GLOBAL EDITION

# The Practice of Computing Using Python

THIRD EDITION

Punch • Enbody

Pearson

# Digital Resources for Students

Your new textbook provides 12-month access to digital resources that may include VideoNotes (step-by-step video tutorials on programming concepts), source code, web chapters, quizzes, and more. Refer to the preface in the textbook for a detailed list of resources.

Follow the instructions below to register for the Companion Website for William Punch and Richard Enbody's *The Practice of Computing Using Python*, Third Edition, Global Edition.

1. Go to www.pearsonglobaleditions.com/punch
2. Enter the title of your textbook
3. Click Companion Website
4. Click Register and follow the on-screen instructions to create a login name and password.

**Use a coin to scratch off the coating and reveal your access code.**
**Do not use a sharp knife or other sharp object as it may damage the code.**

Use the login name and password you created during registration to start using the digital resources that accompany your textbook.

## IMPORTANT:

This prepaid subscription does not include access to MyProgammingLab, which is available at **www.myprogramminglab.com** for purchase.

*This access code can only be used once. This subscription is valid for 12 months upon activation and is not transferrable. If the access code has already been revealed it may no longer be valid.*

For technical support go to https://support.pearson.com/getsupport

This page intentionally left blank

# THE PRACTICE OF COMPUTING USING PYTHON

THIRD EDITION

GLOBAL EDITION

William Punch

Richard Enbody


Boston Columbus Indianapolis New York San Francisco Hoboken
Amsterdam Cape Town Dubai London Madrid Milan Munich Paris Montréal Toronto
Delhi Mexico City São Paulo Sydney Hong Kong Seoul Singapore Taipei Tokyo

Vice President, Editorial Director, ECS: Marcia Horton
Acquisitions Editor: Matt Goldstein
Editorial Assistant: Kristy Alaura
Acquistions Editor, Global Editions: Murchana Borthakur
Vice President of Marketing: Christy Lesko
Director of Field Marketing: Tim Galligan
Product Marketing Manager: Bram Van Kempen
Field Marketing Manager: Demetrius Hall
Marketing Assistant: Jon Bryant
Director of Product Management: Erin Gregg
Team Lead, Program and Project Management: Scott Disanno
Program Manager: Carole Snyder
Project Editor, Global Editions: K.K. Neelakantan
Senior Manufacturing Controller, Global Editions: Jerry Kataria
Senior Specialist, Program Planning and Support: Maura Zaldivar-Garcia
Cover Designer: Lumina Datamatics
Manager, Rights and Permissions: Rachel Youdelman
Project Manager, Rights and Permissions: William Opaluch
Inventory Manager: Meredith Maresca
Media Project Manager: Dario Wong
Media Production Manager, Global Editions: M Vikram Kumar
Full-Service Project Management: Jogender Taneja, iEnerziger Aptara®, Ltd.
Cover Photo Credit: Marietjie/Shutterstock

Credits and acknowledgments borrowed from other sources and reproduced, with permission, in this textbook appear on appropriate page within text. Reprinted with permission.



---

Pearson Education Limited
Edinburgh Gate
Harlow
Essex CM20 2JE
England

and Associated Companies throughout the world

Visit us on the World Wide Web at:
www.pearsonglobaleditions.com



*Authorized adaptation from the United States edition, entitled The Practice of Computing Using Python, 3rd Edition, ISBN 978-0-13-437976-0, by William F. Punch and Richard J. Enbody published by Pearson Education © 2017.*



British Library Cataloguing-in-Publication Data

A catalogue record for this book is available from the British Library

10 9 8 7 6 5 4 3 2 1

ISBN 10: 1-292-16662-2
ISBN 13: 978-1-292-16662-9
Typeset by iEnerziger Aptara®, Ltd.
Printed and bound in Malaysia

*To our beautiful wives Laurie and Wendy and our kids Zach, Alex,*
*Abby, Carina, and Erik,*
*and our parents.*
*We love you and couldn't have done this*
*without your love and support.*

This page intentionally left blank

# BRIEF CONTENTS


VIDEONOTES 24
PREFACE 25
PREFACE TO THE SECOND EDITION 29

PART 1 THINKING ABOUT COMPUTING 33

Chapter 0 The Study of Computer Science 35

PART 2 STARTING TO PROGRAM 67

Chapter 1 Beginnings 69
Chapter 2 Control 119
Chapter 3 Algorithms and Program Development 193

PART 3 DATA STRUCTURES AND FUNCTIONS 219

Chapter 4 Working with Strings 221
Chapter 5 Functions—QuickStart 277
Chapter 6 Files and Exceptions I 303
Chapter 7 Lists and Tuples 343
Chapter 8 More on Functions 427
Chapter 9 Dictionaries and Sets 455
Chapter 10 More Program Development 515

PART 4 CLASSES, MAKING YOUR OWN DATA STRUCTURES AND ALGORITHMS 559

Chapter 11 Introduction to Classes 561
Chapter 12 More on Classes 603
Chapter 13 Program Development with Classes 647

**PART 5** BEING A BETTER PROGRAMMER **675**

**Chapter 14** Files and Exceptions II **677**
**Chapter 15** Recursion: Another Control Mechanism **719**
**Chapter 16** Other Fun Stuff with Python **741**
**Chapter 17** The End, or Perhaps the Beginning **783**

**APPENDICES 785**

**Appendix A** Getting and Using Python **785**
**Appendix B** Simple Drawing with Turtle Graphics **805**
**Appendix C** What's Wrong with My Code? **817**
**Appendix D** Pylab: A Plotting and Numeric Tool **849**
**Appendix E** Quick Introduction to Web-based User Interfaces **861**
**Appendix F** Table of UTF-8 One Byte Encodings **891**
**Appendix G** Precedence **893**
**Appendix H** Naming Conventions **895**
**Appendix I** Check Yourself Solutions **899**

**INDEX 905**

# CONTENTS


VIDEONOTES 24
PREFACE 25
PREFACE TO THE SECOND EDITION 29

1.0.1 Data Manipulation 30
1.0.2 Problem Solving and Case Studies 30
1.0.3 Code Examples 30
1.0.4 Interactive Sessions 31
1.0.5 Exercises and Programming Projects 31
1.0.6 Self-Test Exercises 31
1.0.7 Programming Tips 31

PART 1 THINKING ABOUT COMPUTING 33

Chapter 0 The Study of Computer Science 35

0.1 Why Computer Science? 35
0.1.1 Importance of Computer Science 35
0.1.2 Computer Science Around You 36
0.1.3 Computer "Science" 36
0.1.4 Computer Science Through Computer Programming 38

0.2 The Difficulty and Promise of Programming 38
0.2.1 Difficulty 1: Two Things at Once 38
0.2.2 Difficulty 2: What Is a Good Program? 41
0.2.3 The Promise of a Computer Program 42

0.3 Choosing a Computer Language 43
0.3.1 Different Computer Languages 43
0.3.2 Why Python? 43
0.3.3 Is Python the Best Language? 45

0.4 What Is Computation? 45

0.5 What Is a Computer? 45

0.5.1 Computation in Nature 46
0.5.2 The Human Computer 49
0.6 The Modern, Electronic Computer 50
0.6.1 It's the Switch! 50
0.6.2 The Transistor 51
0.7 A High-Level Look at a Modern Computer 56
0.8 Representing Data 58
0.8.1 Binary Data 58
0.8.2 Working with Binary 59
0.8.3 Limits 60
0.8.4 Representing Letters 61
0.8.5 Representing Other Data 62
0.8.6 What Does a Number Represent? 63
0.8.7 How to Talk About Quantities of Data 64
0.8.8 How Much Data Is That? 64
0.9 Overview of Coming Chapters 66

PART 2 STARTING TO PROGRAM 67

Chapter 1 Beginnings 69
1.1 Practice, Practice, Practice 69
1.2 QuickStart, the Circumference Program 70
1.2.1 Examining the Code 72
1.3 An Interactive Session 74
1.4 Parts of a Program 75
1.4.1 Modules 75
1.4.2 Statements and Expressions 75
1.4.3 Whitespace 77
1.4.4 Comments 78
1.4.5 Special Python Elements: Tokens 78
1.4.6 Naming Objects 80
1.4.7 Recommendations on Naming 81
1.5 Variables 81
1.5.1 Variable Creation and Assignment 82
1.6 Objects and Types 85
1.6.1 Numbers 87
1.6.2 Other Built-In Types 89
1.6.3 Object Types: Not Variable Types 90
1.6.4 Constructing New Values 92

1.7 Operators 93
1.7.1 Integer Operators 93
1.7.2 Floating-Point Operators 96
1.7.3 Mixed Operations 96
1.7.4 Order of Operations and Parentheses 97
1.7.5 Augmented Assignment Operators: A Shortcut! 98
1.8 Your First Module, Math 100
1.9 Developing an Algorithm 101
1.9.1 New Rule—Testing 105
1.10 Visual Vignette: Turtle Graphics 106
1.11 What's Wrong with My Code? 107
Chapter 2 Control 119
2.1 QuickStart Control 119
2.1.1 Selection 119
2.1.2 Booleans for Decisions 121
2.1.3 The *if* Statement 121
2.1.4 Example: What Lead Is Safe in Basketball? 124
2.1.5 Repetition 128
2.1.6 Example: Finding Perfect Numbers 132
2.1.7 Example: Classifying Numbers 137
2.2 In-Depth Control 141
2.2.1 *True* and *False*: Booleans 141
2.2.2 Boolean Variables 142
2.2.3 Relational Operators 142
2.2.4 Boolean Operators 147
2.2.5 Precedence 148
2.2.6 Boolean Operators Example 149
2.2.7 Another Word on Assignments 152
2.2.8 The Selection Statement for Decisions 154
2.2.9 More on Python Decision Statements 154
2.2.10 Repetition: the *while* Statement 158
2.2.11 Sentinel Loop 168
2.2.12 Summary of Repetition 168
2.2.13 More on the *for* Statement 169
2.2.14 Nesting 172
2.2.15 Hailstone Sequence Example 174
2.3 Visual Vignette: Plotting Data with Pylab 175
2.3.1 First Plot and Using a List 176
2.3.2 More Interesting Plot: A Sine Wave 177

2.4 Computer Science Perspectives: Minimal Universal Computing 179
2.4.1 Minimal Universal Computing 179
2.5 What's Wrong with My Code? 180
Chapter 3 Algorithms and Program Development 193
3.1 What Is an Algorithm? 193
3.1.1 Example Algorithms 194
3.2 Algorithm Features 195
3.2.1 Algorithm versus Program 195
3.2.2 Qualities of an Algorithm 197
3.2.3 Can We Really Do All That? 199
3.3 What Is a Program? 199
3.3.1 Readability 199
3.3.2 Robust 203
3.3.3 Correctness 204
3.4 Strategies for Program Design 205
3.4.1 Engage and Commit 205
3.4.2 Understand, Then Visualize 206
3.4.3 Think Before You Program 207
3.4.4 Experiment 207
3.4.5 Simplify 207
3.4.6 Stop and Think 209
3.4.7 Relax: Give Yourself a Break 209
3.5 A Simple Example 209
3.5.1 Build the Skeleton 210
3.5.2 Output 210
3.5.3 Input 211
3.5.4 Doing the Calculation 213
PART 3 DATA STRUCTURES AND FUNCTIONS 219
Chapter 4 Working with Strings 221
4.1 The String Type 222
4.1.1 The Triple-Quote String 222
4.1.2 Nonprinting Characters 223
4.1.3 String Representation 223
4.1.4 Strings as a Sequence 224
4.1.5 More Indexing and Slicing 225
4.1.6 Strings Are Iterable 230

4.2 String Operations 231
4.2.1 Concatenation (+) and Repetition (*) 231
4.2.2 Determining When + Indicates Addition or Concatenation? 232
4.2.3 Comparison Operators 233
4.2.4 The *in* Operator 234
4.2.5 String Collections Are Immutable 235
4.3 A Preview of Functions and Methods 237
4.3.1 A String Method 237
4.3.2 Determining Method Names and Method Arguments 240
4.3.3 String Methods 242
4.3.4 String Functions 242
4.4 Formatted Output for Strings 243
4.4.1 Descriptor Codes 244
4.4.2 Width and Alignment Descriptors 245
4.4.3 Floating-Point Precision Descriptor 246
4.5 Control and Strings 247
4.6 Working with Strings 250
4.6.1 Example: Reordering a Person's Name 250
4.6.2 Palindromes 252
4.7 More String Formatting 255
4.8 Unicode 258
4.9 A GUI to Check a Palindrome 260
4.10 What's Wrong with My Code? 264
Chapter 5 Functions—QuickStart 277
5.1 What Is a Function? 277
5.1.1 Why Have Functions? 278
5.2 Python Functions 279
5.3 Flow of Control with Functions 282
5.3.1 Function Flow in Detail 283
5.3.2 Parameter Passing 283
5.3.3 Another Function Example 285
5.3.4 Function Example: Area of a Triangle 286
5.3.5 Functions Calling Functions 290
5.3.6 When to Use a Function 291
5.3.7 What If There Is No Return Statement? 292
5.3.8 What If There Are Multiple Return Statements? 292

5.4 Visual Vignette: Turtle Flag 293
5.5 What's Wrong with My Code? 294
Chapter 6 Files and Exceptions I 303
6.1 What Is a File? 303
6.2 Accessing Files: Reading Text Files 303
6.2.1 What's Really Happening? 304
6.3 Accessing Files: Writing Text Files 305
6.4 Reading and Writing Text Files in a Program 306
6.5 File Creation and Overwriting 307
6.5.1 Files and Functions Example: Word Puzzle 308
6.6 First Cut, Handling Errors 314
6.6.1 Error Names 315
6.6.2 The `try-except` Construct 315
6.6.3 `try-except` Flow of Control 316
6.6.4 Exception Example 317
6.7 Example: Counting Poker Hands 320
6.7.1 Program to Count Poker Hands 323
6.8 GUI to Count Poker Hands 331
6.8.1 Count Hands Function 332
6.8.2 The Rest of the GUI Code 334
6.9 Error Check Float Input 336
6.10 What's Wrong with My Code? 336
Chapter 7 Lists and Tuples 343
7.1 What Is a List? 343
7.2 What You Already Know How To Do With Lists 345
7.2.1 Indexing and Slicing 346
7.2.2 Operators 347
7.2.3 Functions 349
7.2.4 List Iteration 350
7.3 Lists Are Different than Strings 351
7.3.1 Lists Are Mutable 351
7.3.2 List Methods 352
7.4 Old and New Friends: Split and Other Functions and Methods 357
7.4.1 Split and Multiple Assignment 357
7.4.2 List to String and Back Again, Using `join` 358
7.4.3 The `Sorted` Function 359

7.5 Working with Some Examples 360
7.5.1 Anagrams 360
7.5.2 Example: File Analysis 366
7.6 Mutable Objects and References 372
7.6.1 Shallow versus Deep Copy 377
7.6.2 Mutable versus Immutable 381
7.7 Tuples 382
7.7.1 Tuples from Lists 384
7.7.2 Why Tuples? 385
7.8 Lists: The Data Structure 385
7.8.1 Example Data Structure 386
7.8.2 Other Example Data Structures 387
7.9 Algorithm Example: U.S. EPA Automobile Mileage Data 387
7.9.1 CSV Module 397
7.10 Visual Vignette: Plotting EPA Data 398
7.11 List Comprehension 400
7.11.1 Comprehensions, Expressions, and the Ternary Operator 402
7.12 Visual Vignette: More Plotting 402
7.12.1 Pylab Arrays 403
7.12.2 Plotting Trigonometric Functions 405
7.13 GUI to Find Anagrams 406
7.13.1 Function Model 406
7.13.2 Controller 407
7.14 What's Wrong with My Code? 409

Chapter 8 More on Functions 427

8.1 Scope 427
8.1.1 Arguments, Parameters, and Namespaces 429
8.1.2 Passing Mutable Objects 431
8.1.3 Returning a Complex Object 433
8.1.4 Refactoring `evens` 435
8.2 Default Values and Parameters as Keywords 436
8.2.1 Example: Default Values and Parameter Keywords 437
8.3 Functions as Objects 439
8.3.1 Function Annotations 440
8.3.2 Docstrings 441

8.4 Example: Determining a Final Grade 442
8.4.1 The Data 442
8.4.2 The Design 442
8.4.3 Function: *weighted_grade* 443
8.4.4 Function: *parse_line* 443
8.4.5 Function: *main* 444
8.4.6 Example Use 445
8.5 Pass "by Value" or "by Reference" 445
8.6 What's Wrong with My Code? 446

**Chapter 9** Dictionaries and Sets 455

9.1 Dictionaries 455
9.1.1 Dictionary Example 456
9.1.2 Python Dictionaries 457
9.1.3 Dictionary Indexing and Assignment 457
9.1.4 Operators 458
9.1.5 Ordered Dictionaries 463
9.2 Word Count Example 464
9.2.1 Count Words in a String 464
9.2.2 Word Frequency for Gettysburg Address 465
9.2.3 Output and Comments 469
9.3 Periodic Table Example 470
9.3.1 Working with CSV Files 471
9.3.2 Algorithm Overview 473
9.3.3 Functions for Divide and Conquer 473
9.4 Sets 477
9.4.1 History 477
9.4.2 What's in a Set? 477
9.4.3 Python Sets 478
9.4.4 Methods, Operators, and Functions for Python Sets 479
9.4.5 Set Methods 479
9.5 Set Applications 484
9.5.1 Relationship between Words of Different 484
9.5.2 Output and Comments 488
9.6 Scope: The Full Story 488
9.6.1 Namespaces and Scope 489
9.6.2 Search Rule for Scope 489
9.6.3 Local 489
9.6.4 Global 490
9.6.5 Built-Ins 494
9.6.6 Enclosed 495

