PDF (probability density function), 370, 372  
Pearson correlation, 529  
PEP 8, 41, 67  
Peters, Tim, 247  
Pingala, 125  
Pirandello, 79  
plain text, 118  
plotting in Matplotlib, 258, 336, 366–71  
    annotate, 367, 569  
    axhline function, 349  
    bar chart, 491  
    current figure, 259  
    default settings, 262  
    figure function, 258  
    format string, 260  
    hatch keyword argument, 422  
    hist function, 366  
    histogram, 366

histogram scaling, 422  
keyword arguments, 261  
legend function, 269  
markeredgewidth, 450  
markers, 335  
plot function, 258  
rc settings, 262  
savefig function, 259  
sliders, 275  
style, 332  
table function, 555  
tables, 555  
text box, 272  
title function, 260  
vertical ticks, 352  
weights keyword argument, 422  
xlabel function, 260  
xlim function, 349  
xticks function, 491  
ylabel function, 260  
ylim function, 349  
yticks function, 491  
point of execution, 67  
point, in typography, 262  
pointer, 236  
polyfit, 436  
    fitting an exponential, 449

polymorphic functions and methods, 173, 190

polynomial, 60

  coefficient of, 60, 439

  degree of, 60, 436

polynomial regression, 437

polyval, 439

pop, list method, 100, 620

popping a stack, 76

population mean, 414

positive predictive value, 589

posterior, Bayesian, 484

power set, 288

predictive nondeterminism, 341

prime number, 48

print function, 15, 31

prior probability, 483, 595

prior, Bayesian, 484

private attributes of a class, 200

probability density function (PDF), 370, 372

probability distribution, 369

probability sampling, 413

probability, calculating, 344

program, 15

program counter, 4

programming language, 5, 11

  compiled, 11

  high-level, 11

interpreted, 11  
low-level, 11  
semantics, 7  
static semantics, 7  
syntax, 7  
prompt, shell, 17  
prospective study, 506  
pseudocode, 33, 566  
pseudorandom number generator, 353  
push, in gambling, 396  
p-value, 464–69, 464  
    misinterpretation of, 466  
Pythagorean theorem, 324, 406  
Python, 12, 67  
    shell, 15  
Python 3, versus 2.7, 353  
  
quad function, 373  
quantum mechanics, 341  
quicksort, 246  
  
 $R^2$  (coefficient of determination), 445–47  
rabbits, 125  
raise statement, 174  
random access machine, 214  
random module, 343  
random sampling, 499

random standard library module  
    **random.choice**, 327  
    **random.expovariate**, 383  
    **random.gauss**, 372  
    **random.random**, 319, 343  
    **random.sample**, 417  
    **random.seed**, 353  
    **random.uniform**, 379  
random variable, 369  
random walk, 323  
    biased, 331  
randomized trial, 457  
range built-in function, 37  
range type, 92  
rate of decay, 383  
recall (sensitivity), 589  
receiver operating characteristic curve (ROC), 610  
recurrence, 126  
recursion, 123–32  
    base case, 123  
    recursive (inductive) case, 123  
regression model, 547  
regression testing, 155  
regression to the mean, 349, 508  
regressive fallacy, 508  
rejection, of null hypothesis, 460  
remove, list method, 100, 620

repetition operator (\*), 27, 90  
replace, str method, 109, 620  
representation independence, 183  
representation invariant, 183  
reserved words in Python, 20  
retrospective study, 507  
return on investment (ROI), 399  
return statement, 67  
reverse argument for sort and sorted, 285  
reverse method, 100  
rfind, str method, 109, 620  
Rhind Papyrus, 403  
rindex, str method, 109, 620  
ROC curve (receiver operating characteristic curve), 610  
ROI (return on investment), 399  
root of polynomial, 60  
root of tree, 310  
round built-in function, 59  
R-squared (coefficient of determination), 445–47  
rstrip, str method, 109, 620

sample, 413  
sample function, 417  
sample mean, 414, 422  
sample standard deviation, 428  
sampling  
accuracy, 355

bias, 499  
confidence in, 357, 359  
convenience sampling, 499  
simple random, 413  
stratified, 413