9.7 Using `zip` to Create Dictionaries 496
9.8 Dictionary and Set Comprehensions 497
9.9 Visual Vignette: Bar Graph of Word Frequency 498
9.9.1 Getting the Data Right 498
9.9.2 Labels and the `xticks` Command 499
9.9.3 Plotting 499
9.10 GUI to Compare Files 500
9.10.1 Controller and View 501
9.10.2 Function Model 503
9.11 What's Wrong with My Code? 505
Chapter 10 More Program Development 515
10.1 Introduction 515
10.2 Divide and Conquer 515
10.2.1 Top-Down Refinement 516
10.3 The Breast Cancer Classifier 516
10.3.1 The Problem 516
10.3.2 The Approach: Classification 517
10.3.3 Training and Testing the Classifier 517
10.3.4 Building the Classifier 517
10.4 Designing the Classifier Algorithm 519
10.4.1 Divided, now Conquer 522
10.4.2 Data Structures 523
10.4.3 File Format 523
10.4.4 The `make_training_set` Function 524
10.4.5 The `make_test_set` Function 528
10.4.6 The `train_classifier` Function 529
10.4.7 `train_classifier`, Round 2 531
10.4.8 Testing the Classifier on New Data 534
10.4.9 The `report_results` Function 538
10.5 Running the Classifier on Full Data 540
10.5.1 Training versus Testing 540
10.6 Other Interesting Problems 544
10.6.1 Tag Clouds 544
10.6.2 S&P 500 Predictions 546
10.6.3 Predicting Religion with Flags 549
10.7 GUI to Plot the Stock Market 551
10.7.1 Function Model 551
10.7.2 Controller and View 553

**PART 4** CLASSES, MAKING YOUR OWN DATA STRUCTURES AND ALGORITHMS **559**

**Chapter 11** Introduction to Classes **561**

**11.1** QuickStart: Simple Student Class **561**

**11.2** Object-Oriented Programming **562**
**11.2.1** Python Is Object-Oriented! **562**
**11.2.2** Characteristics of OOP **563**

**11.3** Working with OOP **563**
**11.3.1** Class and Instance **563**

**11.4** Working with Classes and Instances **564**
**11.4.1** Built-In Class and Instance **564**
**11.4.2** Our First Class **566**
**11.4.3** Changing Attributes **568**
**11.4.4** The Special Relationship Between an Instance and Class: instance-of **569**

**11.5** Object Methods **572**
**11.5.1** Using Object Methods **572**
**11.5.2** Writing Methods **573**
**11.5.3** The Special Argument `self` **574**
**11.5.4** Methods Are the Interface to a Class Instance **576**

**11.6** Fitting into the Python Class Model **577**
**11.6.1** Making Programmer-Defined Classes **577**
**11.6.2** A Student Class **577**
**11.6.3** Python Standard Methods **578**
**11.6.4** Now There Are Three: Class Designer, Programmer, and User **582**

**11.7** Example: Point Class **583**
**11.7.1** Construction **585**
**11.7.2** Distance **585**
**11.7.3** Summing Two Points **585**
**11.7.4** Improving the Point Class **586**

**11.8** Python and OOP **590**
**11.8.1** Encapsulation **590**
**11.8.2** Inheritance **591**
**11.8.3** Polymorphism **591**

**11.9** Python and Other OOP Languages **591**
**11.9.1** Public versus Private **591**
**11.9.2** Indicating Privacy Using Double Underscores (__) **592**

11.9.3 Python's Philosophy 593
11.9.4 Modifying an Instance 594
11.10 What's Wrong with My Code? 594

Chapter 12 More on Classes 603

12.1 More About Class Properties 603
12.1.1 Rational Number (Fraction) Class Example 604
12.2 How Does Python Know? 606
12.2.1 Classes, Types, and Introspection 606
12.2.2 Remember Operator Overloading 609
12.3 Creating Your Own Operator Overloading 609
12.3.1 Mapping Operators to Special Methods 610
12.4 Building the Rational Number Class 613
12.4.1 Making the Class 613
12.4.2 Review Fraction Addition 615
12.4.3 Back to Adding Fractions 618
12.4.4 Equality and Reducing Rationals 622
12.4.5 Divide and Conquer at Work 625
12.5 What Doesn't Work (Yet) 625
12.5.1 Introspection 626
12.5.2 Repairing `int + Rational` Errors 628
12.6 Inheritance 630
12.6.1 The "Find the Attribute" Game 631
12.6.2 Using Inheritance 634
12.6.3 Example: The Standard Model 635
12.7 What's Wrong with My Code? 640

Chapter 13 Program Development with Classes 647

13.1 Predator–Prey Problem 647
13.1.1 The Rules 648
13.1.2 Simulation Using Object-Oriented Programming 649
13.2 Classes 649
13.2.1 *Island* Class 649
13.2.2 Predator and Prey, Kinds of Animals 651
13.2.3 `Predator` and `Prey` Classes 654
13.2.4 Object Diagram 655
13.2.5 Filling the Island 655
13.3 Adding Behavior 658
13.3.1 Refinement: Add Movement 658
13.3.2 Refinement: Time Simulation Loop 661

13.4 Refinement: Eating, Breeding, and Keeping Time 662
13.4.1 Improved Time Loop 663
13.4.2 Breeding 666
13.4.3 Eating 668
13.4.4 The Tick of the Clock 669
13.5 Refinement: How Many Times to Move? 670
13.6 Visual Vignette: Graphing Population Size 671

PART 5 BEING A BETTER PROGRAMMER 675

Chapter 14 Files and Exceptions II 677

14.1 More Details on Files 677
14.1.1 Other File Access Methods, Reading 679
14.1.2 Other File Access Methods, Writing 681
14.1.3 Universal New Line Format 683
14.1.4 Moving Around in a File 684
14.1.5 Closing a File 686
14.1.6 The *with* Statement 686
14.1.7 Text File Encodings; Unicode 687
14.2 CSV Files 688
14.2.1 CSV Module 689
14.2.2 CSV Reader 690
14.2.3 CSV Writer 691
14.2.4 Example: Update Some Grades 691
14.3 Module: *os* 693
14.3.1 Directory (Folder) Structure 694
14.3.2 *os* Module Functions 695
14.3.3 *os* Module Example 697
14.4 More on Exceptions 699
14.4.1 Basic Exception Handling 700
14.4.2 A Simple Example 701
14.4.3 Events 703
14.4.4 A Philosophy Concerning Exceptions 704
14.5 Exception: *else* and *finally* 705
14.5.1 *finally* and *with* 705
14.5.2 Example: Refactoring the Reprompting of a File Name 705
14.6 More on Exceptions 707
14.6.1 *Raise* 707
14.6.2 Create Your Own 708
14.7 Example: Password Manager 709

**Chapter 15** Recursion: Another Control Mechanism **719**

**15.1** What Is Recursion? **719**

**15.2** Mathematics and Rabbits **721**

**15.3** Let's Write Our Own: Reversing a String **724**

**15.4** How Does Recursion Actually Work? **726**
**15.4.1** Stack Data Structure **727**
**15.4.2** Stacks and Function Calls **729**
**15.4.3** A Better Fibonacci **731**

**15.5** Recursion in Figures **732**
**15.5.1** Recursive Tree **732**
**15.5.2** Sierpinski Triangles **734**

**15.6** Recursion to Non-recursion **735**

**15.7** GUI for Turtle Drawing **736**
**15.7.1** Using Turtle Graphics to Draw **736**
**15.7.2** Function Model **737**
**15.7.3** Controller and View **738**

**Chapter 16** Other Fun Stuff with Python **741**

**16.1** Numbers **741**
**16.1.1** Fractions **742**
**16.1.2** Decimal **746**
**16.1.3** Complex Numbers **750**
**16.1.4** Statistics Module **752**
**16.1.5** Random Numbers **754**

**16.2** Even More on Functions **756**
**16.2.1** Having a Varying Number of Parameters **757**
**16.2.2** Iterators and Generators **760**
**16.2.3** Other Functional Programming Ideas **765**
**16.2.4** Some Functional Programming Tools **766**
**16.2.5** Decorators: Functions Calling Functions **768**

**16.3** Classes **773**
**16.3.1** Properties **774**
**16.3.2** Serializing an Instance: `pickle` **777**

**16.4** Other Things in Python **780**
**16.4.1** Data Types **780**
**16.4.2** Built-in Modules **780**
**16.4.3** Modules on the Internet **781**

**Chapter 17** The End, or Perhaps the Beginning **783**

## APPENDICES 785


**Appendix A** Getting and Using Python **785**

- **A.1** About Python **785**
  - **A.1.1** History **785**
  - **A.1.2** Python 3 **785**
  - **A.1.3** Python Is Free and Portable **786**
  - **A.1.4** Installing Anaconda **788**
  - **A.1.5** Starting Our Python IDE: SPYDER **788**
  - **A.1.6** Working with Python **789**
  - **A.1.7** Making a Program **792**
- **A.2** The IPython Console **794**
  - **A.2.1** Anatomy of an iPython Session **795**
  - **A.2.2** Your Top Three iPython Tips **796**
  - **A.2.3** Completion and the Tab Key **796**
  - **A.2.4** The ? Character **798**
  - **A.2.5** More iPython Tips **798**
- **A.3** Some Conventions for This Book **801**
  - **A.3.1** Interactive Code **802**
  - **A.3.2** Program: Written Code **802**
  - **A.3.3** Combined Program and Output **802**
- **A.4** Summary **803**

**Appendix B** Simple Drawing with Turtle Graphics **805**

- **B.0.1** What Is a Turtle? **805**
- **B.0.2** Motion **807**
- **B.0.3** Drawing **807**
- **B.0.4** Color **809**
- **B.0.5** Drawing with Color **811**
- **B.0.6** Other Commands **813**
- **B.1** Tidbits **815**
  - **B.1.1** Reset/Close the Turtle Window **815**

**Appendix C** What's Wrong with My Code? **817**

- **C.1** It's Your Fault! **817**
  - **C.1.1** Kinds of Errors **817**
  - **C.1.2** "Bugs" and Debugging **819**
- **C.2** Debugging **821**
  - **C.2.1** Testing for Correctness **821**
  - **C.2.2** Probes **821**
  - **C.2.3** Debugging with SPYDER Example 1 **821**
  - **C.2.4** Debugging Example 1 Using `print()` **825**

C.2.5 Debugging with Spyder Example 2 826
C.2.6 More Debugging Tips 834
C.3 More about Testing 835
C.3.1 Testing Is Hard! 836
C.3.2 Importance of Testing 837
C.3.3 Other Kinds of Testing 837
C.4 What's Wrong with My Code? 837
C.4.1 Chapter 1: Beginnings 837
C.4.2 Chapter 2: Control 839
C.4.3 Chapter 4: Strings 840
C.4.4 Chapter 5: Functions 841
C.4.5 Chapter 6: Files and Exceptions 842
C.4.6 Chapter 7: Lists and Tuples 843
C.4.7 Chapter 8: More Functions 844
C.4.8 Chapter 9: Dictionaries 845
C.4.9 Chapter 11: Classes I 846
C.4.10 Chapter 12: Classes II 847
Appendix D Pylab: A Plotting and Numeric Tool 849
D.1 Plotting 849
D.2 Working with pylab 850
D.2.1 Plot Command 850
D.2.2 Colors, Marks, and Lines 851
D.2.3 Generating X-Values 851
D.2.4 Plot Properties 852
D.2.5 Tick Labels 853
D.2.6 Legend 854
D.2.7 Bar Graphs 856
D.2.8 Histograms 856
D.2.9 Pie Charts 857
D.2.10 How Powerful Is pylab? 858
Appendix E Quick Introduction to Web-based User Interfaces 861
E.0.1 MVC Architecture 862
E.1 Flask 862
E.2 QuickStart Flask, Hello World 863
E.2.1 What Just Happened? 864
E.2.2 Multiple Routes 865
E.2.3 Stacked Routes, Passing Address Arguments 867
E.3 Serving Up Real HTML Pages 868
E.3.1 A Little Bit of HTML 868
E.3.2 HTML Tags 868

E.3.3 Flask Returning Web Pages 870
E.3.4 Getting Arguments into Our Web Pages 871
E.4 Active Web Pages 873
E.4.1 Forms in `wtforms` 873
E.4.2 A Good Example Goes a Long Way 874
E.4.3 Many Fields Example 879
E.5 Displaying and Updating Images 884
E.6 Odds and Ends 889
Appendix F Table of UTF-8 One Byte Encodings 891
Appendix G Precedence 893
Appendix H Naming Conventions 895
H.1 Python Style Elements 896
H.2 Naming Conventions 896
H.2.1 Our Added Naming Conventions 896
H.3 Other Python Conventions 897
Appendix I Check Yourself Solutions 899
I.1 Chapter 1 899
Variables and Assignment 899
Types and Operators 899
I.2 Chapter 2 900
Basic Control Check 900
Loop Control Check 900
More Control Check 900
*`for`* and *`range`* Check 900
I.3 Chapter 4 901
Slicing Check 901
String Comparison Check 901
I.4 Chapter 5 901
Simple Functions Check 901
I.5 Chapter 6 901
Exception Check 901
Function Practice with Strings 902
I.6 Chapter 7 902
Basic Lists Check 902
Lists and Strings Check 902
Mutable List Check 902

I.7 Chapter 8 **902**
Passing Mutables Check **902**
More on Functions Check **903**
I.8 Chapter 9 **903**
Dictionary Check **903**
Set Check **903**
I.9 Chapter 11 **903**
Basic Classes Check **903**
Defining Special Methods **903**
I.10 Chapter 12 **904**
Check Defining Your Own Operators **904**
I.11 Chapter 14 **904**
Basic File Operations **904**
Basic Exception Control **904**

INDEX 905

# VIDEONOTES


VideoNote 0.1 Getting Python **45**
VideoNote 1.1 Simple Arithmetic **96**
VideoNote 1.2 Solving Your First Problem **105**
VideoNote 2.1 Simple Control **128**
VideoNote 2.2 Nested Control **172**
VideoNote 3.1 Algorithm Decomposition **209**
VideoNote 3.2 Algorithm Development **217**
VideoNote 4.1 Playing with Strings **242**
VideoNote 4.2 String Formatting **246**
VideoNote 5.1 Simple Functions **283**
VideoNote 5.2 Problem Design Using Functions **293**
VideoNote 6.1 Reading Files **304**
VideoNote 6.2 Simple Exception Handling **317**
VideoNote 7.1 List Operations **359**
VideoNote 7.2 List Application **381**
VideoNote 8.1 More on Parameters **437**
VideoNote 9.1 Using a Dictionary **469**
VideoNote 9.2 More Dictionaries **497**
VideoNote 10.1 Program Development: Tag Cloud **544**
VideoNote 11.1 Designing a Class **577**
VideoNote 11.2 Improving a Class **586**
VideoNote 12.1 Augmenting a Class **625**
VideoNote 12.2 Create a Class **628**
VideoNote 13.1 Improve Simulation **655**
VideoNote 14.1 Dictionary Exceptions **701**
VideoNote 15.1 Recursion **724**
VideoNote 16.1 Properties **774**

# PREFACE

A FIRST COURSE IN COMPUTER SCIENCE IS ABOUT A NEW WAY OF SOLVING PROBLEMS computationally. Our goal is that after the course, students when presented with a problem will think, "Hey, I can write a program to do that!"

The teaching of problem solving is inexorably intertwined with the computer language used. Thus, the choice of language for this first course is very important. We have chosen Python as the introductory language for beginning programming students—majors and non-majors alike—based on our combined 55 years of experience teaching undergraduate introductory computer science at Michigan State University. Having taught the course in Pascal, C/C++, and now Python, we know that an introductory programming language should have two characteristics. First, it should be relatively simple to learn. Python's simplicity, powerful built-in data structures, and advanced control constructs allow students to focus more on problem solving and less on language issues. Second, it should be practical. Python supports learning not only fundamental programming issues such as typical programming constructs, a fundamental object-oriented approach, common data structures, and so on, but also more complex computing issues such as threads and regular expressions. Finally, Python is "industrial strength" forming the backbone of companies such as YouTube, DropBox, Industrial Light and Magic, and many others.

We emphasize both the fundamental issues of programming and practicality by focusing on data manipulation and analysis as a theme—allowing students to work on real problems using either publicly available data sets from various Internet sources or self-generated data sets from their own work and interests. We also emphasize the development of programs, providing multiple, worked out, examples, and three entire chapters for detailed design and implementation of programs. As part of this one-semester course, our students have analyzed breast cancer data, catalogued movie actor relationships, predicted disruptions of satellites from solar storms, and completed many other data analysis problems. We have also found that concepts learned in a Python CS1 course transitioned to a CS2 C++ course with little or no impact on either the class material or the students.

Our goals for the book are as follows:

- Teach problem solving within the context of CS1 to both majors and nonmajors using Python as a vehicle.
- Provide examples of *developing* programs focusing on the kinds of data analysis problems students might ultimately face.
- Give students who take no programming course other than this CS1 course a practical foundation in programming, enabling them to produce useful, meaningful results in their respective fields of study.

## WHAT'S NEW, THIRD EDITION

We have taught with this material for over eight years and continue to make improvements to the book as well as adapting to the ever changing Python landscape, keeping up to date with improvements. We list the major changes we have made below.

**Anaconda**: One of the issues our students ran into was the complexity associated with getting Python packages, along with the necessary pre-requisites. Though tools like `pip` address this problem to some extent, the process was still a bit overwhelming for introductory students.

Thus we switched to the **Anaconda** distribution made freely available from Continuum Analytics. They make available a full distribution with more than 100 modules pre-installed, removing the need for package installation.

Appendix A, newly written for Anaconda, covers the installation process.

**SPYDER:** Another benefit of the Anaconda distribution is the SPYDER Integrated Development Environment. We have fully adopted SPYDER as our default method for editing and debugging code, changing from the default IDLE editor. SPYDER provides a full development environment and thus has a number of advantages (as listed at the Spyder git page, `https://github.com/spyder-ide/spyder/blob/master/README.md`).

- Integrated editor
- Associated interactive console
- Integrated debugging
- Integrated variable explorer
- Integrated documentation viewer

SPYDER is a truly modern Python IDE and the correct way for students to learn Python programming.

Chapter 1 has been rewritten, incorporating the SPYDER IDE and using SPYDER is sprinkled throughout the book. Appendix A provides a tutorial on SPYDER.

**IPython**: Anaconda also provides the iPython console as its interactive console. IPython is a much more capable console than the default Python console, providing many features including:

- An interactive history list, where each history line can be edited and re-invoked.
- Help on variables and functions using the "?" syntax.
- Command line completion

**Every session of the book** was redone for the iPython console and iPython's features are sprinkled throughout the book. Further, a tutorial on the use of iPython is provided in Appendix A.

**Debugging help**: Debugging is another topic that is often a challenge for introductory students. To address this need, we have introduced a "What's Wrong with My Code" element to the end of chapters 1, 2, 4, 5, 6, 7, 8, 9, and 11. These provide increasingly detailed tips to deal with new Python features as they are introduced. An overall tutorial is also provided in the **new Appendix C**.

As SPYDER provides a debugger, use of that debugger is used for all examples.

**Pylab updated**: We have incorporated graphing through `matplotlib/pylab` into the book since the first edition, but this module has changed somewhat over the years so the "Visual Vignettes" at the end of chapters 1, 2, 5, 7, 9, and 13 have been updated and somewhat simplified. In particular the discussions of NumPy have been removed except for where they are useful for graphing. Appendix D has also been updated with these changes.

**Web-based GUIs**: Building Graphic User Interfaces (GUIs) is a topic many students are interested in. However, in earlier editions we hesitated to focus on GUI development as part of an introductory text for a number of reasons:

- The extant `tkinter` is cross platform, but old and more complex to work with than we would like for an introductory class.
- Getting a modern GUI toolset can be daunting, especially cross platform.
- Just **which** GUI toolset should we be working with?

A discussion with Greg Wilson (thanks Greg!) was helpful in resolving this problem. He suggested doing Web-based GUIs as a modern GUI approach that is cross-platform, relatively stable and provided in modern distributions like Anaconda.

What that left was the "complexity" issue. We choose to use the package `flask` because it was relatively less complex, but more importantly was easily modularized and could be used in a template fashion to design a GUI.

Thus, we wrote some simple GUI development in `flask` at the end of chapters 4, 6, 7, 9, 10, and 15. We also wrote a **new Appendix E** as a tutorial for development of web-based GUIs.

**Functions Earlier**: One of the common feedback points we received was a request to introduce functions earlier. Though we had purposefully done strings first, as a way to

start working with data, we had sympathy for those instructors who wanted to change the order and introduce functions before strings. Rather than pick a right way to do this, we **rewrote Chapter 5** so that it had no dependencies on Chapter 4, the string chapter. Instructors are now free to introduce functions anytime after Chapter 2 on control. Likewise Chapter 4 on strings has no dependencies on Chapter 5, the functions chapters. Thus the instructor can choose the order they prefer.

**Online Project Archive**: We have established an online archive for Python CS1 projects, `http://www.cse.msu.edu/~cse231/PracticeOfComputingUsingPython/`. These describe various projects we have used over the years in our classes. This site and its contents has been recognized by the National Center for Women & Information Technology (NCWIT) and was awarded the 2015 NCWIT EngageCSEdu Engagement Excellence Award.

**New Exercises**: We added 80 new end-of-chapter exercises.

**Other changes**: We have made a number of other changes as well:

- We updated Chapter 16 with a discussion about Python Numbers and the various representations that are available
- We moved the content of some of the chapters in the "Getting Started" part to the "Data Structures and Functions" part. This is really just a minor change to the table of contents.
- We fixed various typos and errors that were either pointed out to us or we found ourselves as we re-read the book.

## ACKNOWLEDGMENTS FOR THE GLOBAL EDITION

Pearson would like to thank and acknowledge the following people for their contributions to the Global Editions.

**Contributor**
Abhinava Vatsa

**Reviewers**
Debajyoti Bera, Indraprastha Institute of Information Technology, Delhi
Shivani Pandit
Somitra Sanadhya, Indraprastha Institute of Information Technology, Delhi

# PREFACE TO THE SECOND EDITION

The main driver for a second edition came from requests for Python 3. We began our course with Python 2 because Python 3 hadn't been released in 2007 (it was first released in December 2008), and because we worried that it would take some time for important open-source packages such as NumPy and MatPlotLib to transition to Python 3. When NumPy and MatPlotLib converted to Python 3 in 2011 we felt comfortable making the transition. Of course, many other useful modules have also been converted to Python 3—the default installation now includes thousands of modules. With momentum building behind Python 3 it was time for us to rewrite our course and this text.

Why Python 3? The Python community decided to break backward compatibility with Python 3 to fix nagging inconsistencies in the language. One important change was moving the default character encoding to Unicode which recognizes the world-wide adoption of the language. In many ways beyond the introductory level, Python 3 is a better language and the community is making the transition to Python 3.

At the introductory level the transition to Python 3 appears to be relatively small, but the change resulted in touching nearly every page of the book.

One notable addition was:

- We added a set of nine **RULES** to guide novice programmers.

We reworked every section of this text—some more than others. We hope that you will enjoy the changes. Thanks.

## BOOK ORGANIZATION

At the highest level our text follows a fairly traditional CS1 order, though there are some differences. For example, we cover strings rather early (before functions) so that we can do more data manipulation early on. We also include elementary file I/O early for the same reason, leaving detailed coverage for a later chapter. Given our theme of data manipulation,

we feel this is appropriate. We also "sprinkle" topics like plotting and drawing throughout the text in service of the data manipulation theme.

We use an "object-use-first" approach where we use built-in Python objects and their methods early in the book, leaving the design and implementation of user-designed objects for later. We have found that students are more receptive to building their own classes once they have experienced the usefulness of Python's existing objects. In other words, we motivate the need for writing classes. Functions are split into two parts because of how Python handles mutable objects such as lists as parameters; discussion of those issues can only come after there is an understanding of lists as mutable objects.

Three of the chapters (3, 10, and 13) are primarily program design chapters, providing an opportunity to "tie things together," as well as showing how to design a solution. A few chapters are intended as supplemental reading material for the students, though lecturers may choose to cover these topics as well. For background, we provide a Chapter 0 that introduces some general concepts of a computer as a device and some computer terminology. We feel such an introduction is important—everyone should understand a little about a computer, but this material can be left for reading. The last chapters in the text may not be reached in some courses.

# BOOK FEATURES

## 1.0.1 Data Manipulation

Data manipulation is a theme. The examples range from text analysis to breast cancer classification. The data.gov site is a wonderful source of interesting and relevant data. Along the way, we provide some analysis examples using simple graphing. To incorporate drawing and graphing, we use established packages instead of developing our own: one is built-in (Turtle graphics); the other is widely used (MatPlotLib with NumPy).

We have tried to focus on non-numeric examples in the book, but some numeric examples are classics for a good reason. For example, we use a rational numbers example for creating classes that overload operators. Our goal is always to use the best examples.

## 1.0.2 Problem Solving and Case Studies

Throughout the text, we emphasize problem solving, especially a divide-and-conquer approach to developing a solution. Three chapters (3, 10, and 13) are devoted almost exclusively to program development. Here we walk students through the solution of larger examples. In addition to design, we show mistakes and how to recover from them. That is, we don't simply show a solution, but show a *process of developing* a solution.

## 1.0.3 Code Examples

There are over 180 code examples in the text—many are brief, but others illustrate piecemeal development of larger problems.

### 1.0.4 Interactive Sessions

The Python interpreter provides a wonderful mechanism for briefly illustrating programming and problem-solving concepts. We provide almost 250 interactive sessions for illustration.

### 1.0.5 Exercises and Programming Projects

Practice, practice, and more practice. We provide over 275 short exercises for students and nearly 30 longer programming projects (many with multiple parts).

### 1.0.6 Self-Test Exercises

Embedded within the chapters are 24 self-check exercises, each with five or more associated questions.

### 1.0.7 Programming Tips

We provide over 40 special notes to students on useful tips and things to watch out for. These tips are boxed for emphasis.

## SUPPLEMENTARY MATERIAL ONLINE

- For students
  - All example source code
  - Data sets used in examples

The above material is freely available at www.pearsonglobaleditions.com/punch

- For instructors
  - PowerPoint slides
  - Laboratory exercises
  - Figures (PDF) for use in your own slides
  - Exercise solutions

Qualified instructors may obtain supplementary material by visiting www.pearsonglobaleditions.com/punch. Register at the site for access. You may also contact your local Pearson Education sales representative

W. F. PUNCH
R. J. ENBODY

BREAKTHROUGH
To improving results

# MyProgrammingLab™

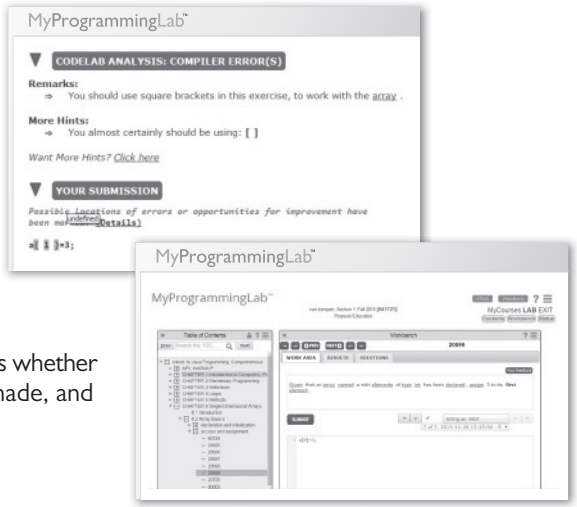
Through the power of practice and immediate personalized feedback, MyProgrammingLab helps improve your students' performance.

## PROGRAMMING PRACTICE

With MyProgrammingLab, your students will gain first-hand programming experience in an interactive online environment.

## IMMEDIATE, PERSONALIZED FEEDBACK

MyProgrammingLab automatically detects errors in the logic and syntax of their code submission and offers targeted hints that enables students to figure out what went wrong and why.




## GRADUATED COMPLEXITY

MyProgrammingLab breaks down programming concepts into short, understandable sequences of exercises. Within each sequence the level and sophistication of the exercises increase gradually but steadily.

## DYNAMIC ROSTER

Students' submissions are stored in a roster that indicates whether the submission is correct, how many attempts were made, and the actual code submissions from each attempt.

## PEARSON eTEXT

The Pearson eText gives students access to their textbook anytime, anywhere.

## STEP-BY-STEP VIDEONOTE TUTORIALS

These step-by-step video tutorials enhance the programming concepts presented in select Pearson textbooks.

For more information and titles available with **MyProgrammingLab**, please visit **www.myprogramminglab.com**.



ALWAYS LEARNING

PEARSON

PART 1

# Thinking About Computing

**Chapter 0** The Study of Computer Science

This page intentionally left blank

CHAPTER 0

# The Study of Computer Science

> Composing computer programs to solve scientific problems is like writing poetry. You must choose every word with care and link it with the other words in perfect syntax.
>
> James Lovelock

## 0.1 WHY COMPUTER SCIENCE?

It is a fair question to ask. Why should anyone bother to study computer science? Furthermore, what is "computer science"? Isn't this all just about programming? All good questions. We think it is worth discussing them before you forge ahead with the rest of the book.

### 0.1.1 Importance of Computer Science

Let's be honest. We wouldn't be writing the book and asking you to spend your valuable time if we didn't think that studying computer science is important. There are a couple of ways to look at why this is true.

First, we all know that computers are everywhere, millions upon millions of them. What were once rare, expensive items are as common place as, well, any commodity you can imagine (we were going to say the proverbial toaster, but there are many times more computers than toasters. In fact, there is likely a small computer *in* your toaster!). However, that isn't enough of a reason. There are millions and millions of cars, and universities don't require auto mechanics as an area of study.

Second, Computers are not only common, but they are also more universally applicable than any other commodity in history. A car is good for transportation, but a computer can

be used in so many situations. In fact, there is almost no area one can imagine where a computer would *not* be useful. That is a key attribute. No matter what your area of interest, a computer could be useful there as a *tool*. The computer's universal utility is unique, and learning how to use such a tool is important.

## 0.1.2 Computer Science Around You

Computing surrounds you, and it is computer science that put it there. There are a multitude of examples, but here are a few worth noting.

**Social Networking** The tools that facilitate social networking such as Facebook or Twitter are, of course, computer programs. However, the tools that help study the interactions within social networks involve important computer science fields such as *graph theory*. For example, the Iraqi dictator Saddam Hussein was located using the graph theoretic analysis of his social network.

**Smartphones** Smartphones are small, very portable computers. Apps for smartphones are simple computer programs written specifically for smartphones.

**Your Car** Your car probably hosts dozens of computers. They control the engine, the brakes, the audio system, the navigation, and the climate control system. They determine if a crash is occurring and trigger the air bags. Some cars park automatically or apply the brakes if a crash is imminent. Fully autonomous cars are being tested, as are cars that talk to each other.

**The Internet** The backbone of the Internet is a collection of connected computers called *routers* that decide the best way to send information to its destination.

## 0.1.3 Computer "Science"

> Any field that has the word science in its name is guaranteed thereby not to be a science.
>
> Frank Harary

A popular view of the term "computer science" is that it is a glorified way to say "computer programming." It is true that computer programming is often the way that people are introduced to computing in general, and that computer programming is the primary reason many take computing courses. However, there is indeed more to computing than programming, hence the term "computer science." Here are a few examples.

### Theory of Computation

Before there were the vast numbers of computers that are available today, scientists were thinking about what it means to do computing and what the limits might be. They would ask questions, such as whether there exist problems that we can conceive of but cannot

compute. It turns out there are. One of these problems, called the "Halting Problem,"[1] cannot be solved by a program running on any computer. Knowing what you can and cannot solve on a computer is an important issue and a subject of study among computer scientists that focus on the theory of computation.

## Computational Efficiency

The fact that a problem is computable does not mean it is easily computed. Knowing roughly how difficult a problem is to solve is also very important. Determining a meaningful measure of difficulty is, in itself, an interesting issue, but imagine we are concerned only with time. Consider designing a solution to a problem that, as part of the solution, required you to sort 100,000 items (say cancer patient records, or asteroid names, or movie episodes, etc.). A slow algorithm, such as the sorting algorithm called the Bubble Sort, might take approximately 800 seconds (about 13 minutes); another sorting algorithm called Quick Sort might take approximately 0.3 seconds. That is a difference of around 2400 times! That large a difference might determine whether it is worth doing. If you are creating a solution, it would be good to know what makes your solution slow or what makes it fast.

## Algorithms and Data Structures

Algorithms and data structures are the currency of the computer scientist. Discussed more in Chapter 3, algorithms are the methods used to solve problems, whereas data structures are the organizations of data that the algorithms use. These two concepts are distinct: a general approach to solving a problem (such as searching for a particular value, sorting a list of objects and encrypting a message) differs from the organization of the data that is being processed (as a list of objects, as a dictionary of key-value pairs, as a "tree" of records). However, they are also tightly coupled. Furthermore, both algorithms and data structures can be examined independently of how they might be programmed. That is, one designs algorithms and data structures and then actually implements them in a particular computer program. Understanding abstractly how to design both algorithms and data structures independent of the programming language is critical for writing correct and efficient code.

## Parallel Processing

It may seem odd to include what many consider an advanced topic, but parallel processing, using multiple computers to solve a problem, is an issue for everyone these days. Why? As it turns out, most computers come with at least two processors or CPUs (see Section 0.6), and many come with four or more. The Playstation4(™) game console uses a special AMD chip with a total of eight processors, and Intel has released its new Phi card with more than 60 processors! What does this mean to us, as both consumers and new computer scientists?

[1] http://www.wired.com/2014/02/halting-problem/

The answer is that new algorithms, data structures, and programming paradigms will be needed to take advantage of this new processing environment. Orchestrating many processors to solve a problem is an exciting and challenging task.

### Software Engineering

Even the process of writing programs itself has developed its own subdiscipline within computer science. Dubbed "software engineering," it concerns the process of creating programs: from designing the algorithms they use, to supporting testing and maintenance of the program once created. There is even a discipline interested in representing a developed program as a mathematical entity so that one can *prove* what a program will do once written.