Samuel, Arthur, 545

scalar type, 16

scaling features, 579

scientific method, 465

scipy library, 373

- scipy.integrate.quad, 373
- scipy.random.standard\_t, 463
- scipy.stats.ttest\_1samp, 470
- scipy.stats.ttest\_ind, 478

scoping, 73

- assignment to variable, impact of, 77
- lexical, 74
- static, 74

script, 15

SE. *See* standard error of the mean

search algorithms, 234–41

- binary search, 238, 240
- bisection search, 55
- breadth-first search (BFS), 302
- depth-first search (DFS), 300
- linear search, 215, 235

search space, 234

self, 181

SEM. *See* standard error of the mean

semantics, 7

sensitivity (recall), 589

seq, sequence method, 107

sequence methods quick reference, 619

sequence types, 28, *See also* str, tuple, list

sets, 110

    infix operators, 111

shallow copy, 101

shell, 15

    shell prompt, 17

shell., 15

short-circuit evaluation of Boolean expressions, 129

shortest path, 297

shortest weighted path, 297

side effect, 97, 99

signal processing, 373

signal-to-noise ratio (SNR), 549

significant digits, 57

simple random sample, 413

simulation, 321, 343

    coin flipping, 346–65

    continuous, 411

    deterministic, 410

    discrete, 411

    dynamic, 411

Monte Carlo, 393–409  
multiple trials, 347  
smoke test, 329  
static, 411  
stochastic, 410  
typical structure, 399  
simulation model, 321, *See also* simulation  
sklearn, 602  
    sklearn.metrics.auc, 610  
slicing for sequence types, 29  
sliders, 275  
SmallTalk, 177  
smoke test, 329  
snake oil, 509  
Snakes and Ladders, 338  
SNR (signal to noise ratio), 549  
social networks, 297  
software quality assurance (SQA), 154  
sort built-in method, 100, 620  
    key parameter, 249  
    mutates argument, 248  
    mutates list, 242  
    polymorphic, 190  
    reverse parameter, 249  
sorted built-in function, 285  
    does not mutate argument, 248  
    key parameter, 249

- returns a list, 242
- reverse parameter, 249
- sorting algorithms, 241–49
  - in-place, 246
  - merge sort, 224, 243–46, 305
  - quicksort, 246
  - stable, 249
  - timsort, 247
- source code, 11
- source node, 291
- space complexity, 224, 246
- specification
  - assumptions, 78, 238
  - docstring, 80
  - guarantees, 78
- specificity (precision), 589
- split, str method, 109, 247, 620
- spring constant, 431
- SQA (software quality assurance), 154
- square root, 50, 51, 54, 61
- stable sort, 249
- stack, 76
- stack frame, 74
- standard deviation, 356, 400
  - relative to mean, 361
- standard error. *See* standard error of the mean
- standard error of the mean (SE, SEM), 427–30, 461

standard library modules

copy, 102

datetime, 187

math, 452

random, 343

standard normal distribution, 579

statement, 15

statements

= (assignment), 19

assert, 176

break, 37

conditional, 22

for loop, 37, 143

global, 132

if, 24

import, 136

import \*, 137

pass, 194

raise, 174

return, 67

try-except, 169

yield, 204

static scoping, 74, *See also* scoping

static semantic checking, 7, 203

statistical machine learning. *See* machine learning

statistical significance, 458–79, 459, 506

correctness versus, 409

## statistical sin

- assuming independence, 488
- confusing correlation and causation, 495
- convenience (accidental) sampling, 499
- Cum Hoc Ergo Propter Hoc*, 494
- extrapolation, 502
- Garbage In Garbage Out (GIGO), 488
- non-response bias, 499
- reliance on measures, 497
- Texas sharpshooter fallacy, 503