### Many Others

We have provided but a taste of the many fields that make computer science such a wonderfully rich area to explore. Every area that uses computation brings its own problems to be explored.

## 0.1.4 Computer Science Through Computer Programming

We have tried to make the point that computer science is not just programming. However, it is also true that for much of the book we will focus on just that aspect of computer science: programming. Beginning with "problem solving through programming" allows one to explore pieces of the computer science landscape as they naturally arise.

# 0.2 THE DIFFICULTY AND PROMISE OF PROGRAMMING

If computer science, particularly computer programming, is so interesting, why doesn't everybody do it? The truth is that it can be hard. We are often asked by beginning students, "Why is programming so hard?" Even grizzled programming veterans, when honestly looking back at their first experience, remember how difficult that first programming course was. Why? Understanding why it might be hard gives you an edge on what you can do to control the difficulty.

## 0.2.1 Difficulty 1: Two Things at Once

Let's consider an example. Let us say that, when you walk into that first day of Programming 101, you discover the course is not about programming but French poetry. French poetry?

Yes, French poetry. Imagine that you come in and the professor posts the following excerpt from a poem on the board.

**A une Damoyselle malade**

Ma mignonne,
Je vous donne
Le bon jour;
Le séjour
C'est prison.

Clément Marot

Your assigned task is to translate this poetry into English (or German, or Russian, whatever language is your native tongue). Let us also assume, for the moment, that:

(a) You do not know French.
(b) You have never studied poetry.

You have two problems on your hands. First, you have to gain a better understanding of the syntax and semantics (the form and substance) of the French language. Second, you need to learn more about the "rules" of poetry and what constitutes a good poem.

Lest you think that this is a trivial matter, an entire book has been written by Douglas Hofstadter on the very subject of the difficulty of translating this one poem ("Le Ton beau de Marot").

So what's your first move? Most people would break out a dictionary and, line by line, try to translate the poem. Hofstadter, in his book, does exactly that, producing the crude translation in Figure 0.1.

| **My Sweet/Cute [One] (Feminine)** | |
|---|---|
| My sweet/cute [one] (feminine) | |
| I [to] you (respectful) give/bid/convey | **A une Damoyselle malade** |
| The good day (i.e., a hello, i.e., greetings). | Ma mignonne, |
| The stay/sojourn/ visit (i.e., quarantine) | Je vous donne |
| {It} is prison. | Le bon jour; |
| | Le séjour |
| | C'est prison. |

**FIGURE 0.1** Crude translation of excerpt.

The result is hardly a testament to beautiful poetry. This translation does capture the syntax and semantics, but not the poetry, of the original. If we take a closer look at the poem, we can discern some features that a good translation should incorporate. For example:

- Each line consists of three syllables.
- Each line's main stress falls on its final syllable.
- The poem is a string of rhyming couplets: *AA, BB, CC,* …
- The semantic couplets are out of phase with the rhyming couplets: *A, AB, BC,* …

Taking some of these ideas (and many more) into account, Hofstadter comes up with the translation in Figure 0.2.

| **My Sweet Dear** | **A une Damoyselle malade** |
|---|---|
| My sweet dear, | Ma mignonne, |
| I send cheer – | Je vous donne |
| All the best! | Le bon jour; |
| Your forced rest | Le séjour |
| Is like jail. | C'est prison. |

**FIGURE 0.2** Improved translation of excerpt.

Not only does this version sound far more like poetry, but it also matches the original poem, following the rules and conveying the intent. It is a pretty good translation!

## Poetry to Programming?

How does this poetry example help? Actually, the analogy is pretty strong. In coming to programming for the first time, you face exactly the same issues:

- You are not yet familiar with the syntax and semantics of the language you are working with—in this case, of the programming language Python and perhaps not of *any* programming language.
- You do not know how to solve problems using a computer—similar to not knowing how to write poetry.

Just like the French poetry neophyte, you are trying to solve two problems simultaneously. On one level, you are just trying to get familiar with the syntax and semantics of the language. At the same time, you are tackling a second, very difficult task: creating poetry in the previous example and solving problems using a computer in this course.

Working at two levels, the meaning of the programming words and then the intent of the program (what the program is trying to solve) are the two problems the beginning programmer has to face. Just like the French poetry neophyte, your first programs will be a bit clumsy as you learn both the programming language and how to use that language to solve problems. For example, to a practiced eye, many first programs look similar in nature

to the literal translation of Hofstadter's in Figure 0.1. Trying to do two things simultaneously is difficult for anyone, so be gentle on yourself as you go forward with the process.

You might ask whether there is a better way. Perhaps, but we have not found it yet. The way to learn programming is to program, just like swinging a baseball bat, playing the piano, and winning at bridge; you can hear the rules and talk about the strategies, but learning is best done by doing.

## 0.2.2 Difficulty 2: What Is a Good Program?

Having mastered some of the syntax and semantics of a programming language, how do we write a good program? That is, how do we create a program that is more like poetry than like the mess arrived at through literal translation?

It is difficult to discuss a good program when, at this point, you know so little, but there are a couple of points that are worth noting even before we get started.

### It's All About Problem Solving

If the rules of poetry are what guides writing good poetry, what are the guidelines for writing good programs? That is, what is it we have to learn in order to transition from a literal translation to a good poem?

For programming, it is *problem solving*. When you write a program, you are creating, in some detail, how it is that *you* think a particular problem or some class of problems, should be solved. Thus, the program represents, in a very accessible way, your thoughts on problem solving. Your thoughts! That means, before you write the program you must *have* some thoughts.

It is a common practice, even among veteran programmers, to get a problem and immediately sit down and start writing a program. Typically that approach results in a mess, and, for the beginning programmer, it results in an unsolved problem. Figuring out how to solve a problem requires some initial thought. If you think before you program, you better understand what the problem requires as well as the best strategies you might use to solve that problem.

Remember the two-level problem? Writing a program as you figure out how to solve a problem means that you are working at two levels at once: the problem-solving level and the programming level. That is more difficult than doing things sequentially. You should sit down and think about the problem and how you want to solve it *before* you start writing the program. We will talk more about this later, but rule 1 is:

| **Rule 1:** Think before you program!

### A Program as an Essay

When students are asked "What is the most important feature a program should have?" many answer, "It should run." By "run," they mean that the program executes and actually does something.

Wrong. As with any new endeavor, it is important to get the fundamentals correct right at the beginning. So **RULE 2** is

> **Rule 2:** A program is a human-readable essay on problem solving that also happens to execute on a computer.

A program is an object to be read by another person, just as any other essay. Although it is true that a program is written in such a way that a computer can execute it, it is still a human-readable essay. If your program is written so that it runs, and even runs correctly (notice we have not discussed "correctly" yet!), but is unreadable, then it is really fairly worthless.

The question is why? Why should it be that people must read it? Why isn't running good enough? Who's going to read it anyway? Let's answer the last question first. The person who is going to read it the most is you! That's correct, *you* have to read the programs you are writing all the time. Every time you put your program away for some period of time and come back to it, you have to re-read what you wrote and understand what you were thinking. Your program is a record of your thoughts on solving the problem and you have to be able to read your program in order to work with it, update it, add to it, and so on.

Furthermore, once you get out of the academic environment where you write programs solely for yourself, you will be writing programs with other people as a group. Your mates have to be able to read what you wrote! Think of the process as developing a newspaper edition. Everyone has to be able to read each others' content so that the edition, as a whole, makes sense. Just writing words on paper isn't enough—they have to fit together.

Our goal is to write programs that other people can read, as well as be run.

## 0.2.3 The Promise of a Computer Program

A program is an essay on problem solving and that will be our major focus. However, it is still interesting that programs do indeed run on a computer. That is, in fact, one of the unique, and most impressive parts about a program. Consider that idea for a moment. You can think of a way to solve a problem, write that thought down in detail as a program, and (assuming you did it correctly) that problem gets solved. More importantly, the problem can be solved again and again because the program can be used *independent of you*. That is, your thoughts not only live on as words (because of the essay-like nature of the program), but also as an entity that actually implements those thoughts. How amazing is that! Do you have some thoughts about how to make a robot dance? Write the program, and the robot dances. Do you have some ideas on how to create music? Write the program, and music is written automatically. Do you have an idea of how to solve a Sudoku puzzle? Write the program, and every puzzle is solved.

So a computer program (and the computer it runs on) offers a huge leap forward, perhaps the biggest since Gutenberg in the mid-1400s. Gutenberg invented moveable type, so that the written word of an individual—that is, their thoughts—could be reproduced

independently of the writer. Now, not only can those thoughts be copied, but also implemented to be *used* over and over again.

Programming is as open, as universally applicable, as the thoughts of the people who do the programming, because a program is the manifest thought of the programmer.

## 0.3 CHOOSING A COMPUTER LANGUAGE

We have selected a particular programming language, the language called Python, for this introductory text. You should know that there are lots of programming languages out there. Wikipedia lists more than 500 programming languages (here is a list of the top 100[2]). Why so many languages, and why Python for this text?

### 0.3.1 Different Computer Languages

If a program is a concrete, runnable realization of a person's thoughts, then it makes sense that different people would create languages that allow them to better reflect those thoughts. In fact, computer scientists are crazy about languages, which is why there are so many. Whatever the reason that some language was created—and there can be many reasons—they all reflect some part of the creator's view of how to best express solving problems on a computer. In fact, computer scientists are specifically trained to write their own language to suit their needs—a talent few other disciplines support so fully!

So given all these languages, why did we pick Python?

### 0.3.2 Why Python?

> I set out to come up with a language that made programmers more productive.
>
> Guido van Rossum, author of Python

There are three features that we think are important for an introductory programming language:

- The language should provide a low "cognitive load" on the student. That is, it should be as easy as possible to express your problem-solving thoughts in the mechanisms provided by the programming language.
- Having spent all your time learning this language, it should be easy to apply it to problems you will encounter. In other words, having learned a language, you should be able to write short programs to solve problems that pop up in your life (sort your music, find a file on your computer, find the average temperature for the month, etc.).

[2] http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

- The programming language you use should have broad support across many disciplines. That is, the language should be embraced by practitioners from many fields (arts to sciences, as they say) and useful packages, collections of support programs, should be available to many different types of users.

So how does Python match up against these criteria?

## Python Philosophy

Python offers a philosophy: *There should be one—and preferably only one—obvious way to do it.* The language should offer, as much as possible, a one-to-one mapping between the problem-solving need and the language support for that need. This is not necessarily the case with all programming languages. Given the two-level problem the introductory programmer already faces, reducing the programming language load as much as possible is important. Though Python does have its short cuts, they are far fewer than many other languages, and there is less "language" one has to remember to accomplish your task.

## A "Best Practices" Language

One of our favorite descriptions of Python is that it is a "best practices" language. This means that Python provides many of the best parts of other languages directly to the user. Important data structures are provided as part of the standard language; iteration (described later) is introduced early and is available on standard data structures; packages for files, file paths, the Web, and so on are part of the standard language. Python is often described as a "batteries included" language in which many commonly needed aspects are provided by default. This characteristic means that you can use Python to solve problems you will encounter.

## Python Is Open Source

One of Python's most powerful features is its support from the various communities and the breadth of that support. One reason is that Python is developed under the *Open Source* model. Open Source is both a way to think about software and a culture or viewpoint used by those who develop software. Open Source for software is a way to make software freely available and to guarantee that free availability to those who develop new software based on Open Source software. Linux, a type of operating system, is such a project, as are a number of other projects, such as Firefox (web browser), Thunderbird (mail client), Apache (web server), and, of course, Python. As a culture, Open Source adherents want to make software as available and useful as possible. They want to share the fruits of their labor and, as a group, move software forward, including the application areas where software is used. This perspective can be summarized as follows:

> A rising tide lifts all boats.

Each person is working explicitly (or implicitly) as part of a larger group towards a larger goal, making software available and useful. As a result of Python's Open Source

development, there are free, specialized packages for almost any area of endeavor including: music, games, genetics, physics, chemistry, natural language, geography, and others. If you know Python and can do rudimentary programming, there are packages available that will support almost any area you care to choose.

### 0.3.3 Is Python the Best Language?

VideoNote 0.1
Getting Python

The answer to that question is that there is no "best" language. All computer programming languages are a compromise to some degree. After all, wouldn't it be easiest to describe the program in your own words and just have it run? Unfortunately, that isn't possible. Our present natural language (English, German, Hindi, whatever) is too difficult to turn into the precise directions a computer needs to execute a program. Each programming language has its own strengths and weaknesses. New programmers, having mastered their first programming language, are better equipped to examine other languages and what they offer. For now, we think Python is a good compromise for the beginning programmer.

## 0.4 WHAT IS COMPUTATION?

It can be difficult to find a good definition for a broadly used word such as "computation." If you look, you will find definitions that include terms such as "information processing," "sequence of operations," "numbers," "symbols," and "mathematical methods." Computer scientists interested in the theory of computation formally define what a computation is, and what its limits are. It is a fascinating topic, but it is a bit beyond our scope.

We will use an English language definition that suits our needs. In this book we will define a computation as:

- *Computation* is the manipulation of data by either humans or machines.

The data that is manipulated may be numbers, characters, or other symbols.

## 0.5 WHAT IS A COMPUTER?

The definition of a computer then is

- A *computer* is something that does computation.

That definition is purposefully vague on *how* a computer accomplishes computation, only that it does so. This imprecision exists because what counts as a computer is surprisingly diverse. However, there are some features that almost any system that does computation should have.

- A computer should be able to *accept input*. What counts as input might vary among computers, but data must be able to enter the computer for processing.
- If a computer is defined by its ability to *do computation*, then any object that is a computer must be able to manipulate data.
- A computer must be able to *output data*.

Another characteristic of many computers is their ability to perform computation using an assembly of only simple parts. Such computers are composed of huge numbers of simple parts assembled in complex ways. As a result, the complexity of many computers comes from the organization of their parts, not the parts themselves.

## 0.5.1 Computation in Nature

There are a number of natural systems that perform computation. These are areas of current, intense investigation in computer science as researchers try to learn more from existing natural systems.

### The Human Brain

When electronic computers were first brought to the public's attention in the 1950s and even into the 1960s, they were often referred to as "electronic brains." The reason for this was obvious. The only computing object we had known up to that time was the human brain.

The human brain is in fact a powerful computing engine, though it is not often thought of in quite that way. It constantly takes in a torrent of input data—sensory data from the five senses—manipulates that data in a variety of ways, and provides output in the form of both physical action (both voluntary and involuntary) as well as mental action.

The amazing thing about the human brain is that the functional element of the brain is a very simple cell called the *neuron* (Figure 0.3).

Though a fully functional cell, a neuron also acts as a kind of small switch. When a sufficient signal reaches the *dendrites* of a neuron, the neuron "fires" and a signal is transmitted down the *axon* of the neuron to the *axon terminals*. Neurons are not directly connected to one another; rather, the dendrites of one neuron are located very close to the axon terminals of another neuron, separated by a space known as the *synapse*. When the signal reaches the axon terminal, chemicals called *neurotransmitters* are secreted across the synapse. It is the transmission of a signal, from neurotransmitters secreted by the axon terminals of one neuron to the dendrites of a connecting neuron that constitutes a basic computation in the brain.

Here are a few interesting facts about neuronal activity:

- The signal within a neuron is not transmitted by electricity (as in a wire), but by rapid chemical changes that propagate down the axon. The result is that the signal is very much slower than an electrical transmission down a wire—about a million times slower.

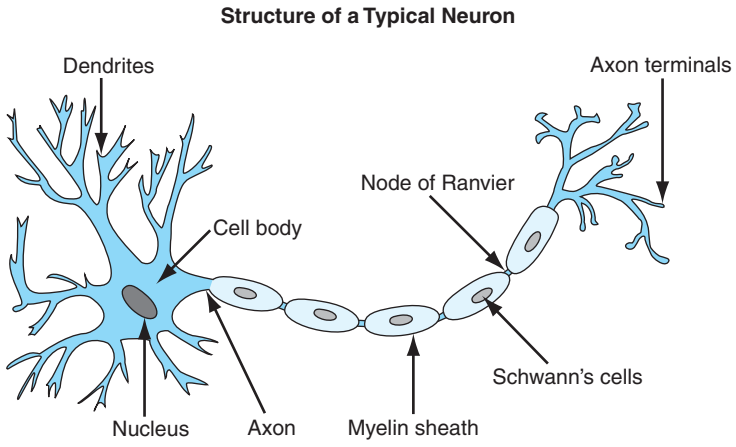



**FIGURE 0.3** An annotated neuron. [SEER Training Modules, U.S. National Institutes of Health, National Cancer Institute.]

- Because the signal is chemical, a neuron must typically recover for a millisecond (one-thousandth of a second) before it can fire again. Therefore, there is a built-in time delay to neuronal firing.
- A neuron is "fired" in response to some combination of the number of signals that are received at its dendrite (how many other neurons are firing in proximity) and the strength of the signal received (how much neurotransmitter is being dumped into the synapse).

One thing to note is how *slow* the neuron is as a switch. As you will see in Section 0.6.2, electronic switches can act many millions of times faster. However, what the brain lacks in terms of speedy switches, it makes up for in sheer size and complexity of organization. By some estimates, the human brain consists of of 100 billion ($10^{11}$) neurons and 100 trillion ($10^{14}$) synapses. Furthermore, the organization of the brain is incredibly complicated. Scientists have spent hundreds of years identifying specific areas of the brain that are responsible for various functions, and how those areas are interconnected. Yet for all this complexity, the main operational unit is a tiny, slow switch.

Scientists have been fascinated by the brain, so much so that there is a branch of computer science that works with simulations of *neural networks*, networks consisting of simple switches such as those found in the brain. Neural networks have been used to solve many difficult problems.

## Evolutionary Computation

The evolution of biological species can be viewed as a computation process. In this view, the inputs of the computational process are the environmental variables the biological entity is

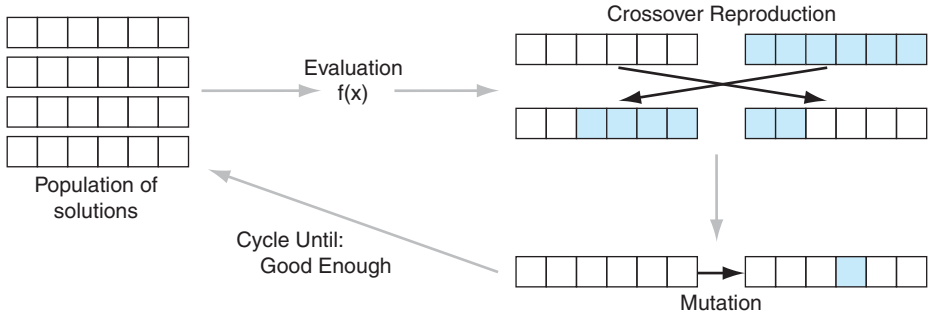



**FIGURE 0.4** A genetic algorithm.

subjected to; the computational process is the adaptation of the genetic code and the output is the adaptation that results from the genetic code modification.

This point of view has been incorporated into approaches known as *genetic algorithms*. These techniques take the concepts of simple genetics, as proposed by Gregor Mendel, and the processes of evolution as described by Charles Darwin, and use them to compute.

The basic parts of a genetic algorithm, shown in Figure 0.4, are:

- A way to *encode* a solution in a linear sequence, much like the sequence of information contained in a chromosome. This encoding depends on the problem, but it typically consists of parameters (struts for a bridge, components for a circuit, jobs for a schedule, etc.) that are required to constitute a solution.
- A method to *evaluate* the "goodness" of a particular solution, called the *evaluation function* and represented as f(x) in the diagram.
- A *population* of solutions often initially created by random selection of the solution components, from which new solutions can be created.
- Some genetic *modification* methods to create new solutions from old solutions. In particular, there is *mutation*, which is a modification of one aspect of an existing solution and *crossover*, which combines aspects of two parent solutions into a new solution.

The process proceeds as follows. Each solution in the population is evaluated to determine how fit it is. Based on the fitness of the existing solutions, new solutions are created using the genetic modification methods. Those solutions that are more fit are given more opportunity to create new solutions; less fit solutions are given less opportunity. This process incorporates a "survival of the fittest" notion. Over time, better and better solutions are evolved that solve the existing problem.

Genetic algorithms and other similar approaches have been used to solve many complex problems such as scheduling, circuit design, and others.

**FIGURE 0.5** NACA (National Advisory Committee for Aeronautics) High-Speed-Flight Station Computer Room. [NASA/Courtesy of nasaimages.org]

## 0.5.2 The Human Computer

The common use of the word "computer" from around the seventeenth century to about World War II referred to *people*. To compute difficult, laborious values for mathematical constants (such as $\pi$ or $e$), fission reactions (for the Manhattan Project), and gun trajectory tables, people were used (Figure 0.5). However, using people had problems.

A classic example of such a problem was created by William Shanks, an English amateur mathematician, who in 1873 published $\pi$ calculated to 707 decimal places. The calculation by hand took 28 years. Unfortunately, he made a calculation error at the 528th digit that made the last two years of calculation a waste of time. His error wasn't found until 70 years later using a mechanical calculator.

The U.S. Army's Ballistics Research Laboratory was responsible for the creation of gun trajectory tables. Each new artillery piece required a table for the gunner to use to calculate where the round would land. However, it took significant human effort to make these tables. Kay Mauchly Antonelli, a mathematician from the University of Pennsylvania and one

of the six original programmers of the ENIAC, the first general-purpose electronic digital computer, said, "To do just one trajectory, at one particular angle, usually took between 30 to 40 hours of calculation on this [mechanical] desk calculator." These tables exceeded 1800 entries and required up to four years to produce by hand.[3]

It was obvious that something had to be done. People were neither accurate enough nor fast enough, to do this kind of calculation. A more modern, faster, and accurate approach was needed.

## 0.6 THE MODERN, ELECTRONIC COMPUTER

Although there may be many notions of a computer, we all know what a modern computer is. We read e-mail on it, text message each other, listen to music, play videos, and play games on them. The design of these computers was first conceived around World War II to solve those tedious calculation problems that humans did so slowly, especially the Army's ballistics tables. How do electronic computers work? What makes them so special?

### 0.6.1 It's the Switch!

Modern digital computers use, as their base component, nothing more complicated than a simple switch. Very early computers used mechanical switches or relays; later versions used vacuum tubes, and finally, modern computers use transistors (see Figure 0.6).

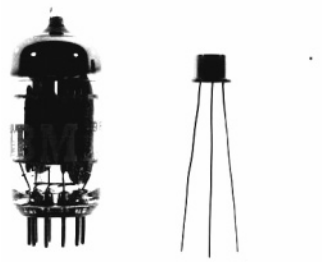


**FIGURE 0.6** Vacuum tube, single transistor, and chip transistor (the dot). [Reprint Courtesy of International Business Machines Corporation, copyright © International Business Machines Corporation].

A switch's function is pretty obvious. It is either on or off. When turned on, electricity flows through the switch and when turned off, no electrical flow occurs. Using a switch and its on/off property, you can construct simple logic circuits. In logic, we have only two states: True and False. In our logic circuit, we translate True and False to the physical process of a switch. The True condition is represented by a current flowing through the circuit and

[3] http://www.comphist.org/pdfs/Video-Giant%20Brains-MachineChangedWorld-Summary.pdf

False is represented by lack of current flow. Though we will cover Boolean logic in more detail in Chapter 2, there are two simple combinations of switches we can show: the ***and*** and ***or*** circuits.

For the Boolean ***and***, a True value results *only* if both of the two input values are True. All other combinations of input have a False output. In the ***and*** circuit, we connect two switches together in series as shown in Figure 0.7. Electricity can flow, that is, the circuit represents a True value, only if *both* switches are turned on. If either switch is turned off, no electricity flows and the circuit represents a False value.

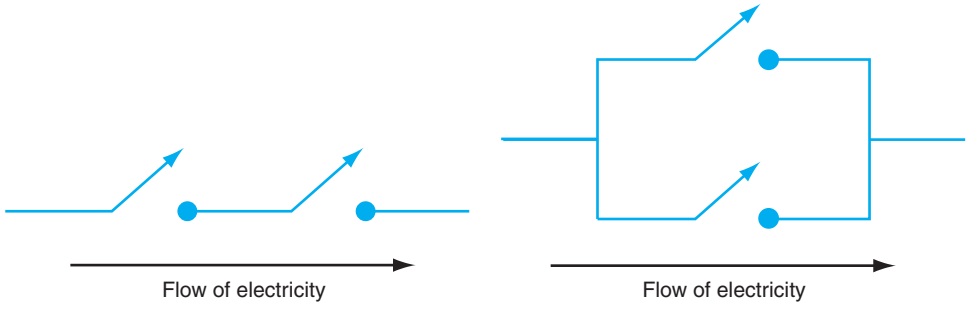



**FIGURE 0.7** Switches implementing an ***and*** gate (left) and an ***or*** gate (right).

We can do the same for the Boolean ***or***. An ***or*** is True if *either one or both* of its inputs are True; otherwise, it is False. In the ***or*** circuit of Figure 0.7, we connect two switches together in parallel. Electricity can flow if *either* switch is turned on, representing a True value. Only when both switches are turned off does the circuit represent a False value.

Similar Boolean logic elements, usually called logic *gates*, can be constructed from simple circuits. The amazing thing is that using only simple logic circuits, we can assemble an entire computer. For example, an ***and*** and ***or*** gate can be combined to make a simple circuit to add two values, called an *adder*. Once we can add, we can use the adder circuit to build a subtraction circuit. Further, once we have an adder circuit we can do multiplication (by repeated addition), and then division, and so on. Just providing some simple elements like logic gates allows us to build more complicated circuits until we have a complete computer.

## 0.6.2 The Transistor

Although any switch will do, the "switch" that made the electronic computer what it is today is called a *transistor*. The transistor is an electronic device invented in 1947 by William Shockley, John Bardeen, and Walter Brattain at Bell Labs (for which they eventually won the Nobel Prize in Physics in 1956). It utilized a new technology, a material called a semiconductor, that allowed transistors to supersede the use of other components such as vacuum tubes. Though a transistor has a number of uses, the one we care most about for computer use is as a switch. A transistor has three wires with the names Source, Sink, and Gate. Electricity flows from the Source to the Sink. If there is a signal, a voltage, or current, on the Gate,

then electricity flows—the switch is "on." If there is no signal on the Gate, no electricity flows—the switch is "off." See Figure 0.8.

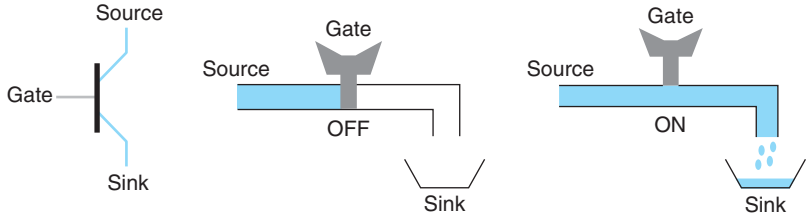



**FIGURE 0.8** A diagram of a transistor and its equivalent "faucet" view.

In the switch examples shown previously, we can use transistors as switches, for example, as in Figure 0.7. In that way, transistors are used to construct logic gates, then larger circuits, and so on, eventually constructing the higher-level components that become the parts of a modern electronic computer.

What makes the transistor so remarkable is how it has evolved in the 60 years since its first creation. It is this remarkable evolution that has made the modern computer what it is today.

## Smaller Size

The size of a transistor has changed dramatically since its inception. The first Shockley transistor was very large, on the order of inches (see Figure 0.9). By 1954, Texas Instruments was selling the first commercial transistor and had shrunk the transistor size to that of a postage stamp.

However, even the small size of individual transistor components was proving to be a limitation. More transistors were needed in a smaller area if better components were to be designed. The solution was the *integrated circuit*, invented by Jack Kilby of Texas Instruments

**FIGURE 0.9** The Shockley transistor—the first transistor. [Image courtesy of Science & Society Picture Library/Getty Images.]

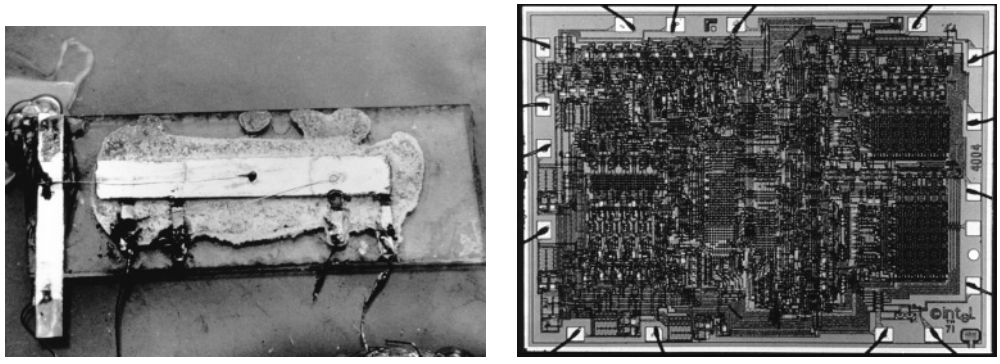


**FIGURE 0.10** Kilby's first integrated circuit (left) and the Intel 4004 microprocessor (right). [Image courtesy of Texas Instruments (left). Image courtesy of Intel Corporation (right).]

in 1958–1959 (for which he won the 2000 Nobel Prize in Physics); see Figure 0.10. The integrated circuit was a contiguous piece of semiconductor material upon which multiple transistors could be manufactured. The integrated circuit allowed many transistors to be embedded in a single piece of material, allowing a more complex functional circuit on a very small area. By 1971, Intel managed to manufacture the first complete computer processing unit (or CPU) on a single chip, a microprocessor named the Intel 4004 (see Figure 0.10). It was approximately the size of a human fingernail (3 × 4 mm) with 2300 transistors. By this time, the size of the transistor on this microprocessor had shrunk to 10 microns, the width of the a single fiber of cotton or silk.

The shrinkage of the transistor has continued. Today, the size of a transistor has reached amazingly small levels. Figure 0.11 shows an electron microscope picture of an IBM

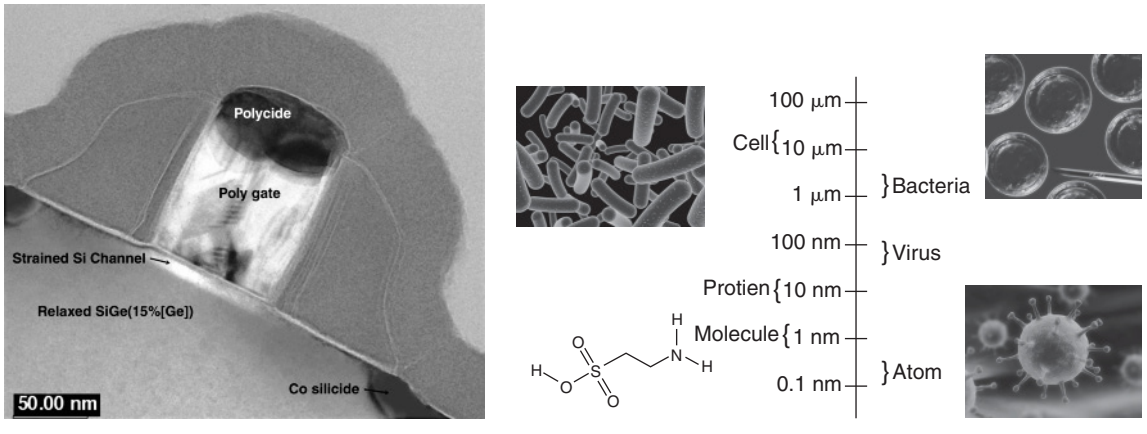



**FIGURE 0.11** A photomicrograph of a single 50 nm IBM transistor (left) and a scale small items (right). [Reprint Courtesy of International Business Machines Corporation, copyright © International Business Machines Corporation (left) Sebastian Kaulitzki/Shutterstock (right)].

| Year | Transistor Count | Model |
|---|---|---|
| 1971 | 2,300 | 4004 |
| 1978 | 29,000 | 8086 |
| 1982 | 134,000 | 80286 |
| 1986 | 275,000 | 80386 |
| 1989 | 1,200,000 | 80486 |
| 1993 | 3,100,000 | Pentium |
| 1999 | 9,500,000 | Pentium III |
| 2001 | 42,000,000 | Pentium 4 |
| 2007 | 582,000,000 | Core 2 Quad |
| 2011 | 2,600,000,000 | 10-core Westmere |
| 2014 | 3,000,000,000 | Apple A8X (iPad2) |
| 2015 | 5,560,000,000 | 18-core Xeon Haswell |

**TABLE 0.1** Transistor counts in Intel microprocessors, by year.

transistor gate that is 50 nanometers wide, $50 \times 10^{-9}$ meters, a thousand times smaller than transistors in the Intel 4004. That is more than 10 times smaller than a single wavelength of visible light. It is approximately the thickness of a cell membrane and only 10 times larger than a single turn of a DNA helix. Current transistors are nearly a third that size at 14 nm with smaller devices being tested in labs.

## Quantity and Function

As the size of a transistor shrank, the number of transistors that could be put on a single chip increased. This increase has been quite dramatic. In fact, there is a famous statement made by the founder of Intel, Gordon Moore, that predicts this amazing trend. In 1965, Moore predicted that the number of transistors that can be placed inexpensively on a single chip would double about every two years.[4] This trend has proven to be remarkably accurate and continues to hold to this day. Named in the popular press as "Moore's Law," its demise has been predicted for many years, yet it continues to hold true. A summary of this trend is shown in Table 0.1.

By increasing the number of transistors in a CPU, more functionality can be introduced on each CPU chip. In fact, recently the *number* of CPUs on a single chip has also increased. A common chip in production at the writing of this book is the quad-core processor (see Figure 0.12) which contains four complete CPUs.

## Faster

Smaller transistors are also faster transistors, so smaller transistors provide a doubling factor: more and faster. But how does one measure speed?

When you buy a computer, the salesman is more than happy to tell you how fast it is, usually in terms of how many "gigahertz" the computer will run. The value the salesman mentions is really a measure of a special feature of every computer called its *clock*. The clock

[4] The original estimate was one year, but was later revised upward.