## statistics

- alternative hypothesis, 460
- coefficient of variation, 361–65, 364, 365
- confidence interval, 365, 376, 425
- confidence interval, overlapping, 400
- confidence level, 376
- correctness vs. statistical validity, 408
- correlation, 495
- error bars, 377
- frequentist approach, 469
- null hypothesis, 460
- sample standard deviation, 428
- significance, 458–79
- standard error of the mean, 427
- t-distribution, 429, 462
- test statistic, 460
- $\alpha$  (alpha), 461

## stats module

stats.ttest\_1samp, 510

stats.ttest\_ind, 465

step (of a computation), 214

stochastic process, 321

stochastic programs, 343

stored-program computer, 4

## str

built-in methods, 108

character as, 27

concatenation (+), 27

escape character, \, 192

f-string, 30

indexing, 28

len, 28

newline character (\n), 142

slicing, 29

split method, 247, 620

substring, 29

str methods quick reference, 620

straight-line programs, 21

stratified sampling, 413

string type. *See str*

stub, in testing, 155

Student's t-distribution, 462

study power, 468

substitution principle, 197, 295

substring, 29  
successive approximation, 56, 436  
sum builtin function, 93  
sum built-in function, 209  
super, 191  
supervised learning, 547, 585  
support, Bayesian, 484  
switch, 57  
symbol table, 73, 136  
syntax, 7

table function, Matplotlib, 555  
table lookup, 307  
tables, in Matplotlib, 555  
t-distribution, 429, 462  
tea test, 459  
termination  
    of loop, 35, 46  
    of recursion, 240  
test set, 454, 507, 586  
test statistic, 460  
testing, 147, 148–55  
    black-box, 149–51, 151  
    boundary conditions, 150  
    driver, 155  
    glass-box, 152–53, 152–53  
    integration testing, 154

partitioning inputs, 149  
path-complete, 152  
regression testing, 155  
stub, 155  
test functions, 69  
test suite, 148  
unit testing, 154

Texas sharpshooter fallacy, 503

Thanksgiving, 140

timsort, 247

*Titanic*, 612

total ordering, 53

training error, 586

training set (training data), 454, 507, 546, 586, 591

translating text, 113

treatment effect, 508

treatment group, 457

tree, 310

- enumeration, left-first depth-first, 311
- leaf node, 310
- root, 310
- rooted binary tree, 310

triangle inequality, 553

true negative, 587

true positive, 483

try block, 169

try-except, 168

try-except statement, 169  
t-statistic, 461  
t-test, 465  
tuple, 37  
tuple type, 89  
Turing complete, 63  
Turing completeness, 6  
Turing machine, universal, 5  
two-class learning, 585  
type, 16, 177  
    immutable, 94  
    mutable, 94  
type built-in function, 17  
type cast, 29  
type checking, 28  
type conversion, 29  
    list to array, 265  
type I error, 461  
type II error, 461  
type type, 179  
types  
    array, 265  
    bool, 17  
    dict. *See* dict type  
    float, 16, *See also* floating point  
    int, 16  
    list. *See* list type

None, 17  
range, 92  
str. *See* str  
tuple, 89  
type, 179  
view, 116

U.S. citizen, definition of natural-born, 123  
Ulam, Stanislaw, 393  
unary function, 106  
uniform distribution, 251, 378, 379  
unpacking operator, 72  
unsupervised learning, 548  
utf-8, 32  
value, 17  
value equality vs. object equality, 164  
values, dict method, 621  
variability, of a cluster, 562  
variable, 19  
    choosing a name, 20  
variance, 355, 562  
versions of Python, 12  
vertex of a graph, 291  
view type, 116  
von Neumann, John, 244  
von Rossum, Guido, 12

white-box testing, 149

whitespace characters, 110  
Wing, Jeannette, 197  
word size, 236  
Words with Friends game, 471  
World Series, 389  
wrapper functions, 239  
write method for files, 142

xlim, 349  
xticks, 491

yield statement, 204

ylim, 349  
yticks, 491

zero-based indexing, 28  
z-scaling, 579