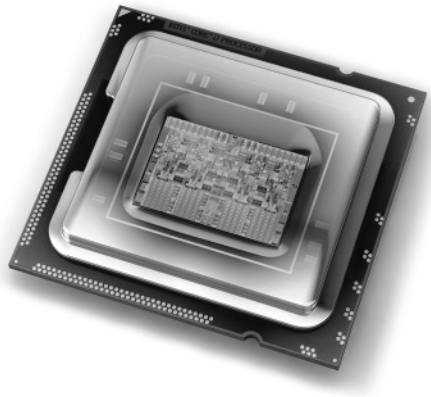


**FIGURE 0.12** Intel Nehalem Quad Core Processor. [Courtesy of Intel Corporation.]

of a computer is not much like a wall clock, however. Rather, it is more like a drummer in a band. The drummer's job in the band is to keep the beat, coordinating the rhythm of all the other members of the band. The faster the drummer beats, the faster the band plays. The components on a CPU chip are like the band—they need something to help them coordinate their efforts. That is the role of the clock. The clock regularly emits a signal indicating that the next operation is to occur. Upon every "beat" of the clock, another "cycle" occurs on the chip, meaning another set of operations can occur.

Therefore, the faster the clock runs, the faster the chip runs and, potentially, the more instructions that can be executed. But how fast is a gigahertz (GHz)? Literally, 1 gigahertz means that the clock emits a signal every nanosecond, that is, every billionth of a second ($10^{-9}$). Thus, your 1 GHz computer executes instructions once every nanosecond. That is a very fast clock indeed!

Consider that the "universal speed limit" is the speed of light, roughly 186,282 miles/second (299,792,458 meters/second) in a vacuum. Given that an electrical signal (data) is carried at this speed, how far can an electric signal travel in a nanosecond? If we do the math, it turns out that an electric signal can only travel about 11.8 inches. Only 11.8 inches! At 2 GHz, it can only travel 5.9 inches; at 4GHz, only 2.95 inches. Consider that 5.9 inches is not even the width of a typical computer board! In fact, at the writing of this book a further doubling of speed of present computers is limited by the distance electricity can travel.

Given a measure of the speed of a computer, how does that translate to actual work done by the computer? The process of measuring computing operations, known as "benchmarking," can be a difficult task. Different computing systems are better or worse, depending on how you measure the benchmark. However, manufacturers do list a measure called *instructions per second* or *IPS*. That is, it's a measure of how many instructions, such as an addition, can be done every second. In Table 0.2, we can see how the measure of increased clock rate affects the IPS (MIPS is millions of IPS). One interesting note is that, since about 2000, the

| Year | CPU | Instructions/second | Clock Speed |
|---|---|---|---|
| 1971 | Intel 4004 | 1 MIPS | 740 kHz |
| 1972 | IBM System/370 | 1 MIPS | ? |
| 1977 | Motorola 68000 | 1 MIPS | 8 MHz |
| 1982 | Intel 286 | 3 MIPS | 12 MHz |
| 1984 | Motorola 68020 | 4 MIPS | 20 MHz |
| 1992 | Intel 486DX | 54 MIPS | 66 MHz |
| 1994 | PowerPC 600s (G2) | 35 MIPS | 33 MHz |
| 1996 | Intel Pentium Pro | 541 MIPS | 200 MHz |
| 1997 | PowerPC G3 | 525 MIPS | 233 MHz |
| 2002 | AMD Athlon XP 2400+ | 5,935 MIPS | 2.0 GHz |
| 2003 | Pentium 4 | 9,726 MIPS | 3.2 GHz |
| 2005 | Xbox360 | 19,200 MIPS | 3.2 GHz |
| 2006 | PS3 Cell BE | 10,240 MIPS | 3.2 GHz |
| 2006 | AMD Athlon FX-60 | 18,938 MIPS | 2.6 GHz |
| 2007 | Intel Core 2 QX9770 | 59,455 MIPS | 3.2 GHz |
| 2011 | Intel Core i7 990x | 159,000 MIPS | 3.46 GHz |
| 2015 | Inel Core i7 5960X | 238,310 MIPS | 3.0 GHz |

**TABLE 0.2** Speed (in millions of IPS or MIPS) and clock rates of microprocessors, by year.

clock speed has not increased dramatically for the reasons mentioned previously. However, the existence of multiple CPUs on a chip still allows for increases in IPS of a CPU.

## 0.7 A HIGH-LEVEL LOOK AT A MODERN COMPUTER

Now that you know something about the low-level functional elements of a computer, it is useful to step up to a higher-level view of the elements of a computer, often termed the computer's *architecture*. The architecture is used to describe the various parts of the computer and how they interact. The standard architecture, named after the stored program model of John von Neumann, looks something like the following (see Figure 0.13):

- **Processor:** As we have mentioned, the processor is the computational heart of a computer. Often called the CPU (Central Processing Unit), it can itself consist of a number of parts including the ALU (Arithmetic and Logic Unit), where logical calculations are done; local fast storage, called the cache; connections among components, called the bus, as well as other elements.
- **Main Memory:** A processor needs data to process, and main memory is where data is stored. Main memory is traditionally volatile; that is, when power is turned off, data is lost. It is also called RAM = Random Access Memory, that is, retrievable in any order. Use of non-volatile memory is increasing, especially in portable devices.

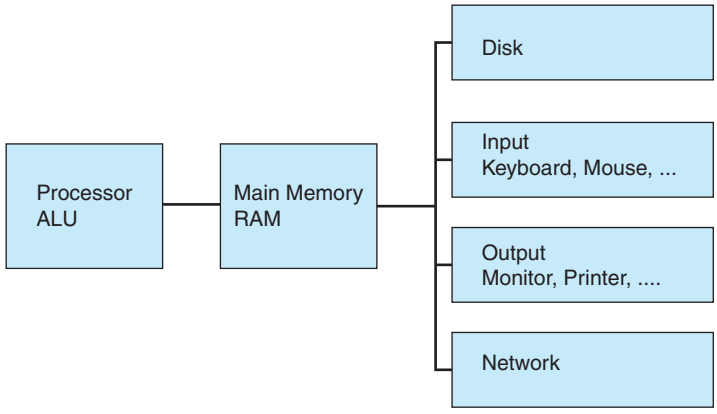



**FIGURE 0.13** A typical computer architecture.

- **Disk:** The disk is for permanent (non-volatile) storage of data. The disk is also known as the "hard drive." Data must be moved from the disk to main memory before it can be used by the processor. Disks are relatively slow, mechanical devices that are being replaced by non-mechanical memory cards in some portable devices.
- **Input/Output:** These devices convert external data into digital data and vice versa for use and storage in the computer.
- **Network:** A network is needed to communicate with other computers. From the viewpoint of the processor the network is simply another input/output device.

Consider a simple operation `theSum = num1 + num2`. The `theSum`, `num1`, and `num2` terms are called *variables*, readable names that contain values to be used in a program. The statement adds the two numbers stored in `num1` and `num2` to produce a result which is stored in `theSum`. Assume that `num1` and `num2` represent numbers that are stored on the disk and that the result `theSum` will also be stored on the disk. Assume that the instruction itself, `theSum = num1 + num2`, also resides on the disk. Here is how it works:

1. **Fetch Instruction:** When the processor is ready to process an instruction, it fetches the instruction from memory. If the instruction is not in memory but on the disk, the memory must first fetch it from the disk. In this way, the instruction `theSum = num1 + num2` will move from the disk through memory to the processor.
2. **Decode Instruction:** The processor examines the instruction ("decodes" it) and sees that operands `num1` and `num2` are needed, so it fetches them from memory. If `num1` and `num2` are not in memory, but on the disk, the memory must first fetch them from the disk.
3. **Execute Operation:** Once the processor has both the instruction and the operands, it can perform the operation, addition in this case.

4. **Store Result:** After calculating the sum, the processor will store the resulting sum in memory. At some point before power is turned off, the data in memory will be stored on the disk.
5. **Repeat:** Go back to Fetch Instruction to fetch the next instruction in a program.

The Fetch-Decode-Execute-Store cycle is fundamental for all computers. This simple, sequential process is the basis for all modern computers. These four steps are done in lock-step to the beat of a clock, as we described previously.

How complex is each operation? Not very. The ALU of a processor can add, subtract, multiply, and divide. It can also compare values and choose which instruction to do next based on that comparison. That's it. In reality, it is slightly more complex than that, but not much.

Also, the processor can handle only two types of operands: integers and floating points. For our purposes, you can think of floating-point values as fractional values represented in decimal notation, for example, 37.842. There will be a separate ALU for integers and a separate one for floating-point numbers, called the FPU = Floating-Point Unit.

The important concept is that everything a computer does boils down to a few simple operations, but at billions of operations per second, the result is significant computational power.

## 0.8 REPRESENTING DATA

The underlying element of a computer is typically a switch, usually a transistor. Given that, what is the most obvious way to represent data values? The obvious choice is binary. Binary values can only be either 1 or 0, which corresponds to the physical nature of a switch, which is on or off. What is interesting is that we can represent not only numbers in binary, but music, video, images, characters, and many other kinds of data also in binary.

### 0.8.1 Binary Data

By definition, a digital computer is binary, which is base 2. Our normal number system is decimal, base 10, probably because we have 10 fingers for counting. People haven't always worked in base 10. For example, the ancient Babylonians (2000 BC) used base 60 (sexagesimal) for the most advanced mathematics of that time. As a result, they are the source of modern time-keeping and angle measurement: 60 seconds in a minute, 60 minutes in an hour, and 360 degrees in a circle. For decimals we use ten digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. For sexagesimal the Babylonians used sixty digits: 0, 1, 2, 3, …, 58, 59. For binary there are only two digits: 0, 1.

Why binary? Two reasons, really. As we have said, the first reason is the hardware being used. Electronic transistors lend themselves very naturally to base two. A transistor is either on or off, which can be directly translated to 1 or 0. However, the second reason is that two digits are easy to store and easy to operate on. Storage devices in binary need a

medium that has two states: a state called "one" and another state "zero." Anything with two states can become digital storage. Examples include high/low voltage, right/left magnetism, charge/no-charge, on/off, and so on. For example, main memory has small capacitors that hold a charge (1) or not (0). Disks are magnetized one way (1) or the other (0). CDs and DVDs reflect light one way (1) or the other (0).

Manipulations of the underlying data can also be done simply using electronic gates that we discussed previously, that is, the Boolean logic: and, or, not. Because such logical circuits can be extremely small and fast, they can be implemented to do calculations quickly and efficiently. For example, the adder circuit we discussed previously that adds two binary digits or *bits* (bit = BInary digiT) can be done with logical circuits: sum = (A and (not B)) or ((not A) and B)). From such simple logic all arithmetic operations can be built. For example, subtraction is the addition of a negative value, multiplication is repeated addition, and division can be done using the other three operations. A choice can be made based on the value of a bit: choose one thing or another. That choice bit can be calculated using any arbitrary logical expression using the logical operators: and, or, not. Therefore, the entire ALU (and the rest of the computer) can be built from the logical operators: and, or, not.

## 0.8.2 Working with Binary

A brief look at the binary representation of numbers and characters provides useful background for understanding binary computation. Because our world is a world of decimal numbers, let's look at representing decimals in binary. We begin with a review of place holding in decimals, by taking you back to elementary school. For example, consider the number 735 in base 10 (written as $735_{10}$). Notice in the last line how the exponents start at 2 and work down to 0 as you move from left to right.[5]

$$\begin{aligned} 735_{10} &= 7\ \mathit{hundreds} + 3\ \mathit{tens} + 5\ \mathit{ones} \\ 735_{10} &= 7 * 100 + 3 * 10 + 5 * 1 \\ 735_{10} &= 7 * 10^2 + 3 * 10^1 + 5 * 10^0 \end{aligned}$$

In binary we only have two digits: 0, 1. Also, our base is 2 rather than 10. Therefore, our rightmost three places are fours, twos, ones. As with base 10, the exponents will decrease as we move from left to right—in this case: 2, 1, then 0. Using the previous notation, but working backwards from the last line to the first line, let us determine what 101 in binary ($101_2$) is in base 10 (decimal).

$$\begin{aligned} 101_2 &= 1 * 2^2 + 0 * 2^1 + 1 * 2^0 \\ 101_2 &= 1 * 4 + 0 * 2 + 1 * 1 \\ 101_2 &= 4 + 0 + 1 \\ 101_2 &= 5_{10} \end{aligned}$$

[5] We use an asterix(*) to represent multiplication.

In a similar way, any decimal integer can be represented in binary. For example,

$$1052_{10} = 10000011100_2$$

Fractions can be represented using integers in the scientific notation that you learned in science classes:

$$1/8 = 0.125 = 125 * 10^{-3}$$

The mantissa (125) and exponent (−3) are integers that can be expressed and stored in binary. The actual implementation of binary fractions is different because the starting point is binary, but the principle is the same: binary fractions are stored using binary mantissas and binary exponents. How many bits are allocated to mantissa and how many to exponents varies from computer to computer, but the two numbers are stored together.

There are four important concepts that you need to know about representation:

- All numbers in a computer are represented in binary.
- Because of fixed, hardware limits on number storage, there is a limit to how big an integer can be stored in one unit of computer memory (usually 32 or 64 bits of storage).
- Fractions are represented in scientific notation and are approximations.
- Everything is converted to binary for manipulation and storage: letters, music, pictures, and so on.

## 0.8.3 Limits

We have covered the representation of numbers in binary. Let's look at limits. Most computers organize their data into *words* that are usually 32 bits in size (though 64-bit words are growing in use). There are an infinite number of integers, but with only 32 bits available in a word, there is a limited number of integers that can fit. If one considers only positive integers, one can represent $2^{32}$ integers with a 32-bit word, or a little over 4 billion integers. To represent positive and negative integers evenly, the represented integers range from negative 2 billion to positive 2 billion. That is a lot of numbers, but there is a limit and it is not hard to exceed it. For example, most U.S. state budgets will not fit in that size number (4 billion). On the other hand, a 64-bit computer could represent $2^{64}$ integers using a 64-bit word, which in base 10 is $1.8 \times 10^{19}$: a huge number—over 4 billion times more than can be stored in a 32-bit word.

Fractional values present a different problem. We know from mathematics that between every pair of Real numbers there are an infinite number of Real numbers. To see that, choose any two Real numbers A and B, and (A + B)/2 is a Real number in between. That operation can be repeated an infinite number of times to find more Real numbers between A and B. No matter how we choose to represent Real numbers in binary, the representation will always be an approximation. For example, if you enter 19/5.0 into Python, it will be calculated as 3.7999999999999998 instead of the expected 3.8 (try it!). The approximation is a feature of storage in binary rather than a feature of Python.

### Bits, Bytes, and Words

Computer words (as opposed to English words) are built from *bytes*, which contain 8 bits so one 32-bit word is made of 4 bytes. Storage is usually counted in bytes, for example, 2 GB of RAM is approximately 2 billion bytes of memory (actually, it is $2^{31}$ bytes, which is 2,147,483,648 bytes). Bytes are the unit of measurement for size mainly for historical reasons.

## 0.8.4 Representing Letters

So far we've dealt with numbers. What about letters (characters): how are characters stored in a computer? Not surprisingly, everything is still stored in binary, which means that it is still a number. Early developers created ways to map characters to numbers.

First, what is a character? Characters are what we see on a printed page and are mostly made from letters (a, b, c, …), digits (0, 1, 2, …), and punctuation (period, comma, semicolon, …). However, there are also characters that are not printable such as "carriage return" or "tab" as well as characters that existed at the dawn of computing to control printers such as "form feed." The first standardized set of computer characters was the ASCII (American Standard Code for Information Interchange) set developed in 1963. ASCII served the English-speaking world, but could not handle other alphabets. As computing became more universal the restrictions of the 128 ASCII characters became a problem. How could we handle the tens of thousands of Chinese characters? To address this problem a universal encoding named Unicode was first defined in 1991 and can handle over 1 million different characters. One implementation of Unicode is UTF-8 which is popular because it is backward compatible to the ASCII encoding that dominated the first 30 years of computing. Currently over 85% the pages on the World Wide Web use UTF-8. It is the default character representation for Python 3. For those reasons this text will use UTF-8.

Table 0.3 has part of the UTF-8 encoding showing some English characters, some symbols, and some control characters. The table shows a mapping between numbers and

| Char | Dec | Char | Dec | Char | Dec | Char | Dec |
|---|---|---|---|---|---|---|---|
| NUL | 0 | SP | 32 | @ | 64 | ` | 96 |
| SOH | 1 | ! | 33 | A | 65 | a | 97 |
| STX | 2 | " | 34 | B | 66 | b | 98 |
| ETX | 3 | # | 35 | C | 67 | c | 99 |
| EOT | 4 | $ | 36 | D | 68 | d | 100 |
| ENQ | 5 | % | 37 | E | 69 | e | 101 |
| ACK | 6 | & | 38 | F | 70 | f | 102 |
| BEL | 7 | ' | 39 | G | 71 | g | 103 |
| BS | 8 | ( | 40 | H | 72 | h | 104 |

**TABLE 0.3** Table of UTF-8 characters (first few rows).

characters. Each character has an associated number, that is "A" is 65 while "a" is 97. Knowing this relationship, we can interpret a set of numbers as characters and then manipulate those characters just as we would numbers. We'll talk more about this topic in Chapter 4.

## 0.8.5 Representing Other Data

You must get the idea by now: what computers can represent is numbers. If you want to represent other data, you must find a way to encode that data as numbers.

### Images

How to store an image? If the image is discrete—that is, built of many individual parts—we can represent those individual parts as numbers. Take a close look at your monitor or TV screen (with a magnifying glass, if you can). The image is made up of thousands of very small dots. These dots, called *pixels* (short for picture elements), are used to create an image. Each pixel has an associated color. If you put a lot of these pixels together in a small area, they start to look like an image (see Figure 0.14).

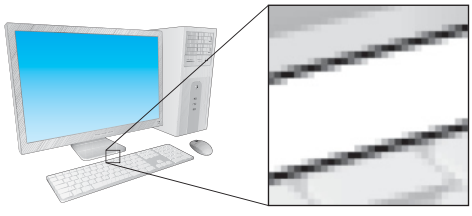


**FIGURE 0.14** A computer display picture with a close-up of the individual pixels. [Juliengrondin/Shutterstock]

Each pixel can be represented as a location (in a two-dimensional grid) and as a color. The location is two numbers (which row and which column the pixel occupies), and the color is also represented as a number. Although there are a variety of ways to represent color, a common way is to break each color into its contribution from the three basic colors: red, green, and blue. The color number represents how much of each basic color is contributed to the final color. An 8-bit color scheme means that each color can contribute 8 bits, or $2^8 = 256$ possible shades of that color. Thus, a 24-bit color system can represent $2^{24}$ different colors, or 16,777,216.

The quantity of pixels is important. Standard analog television (extinct as of the writing of this book) used 525 lines of pixels, with each line containing 480 pixels, a total of

252,000 pixels. High-definition television has 1920 × 1080 pixels for a total of 2,073,600, a much higher resolution and a much better image.

### Music

How to represent music as numbers? There are two types of musical sound that we might want to capture: recorded sound and generated sound. First, let's look at recording sound. A sound "wave" is a complex wave of air pressure that we detect with our ears. If we look at the shape of this sound wave (say with an oscilloscope), we can see how complex the wave can be. However, if we record the height of that wave at a very high rate—say, at a rate of 44,100 times/second (or 44kHz, the sampling rate on most MP3s)—then we can record that height as a number for that time in the sound wave. Therefore, we record two numbers: the height of the recorded sound and the time when that height was recorded. When we play sound back, we can reproduce the shape of the sound wave by creating a new sound wave that has the same height at the selected times as the recorded sound at each point in time. The more often we "sample" the sound wave, the better our ability to reproduce it accurately. See Figure 0.15 for an example.

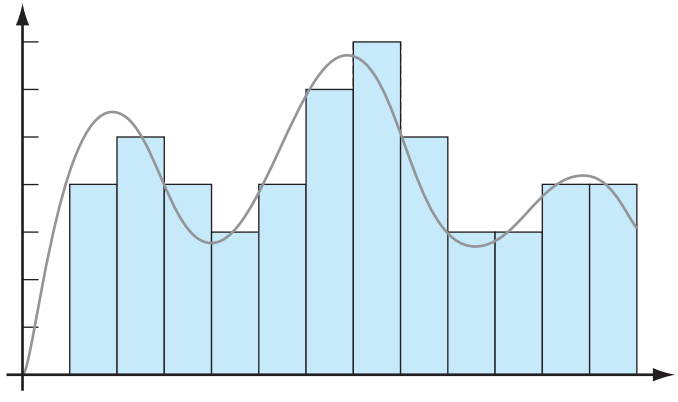


**FIGURE 0.15** A sound wave and the samples of that sound wave (green bar height) over time.

To generate our own new sound, we can write computer programs that generate the same data, a wave height at some point in time, and then that data can be played just like recorded sound.

## 0.8.6 What Does a Number Represent?

We've shown that all the data we record for use by a computer is represented as a number, whether the data is numeric, text, image, audio, or any other kind of data. So how can you tell

what a particular recorded data value represents? You cannot by simply looking at the bits. Any particular data value can be interpreted as an integer, a character, a sound sample, and so on. It depends on the use for which that data value was recorded. That is, it depends on the *type* of the data. We will talk more of types in Chapter 1, but the type of the data indicates for what use the data values are to be used. If the type is characters, then the numbers represent UTF-8 values. If the type is floating-point, then the numbers represent the mantissa and exponent of a value. If the type is an image, then the number would represent the color of a particular pixel. Knowing the type of the data lets you know what the data values represent.

## 0.8.7 How to Talk About Quantities of Data

How much data can a computer hold, and what constitutes "a lot" of data? The answer to that question varies, mostly depending on when you ask it. Very much like Moore's law and processing speed, the amount of data that a commercial disk can hold has grown dramatically over the years.

Again, there are some terms in common usage in the commercial world. In the context of data amounts, we talk about values like "kilobytes" (abbreviated KB) or "megabytes" (abbreviate MB) or "gigabytes" (abbreviated GB), but the meaning is a bit odd. "Kilo" typically refers to $10^3$ or 1 thousand of something, "mega" $10^6$ or 1 million of something, and "giga" usually to $10^9$ or 1 billion of something, but its meaning here is a little off. This is because of the fact that $10^3$ or 1000 is *pretty close* to $2^{10}$ or 1024.

So in discussing powers of 2 and powers of 10, most people use the following rule of thumb. Any time you talk about multiplying by $2^{10}$, that's pretty close to multiplying $10^3$, so we will just use the power of 10 prefixes we are used to. Another way to say it is that every 3 powers of 10 is "roughly equivalent" to 10 powers of 2. Thus a kilobyte is not really 1000 ($10^3$) bytes, but 1024 ($2^{10}$) bytes. A megabyte is not 1 million bytes ($10^6$), but 1,048,576 ($2^{20}$) bytes. A gigabyte is not 1 billion bytes ($10^9$), but 1,073,741,824 bytes ($2^{30}$).

## 0.8.8 How Much Data Is That?

At the writing of this book, the largest standard size commercial disk is 1 terabyte, 1 trillion bytes ($10^{12}$). On the horizon is the possibility of a 1 petabyte, 1 quadrillion ($10^{15}$) bytes. Again, the growth in disk sizes is dramatic. The first disk was introduced by IBM in 1956 and held only 4 megabytes. It took 35 years before a 1 gigabyte disk was made, but only 14 more years until we had 500 gigabyte disks, and only two more years until a 1 terabyte (1,000 gigabyte) disk was available. A petabyte disk (1,000,000 gigabytes) may happen soon.

So, how big is a petabyte? Let's try to put it into terms you can relate to:

- A book is roughly a megabyte of data. If you read one book a day every day of your life, say 80 years, that will be less than 30 gigabytes of data. Because a petabyte is 1 million gigabytes, you will still have 999,970 gigabytes left over.
- How many pictures can a person look at in a lifetime? If we assume 4 megabytes per picture and 100 images a day, after 80 years that collection of pictures would add up to 30 terabytes. So your petabyte disk will have 970,000 gigabytes left after a lifetime of photos and books.
- What about music? MP3 audio files run about a megabyte a minute. At that rate, a lifetime of listening—all day and all night for 80 years—would consume 42 terabytes of space. So with a lifetime of music, pictures, and books, you will have 928,000 gigabytes free on your disk. That is, almost 93% of your disk is still empty.
- The one kind of content that might overflow a petabyte disk is video. For DVDs the data rate is about two gigabytes per hour. Therefore, the petabyte disk will hold about 500,000 hours worth of video. If you want to record 24 hours a day, 7 days a week, the video will fill up your petabyte drive after about 57 years.

Of course, why stop there? More prefixes have been defined:

- exa is $2^{60} = 1{,}152{,}921{,}504{,}606{,}846{,}976$
- zetta is $2^{70} = 1{,}180{,}591{,}620{,}717{,}411{,}303{,}424$
- yotta is $2^{80} = 1{,}208{,}925{,}819{,}614{,}629{,}174{,}706{,}176$

Does anything get measured in exabytes or even zetabytes? Yes, it does though such a large measure is not (yet) applied to the the capacity of a single device. It is, however, applied to global measures such as Internet traffic.

Cisco Systems Inc., one of the leading providers of network equipment, has provided a white paper on the growth of Internet traffic. Their 2014–2019 forecast[6] made the following report:

- In **2014**, global internet traffic was 59.9 exabytes **per month**. By **2019** they estimate that rate will climb to 168 exabytes per month.
- By **2019**, global internet traffic will reach 2 zetabytes/year.

The storage company EMC has sponsored the IDC Digital Universe study to estimate the amount of data in use. Figure 0.16 shows a decade of estimates that reach 8 zettabytes by 2015 and their study estimates that to grow to 35 zettabytes by 2020. As fanciful as those numbers seem, they illustrate a practical use of the term zettabyte.

[6] http://www.cisco.com/c/en/us/solutions/collateral/service-provider/ip-ngn-ip-next-generation-network/white_paper_c11-481360.html

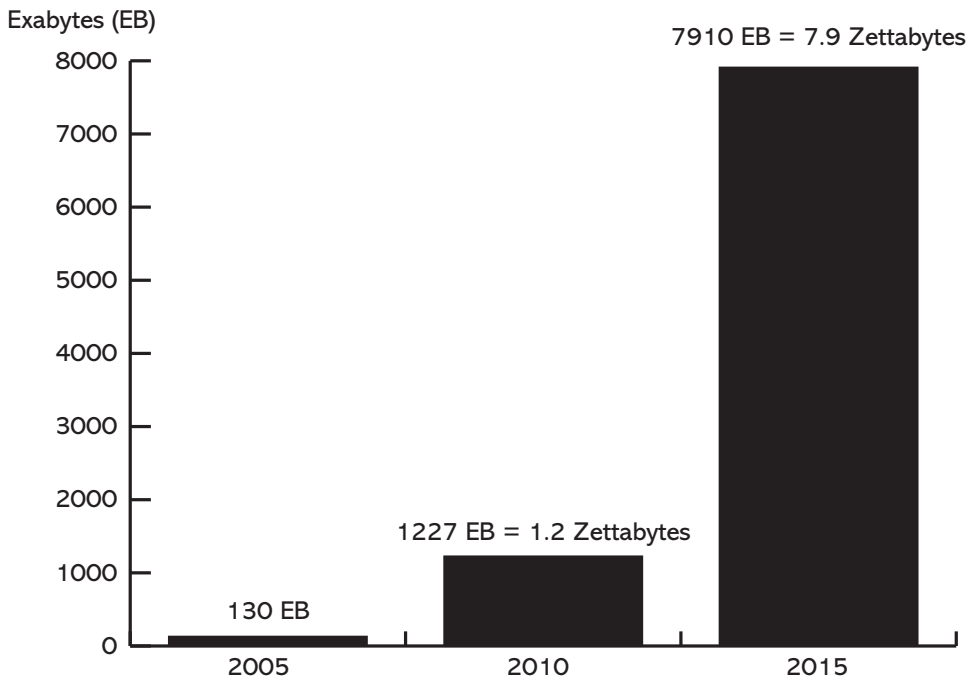



**FIGURE 0.16** Estimates of data created and stored. [Data from The 2011 IDC Digital Universe Study]

## 0.9 OVERVIEW OF COMING CHAPTERS

This text is divided into five parts. The first gets you started on Python, computing, and problem solving. With that in hand, we get down into details in the next part where we develop both the Python language and problem-solving skills sufficient to solve interesting problems. The third part provides more tools in the form of Python built-in data structures, algorithm and problem-solving development, and functions. Part 4 shows you how to build classes—your own data structures. The final part includes more on Python.

## Summary

In this chapter, we considered ways that data can be represented and manipulated—at the hardware level. In subsequent chapters, we will introduce ways in which you, as a programmer, can control the representation and manipulation of data.

PART 2

# Starting to Program

**Chapter 1** Beginnings

**Chapter 2** Control

**Chapter 3** Algorithms and Program Development

This page intentionally left blank

CHAPTER 1

# Beginnings

> A good workman is known by his tools.
>
> Proverb

OUR FIRST STEPS IN PROGRAMMING ARE TO LEARN THE DETAILS, SYNTAX, and semantics of the Python programming language. This necessarily involves getting into some of the language details, focusing on level 1 (language) issues as opposed to level 2 (problem solving) issues. Don't worry: we haven't forgotten that the goal is to do effective problem solving, but we have to worry about both aspects, moving between levels as required. A little proficiency with Python will allow us to write effective, problem-solving programs.

Here are our first two **RULES** of programming:

> **Rule 1:** Think before you program!
> **Rule 2:** A program is a human-readable essay on problem solving that also happens to execute on a computer.

## 1.1 PRACTICE, PRACTICE, PRACTICE

Let's start experimenting with Python. Before we get too far along, we want to emphasize something important. One of the best reasons to start learning programming using Python is that you can easily experiment with Python. That is, Python makes it easy to try something out and see the result. Anytime you have a question, simply try it out.

Learning to experiment with a programming language is a very important skill and one that seems hard for introductory students to pick up. So let's add a new **RULE**.

> **Rule 3:** The best way to improve your programming and problem skills is to practice!

Problem solving—and problem solving using programming to record your solution—requires practice. The title of this section is the answer to the age-old joke:

> **Student:** How do you get to Carnegie Hall?
> **Teacher:** Practice, practice, practice!

Learning to program for the first time is not all that different from learning to kick a soccer ball or play a musical instrument. It is important to learn about fundamentals by reading and talking about them, but the best way to *really* learn them is to practice. We will encourage you throughout the book to type something in and see what happens. If what you type in the first time doesn't work, who cares? Try again; see if you can fix it. If you don't get it right the first time, you will eventually. *Experimenting* is an important skill in problem solving, and this is one place where you can develop it!

We begin with our first QUICKSTART. A QUICKSTART shows the development of a working program followed by a more detailed explanation of the details of that program. A QUICKSTART gets us started, using the principal of "doing" before "explaining." Note that we ask you to try some things in the Python shell as we go along. Do it! Remember **RULE 3**. (To get Python see Appendix A.)

## 1.2 QUICKSTART, THE CIRCUMFERENCE PROGRAM

Let's start with a simple task. We want to calculate the circumference and area of a circle given its radius. The relevant mathematical formulas are:

- circumference = $2 * \pi * radius$
- area = $\pi * radius^2$

To create the program, we need to do a couple of things:

1. We need to prompt the user for a radius.
2. We need to apply the mathematical formulas listed previously using the acquired radius to find the circumference and area.
3. We need to print out our results.

Here is Code Listing 1.1. Let's name it `circumference.py`. The ".py" is a file *suffix*.

### PROGRAMMING TIP

Most computer systems add a suffix to the end of a file to indicate what "kind" of file it is—what kind of information it might store: music (".mp3"), pictures (".jpg"), text (".txt"), etc. Python does the same and expects that a Python file have a ".py" suffix. The SPYDER editor (see Appendix A) is pretty helpful in that it, by default, saves the files you edit with a ".py" suffix. However, if you choose to save it as a ".txt" file or something else, then the SPYDER's color scheme for your code disappears. Those colors are useful in that each color indicates a type of thing (green for strings, purple for keywords) in the program, making it more readable.

**Code Listing 1.1**

```
1  # calculate the area and circumference of a circle from its radius
2  # Step 1: prompt for a radius
3  # Step 2: apply the area formula
4  # Step 3: Print out the results
5
6  import math
7
8  radius_str = input("Enter the radius of your circle: ")
9  radius_int = int(radius_str)
10
11 circumference = 2 * math.pi * radius_int
12 area = math.pi * (radius_int ** 2)
13
14 print ("The cirumference is:",circumference,  \
15       ", and the area is:",area)
```

**Important:** The line numbers shown in the program are *not* part of the program. We list them here only for the reader's convenience.

Before we examine the code, let's illustrate how the program runs with two different input radii to confirm that it works properly.

The easiest way to run a program is to open that program in the SPYDER editor, then select Run → Run (F5). If you haven't saved the file yet, you will be prompted where to save it. Then, you will see a `runfile` message in the iPython console. The `runfile` *imports* the file into the shell and runs it. Note that every time you run the program, the iPython console shell prints the `runfile` line, indicating that the shell is restarting and running your program.

You can choose some "obvious" values to see if you get expected results. For example, a radius value of 1 results in the area having the recognizable value of $\pi = 3.14159\ldots$. Although not a complete test of the code, it does allow us to identify any gross errors quickly. The other case has a radius of 2 that can be easily checked with a calculator:

```
In  [1]: runfile('/Users/user/python/file.py', wdir='/Users/user/python')

Enter the radius of your circle: 1
The cirumference is: 6.283185307179586 , and the area is: 3.141592653589793


In  [2]: runfile('/Users/user/python/file.py', wdir='/Users/user/python')

Enter the radius of your circle: 2
The cirumference is: 12.566370614359172 , and the area is: 12.566370614359172
```

### 1.2.1 Examining the Code

Let's examine the circumference.py code line-by-line. In this first example, there are many topics that we will touch on briefly, but will be explained in more detail in subsequent chapters. Note that we number only every fifth line in the code listing. Here is a walk through of this code.

**Lines 1–4:** Anything that follows a pound sign (#) is a comment for the human reader. The Python interpreter ignores it. However, it does provide us as readers some more information on the intent of the program (more on this later). Remember **RULE 2**. Comments help make your document easier to understand for humans.

**Line 6:** This line imports special Python code from the math *module*. A module is a Python file containing programs to solve particular problems; in this case, the `math` module provides support for solving common math problems. Modules are described in more detail in Section 1.4.1. Python has many such modules to make common tasks easier. In this case we are interested in the value $\pi$ provided by the math module, which we indicate in the program using the code `math.pi`. This is a naming convention we will explore in more detail in Section 1.8, but essentially the code `math.pi` means—that within the module named `math` there is a value named `pi`, with a "." separating the module and value name.

**Line 8:** This line really has two parts: the Python code to the right of the `=` sign and the Python code on the left:

- On the right, `input` is a small Python program called a *function*. Functions (see Chapter 8) are often-used, small program utilities that do a particular task. The `input` function prints the characters in quotes, "Enter the radius of your circle:", to the Python shell and waits for the user to type a response. Whatever the user types in the shell before pressing the Enter key at the end is *returned*, that is, provided as input to a program.
- On the left side of the `=` is a *variable*. For now, consider a variable to be a name that is associated with a value. In this case, the value returned from `input` will be associated with the name `radius_str`. (Programmers traditionally shorten "string" to "str".)

  Think of the `=` as a kind of glue, linking the values on the right side with the variable on the left. The `=` is called an *assignment statement*; we will have more to say about assignment in Section 1.5.1.

**Line 9:** The user's response returned by `input` is stored as a sequence of characters, referred to in computer science as a *string* (see Chapter 4). Python differentiates a sequence of characters, such as those that constitute this sentence, from numbers on which we can perform operations such as addition, subtraction and so on. Strings are differentiated from numbers by using quotes, single or double quotes are acceptable ("hi mom" or `'monty'`). For this program we want to work with numbers, not characters, so we must convert the user's response from a string of characters to numbers. The `int` function takes the value associated with the variable `radius_str` and returns the

integer value of `radius_str`. In other words, it converts the string to an integer. As in the previous line, the value returned by ***`int`*** is associated with a variable using the assignment statement. For example, if `radius_str` holds the string of characters `"27"`, the ***`int`*** function will convert those characters to the integer 27 and then associate that value the new variable name `radius_int`.

The difference between the characters `"27"` and the number 27 will likely seem strange at first. It is our first example of value *types*. Types will be described in more detail in Section 1.6, but for now, let's say that a type indicates the kinds of things we can do to a value and the results we obtain.

**Line 11:** Here we calculate the circumference using the following formula:

```
circumference = 2 * pi * radius
```

While +, -, and / mean what you expect for math operations (addition, subtraction, and division), we use the * symbol to represent multiplication as opposed to · or ×. This is to avoid any confusion between "x" and × or "." and "·". The integer 2, the value associated with the variable `math.pi`, and the value associated with the `radius_int` are multiplied together, and then the result is associated with the variable named `circumference`. As you will later see, the ordering is important: the mathematical expression on the right-hand side of the equal sign is evaluated, and then the result is associated with the variable on the left-hand side of the equal sign.

**Line 12:** Similarly, we calculate the area using the formula listed previously. There isn't a way to type in an exponent (superscript) on a keyboard, but Python has an exponentiation operator `**` by which the value on the left is raised to the power on the right. Thus `radius_int ** 2` is the same as `radius_int` squared or $\texttt{radius\_int}^2$. Note that we use parentheses to group the operation. As in normal math, expressions in parentheses are evaluated first, so the expression `math.pi * (radius_int ** 2)` means square the value of `radius_int`, then take that result and multiply it by *pi*.

**Lines 14 and 15:** We print the results using the Python ***`print`*** statement. Like the `input` statement, ***`print`*** is a function that performs a much used operation, printing values to the user console. The print statement can print strings bracketed by quotes (either single or double quotes) and a value associated with a variable. Printable elements are placed in the parentheses after the ***`print`*** statement. If the element being printed is quoted, it is printed exactly as it appears in the quotes; if the element is a variable, then *the value associated with the variable* is printed. Each object (string, variable, value, etc.) that is to be printed is separated from other objects by commas. In this case, the ***`print`*** statement outputs a string, a variable value, another string, and finally a variable value. The backslash character (\) indicates that the statement continues onto the next line—in this case, the two-line ***`print`*** statement behaves as if it were one long line. Stretching a statement across two lines can enhance readability—particularly on a narrow screen or page. See Section 1.4.3.

## 1.3 AN INTERACTIVE SESSION

An important feature of Python, particularly when learning the language, is that it is an *interpreted* language. By interpreted we mean that there is a program within Python called the interpreter that takes each line of Python code, one line at a time, and executes that code. This feature allows us to try out lines of code one at a time by typing into the Python shell. The ability to experiment with pieces of code in the Python shell is something that really helps while you're learning the language; you can easily try something out and see what happens. That is what is so great about Python—you can easily explore and learn as you go.

Consider our circumference program. The following code shows a session in a Python shell in which a user types each line of the program we listed previously to see what happens. We also show a few other features, just to experiment. *Open up a Python shell and follow along by trying it yourself.* The comments (text after #, the pound sign), are there to help walk you through the process. There is no need to type them into the Python shell.

```
In  [1]: import math
In  [2]: radius_str = input("Enter the radius of your circle: ")
Enter the radius of your circle: 20
In  [3]: radius_str # what is the value associated with radius_str
Out [3]: '20'

In  [4]: radius_int = int(radius_str) # convert the string to an integer
In  [5]: radius_int # check the value of the integer
Out [5]: 20                    # look, no quotes because it is a number

In  [6]: int(radius_str) # what does int() return without assignment (=)
Out [6]: 20

In  [7]: radius_str # int() does not modify radius_str!!
Out [7]: '20'

In  [8]: math.pi # let's see what value is associated with math.pi
Out [8]: 3.141592653589793

In  [9]: circumference = 2 * math.pi * radius_int # try our formula
In  [10]: circumference
Out [10]: 125.66370614359172

In  [11]: area = math.pi * radius_int ** 2 # area using exponentiation
In  [12]: area
Out [12]: 1256.6370614359173

In  [13]: math.pi * radius_int ** 2 # area calculation without assignment
Out [13]: 1256.6370614359173

In  [14]: print("Circumference: ", circumference, ", area: ", area)
Circumference:  125.66370614359172 , area:  1256.6370614359173
```

Within the Python shell, you can find out the value associated with any variable name by typing its name followed by the Enter key. For example, when we type `radius_str` in the shell as shown in the example, `'20'` is output. The quotes indicate that it is a string of characters, `'20'`, rather than the integer 20.

Interestingly, Python treats single quotes (`'20'`) the same as double quotes (`"20"`). You can choose to designate strings with single or double quotes. It's your choice!

We can see that after the conversion of the string to an integer using the `int` function, simply typing `radius_int` results in the integer 20 being printed (no quotes).

We can also try the expression `int(radius_str)` without assigning the result to `radius_int`. Note that `radius_str` is unchanged; it is provided as a value for the `int` function to use in its calculations. We then check the value of *pi* in the `math` module named `math.pi`. Next we try out the circumference and area formulas. The area calculation is then performed using the exponentiation operator (`**`), raising `radius_int` to the second power. Finally, we try the `print` statement. Remember **RULE 3.**

An advantage of Python over other languages such as C, C++, and Java is the Python shell. Take advantage of this feature because it can greatly enhance your learning of what the Python language can do. If you wonder about how something might or might not work, try it in the Python shell!

## 1.4 PARTS OF A PROGRAM

**RULE 2** describes a program as an essay on problem solving that is also executable. A program consists of a set of *instructions* that are executed sequentially, one after the other in the order in which they were typed. We save the instructions together in a *module* for storage on our file system. Later, the module can be *imported* into the Python interpreter, which runs programs by executing the instructions contained in the module.

### 1.4.1 Modules

- A *module* contains a set of Python commands.
- A module can be stored as a file and *imported* into the Python shell.
- Usage:

```
import module  # load the module
```

Hundreds of modules come with the standard Python distribution; the math module and many more can be found and imported. You can even write your own modules and use them as tools in your own programming work!

### 1.4.2 Statements and Expressions

Python differentiates code into one of two categories: *expressions* or *statements*. The concept of an expression is consistent with the mathematical definition, so it may be familiar to you.

**Expression:** A combination of values and operations that creates a new value that we call a *return value*, that is, the value returned by the operation(s). If you enter an expression into

the Python shell, a value will be returned and displayed; that is, the expression `x + 5` will display 7 if the value of `x` is 2. Note that the value associated with `x` is not changed as a result of this operation!

**Statement:** Does not *return a value*, but does perform some task. Some statements may control the flow of the program, and others might ask for resources; statements perform a wide variety of tasks. As a result of their operation, a statement may have a *side effect*. A side effect is some change that results from executing the statement. Take the example of the *assignment statement* `my_int = 5` (shown in the following example). This statement does not have a *return value*, but it does set the value associated with the variable `my_int` to 5, a *side effect*. When we type such an assignment into the Python shell, no value is returned, as you can see here:

```
In  [1]: my_int = 5 # statement, no return value but my_int now has value 5
In  [2]: my_int
Out [2]: 5

In  [3]: my_int + 5 # expression, value associated with my_int added to 5
Out [3]: 10

In  [4]: my_int # no side effect of expression, my_int is unchanged
Out [4]: 5
```

However, after we type the assignment statement, if we type the variable `my_int`, we see that it does indeed now have the value of 5 (see the example session). A statement never returns a value, but some statements may have a side effect. You will see more of this behavior as we explore the many Python statements.

### PROGRAMMING TIP

Knowing that an expression has a value but a statement does not is useful. For example, you can print the value generated by an expression, for example, `print(x + 5)` (as long as `x` has a value). However, if you try to print a statement, Python generates an error. If no value is returned from a statement, then what will be the print output? Python avoids this by not allowing a statement to be printed. That is, it is not allowable syntax.

```
In  [1]: print(x + 5) # printing an expression
7
In  [2]: print(y = x + 5) # trying to print a statement
TypeError: 'y' is an invalid keyword argument for this function
```

There are a number of instances of entering expressions and statements into the Python shell in the previous examples. Whenever an expression was entered, its return value was printed on the following line of the console. Whenever a statement was entered, nothing was

printed: if we wanted to see what a statement changed (side effect), we, the programmers, have to inquire about the change.

## 1.4.3 Whitespace

When we type, we usually separate words with what is typically called *whitespace*. Python counts as whitespace the following characters: space, tab, return, linefeed, formfeed and vertical tab. Python has the following rules about how whitespace is used in a program:

- Whitespace is *ignored* within both expressions and statements.
  For example, `Y=X+5` has exactly the same meaning as `Y  =  X  +  5`.
- *Leading* whitespace, whitespace at the beginning of a line—defines *indentation*. Indentation plays a special role in Python (see the following section).
- Blank lines are also considered to be whitespace, and the rule for blank lines is trivial: blank lines are allowed anywhere and are ignored.

### Indentation

Indentation is used by all programmers to make code more readable. Programmers often indent code to indicate that the indented code is *grouped together*, meaning those statements have some common purpose. However, indentation is treated uniquely in Python. Python *requires* it for grouping. When a set of statements or expressions needs to be grouped together, Python does so by a consistent indentation. Grouping of statements will be important when we discuss control statements in Chapter 2.

Python requires consistency in whitespace indentation. If previous statements use an indentation of four spaces to group elements, then that must be done consistently throughout the program.

The benefit of indentation is *readability*. While other programming languages encourage indentation, Python's whitespace indentation *forces* readability. The disadvantage is that maintaining consistency with the number of spaces versus tabs can be frustrating, especially when cutting and pasting. Fortunately, Python-aware editors such as SPYDER automatically indent and can repair indentation.

PROGRAMMING TIP

SPYDER provides a command under the Source → Fix Indentation menu to help fix indentation. If you are having problems with indentation, or if Python is complaining about irregular indentation, try this command to repair the problem.

### Continuation

Long lines of code—those wider than the width of a reasonably sized editing window—can make reading code difficult. Because readability is very important (remember **RULE 2**), Python provides ways to make long lines more readable by splitting them. Such splitting is

called a *continuation*. If a single statement runs long, that statement can be continued onto another line (or lines) to help with readability. That is, a continued line is still a single line, it just shows up over multiple lines in the editor. Indicate a continuation by placing a backslash character (\) at the end of a line. Multiple-line expressions can be continued similarly. In this way, a long line that may be difficult to read can be split in some meaningful and readable way across multiple lines. The first program in this chapter, `circumference.py`, used such a backslash character in the print statement.

## 1.4.4 Comments

> Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.
>
> Donald Knuth[1]

We will say many times, in different ways, that a program is more than "just some code that does something." A program is a document that describes the thought process of its writer. Messy code implies messy thinking and is both difficult to work with and to understand. Code also happens to be something that can run, but just because it can run does not make it a good program. Good programs can be read, just like any other essay. Comments are one important way to improve *readability*. Comments contribute nothing to the running of the program because Python ignores them. In Python, anything following a pound character (#) is ignored on that line. However, comments are critical to the readability of the program.

There are no universally agreed-upon rules for the right style and number of comments, but there is near-universal agreement that they can enhance readability. Here are some useful guidelines:

- The *why* philosophy: "Good comments don't repeat the code or explain it. They clarify its intent. Comments should explain, at a higher level of abstraction than the code, what you're trying to do."[2] (*Code Complete* by McConnell)
- The *how* philosophy: If your code contains a novel or noteworthy solution, add comments to explain the methodology.

## 1.4.5 Special Python Elements: Tokens

As when learning any language, there are some details that are good to know before you can fully understand how they are used. Here we show you the special keywords, symbols, and characters that can be used in a Python program. These language elements are known generically as *tokens*. We don't explain in detail what each one does, but it is important to note that Python has special uses for all of them. You will become more familiar with them as

[1] Excerpt from "Literate Programming" by Donald E. Knuth in *The Computer Journal*, 27(2), pp. 97–111. Published by Oxford University Press, © 1984.

[2] Excerpt from "Code Complete: A Practical Handbook of Software Construction, 2e" by Steve McConnell. Published by Microsoft Press, © 2004.

we proceed through the book. More importantly than what they do is the fact that Python reserves them for its own use. You can't redefine them to do something else. Be aware they exist so that you don't accidentally try to use one (as a variable, function, or the like).

## Keywords

*Keywords* are special words in Python that cannot be used by you to name things. They indicate commands to the Python interpreter. The complete list is in Table 1.1. We will introduce commands throughout the text, but for now know that you cannot use these words as names (of variables, of functions, of classes, etc.) in your programs. Python has already taken them for other uses.

| | | | | |
|---|---|---|---|---|
| `and` | `del` | `from` | `not` | `while` |
| `as` | `elif` | `global` | `or` | `with` |
| `assert` | `else` | `if` | `pass` | `yield` |
| `break` | `except` | `import` | `print` | `class` |
| `exec` | `in` | `raise` | `continue` | `finally` |
| `is` | `return` | `def` | `for` | `lambda` |
| `try` | `True` | `False` | `None` | |

**TABLE 1.1** Python keywords.

## Operators

*Operators* are special tokens (sequences of characters) that have meaning to the Python interpreter. Using them implies some operation, such as addition, subtraction, or something similar. We will introduce operators throughout the text. The complete list is in Table 1.2. You can probably guess many of them, but some of them will not be familiar.

| | | | | | | |
|---|---|---|---|---|---|---|
| + | - | * | ** | / | // | % |
| << | >> | & | \| | ^ | ~ | |
| < | > | <= | >= | == | != | <> |
| += | -= | *= | /= | //= | %= | |
| &= | \|= | ^= | >>= | <<= | **= | |

**TABLE 1.2** Python operators.

## Punctuators and Delimiters

*Punctuators*, also known as delimiters, separate different elements in Python statements and expressions. Some you will recognize from mathematics; others you will recognize from English. We will introduce them throughout the text. The complete list is in Table 1.3.

| | | | | | |
|---|---|---|---|---|---|
| ( | ) | [ | ] | { | } |
| , | : | . | ` | = | ; |
| ' | " | # | \ | @ | |

**TABLE 1.3** Python punctuators.

### Literals

In computer science, a *literal* is a notation for representing a fixed value, which is a value that cannot be changed in the program. Almost all programming languages have notations for atomic values such as integers, floating-point numbers, strings, and Booleans. For example, 123 is a literal; it has a fixed value and cannot be modified. In contrast to literals, variables are symbols that can be assigned a value that can be modified during the execution of the code.

## 1.4.6 Naming Objects

> The practitioner of . . . programming can be regarded as an essayist, whose main concern is with exposition and excellence of style. Such an author, with thesaurus in hand, chooses the names of variables carefully and explains what each variable means. He or she strives for a program that is comprehensible because its concepts have been introduced in an order that is best for human understanding, using a mixture of formal and informal methods that reinforce each other.
>
> Donald Knuth[3]

If writing a program is like writing an essay, then you might guess that the names you use in the program, such as the names of your variables, would help greatly in making the program more readable. Therefore, it is important to choose names well. Later we will provide you with some procedures to choose readable names, but here we can talk about the rules that Python imposes on name selection.

1. Every name must begin with a letter or the underscore character ( _):
   - A numeral is not allowed as the first character.[4]
   - Multiple-word names can be linked together using the underscore character ( _ ), for example, `monty_python`, `holy_grail`. A name *starting* with an underscore is often used by Python and Python programmers to denote a variable with special characteristics. You will see these as we go along, but for now it is best to *not* start variable names with an underscore until you understand what that implies.
2. After the first letter, the name may contain any combination of letters, numbers, and underscores:
   - The name cannot be a *keyword* as listed in Table 1.1.
   - You cannot have any delimiters, punctuation, or operators (as listed in Tables 1.2 and 1.3) in a name.

[3] Excerpt from "Literate Programming" by Donald E. Knuth in *The Computer Journal*, 27(2), pp. 97–111. Published by Oxford University Press, © 1984.

[4] This is so that Python can easily distinguish variable names from numbers.

3. A name can be of any length.
4. UPPERCASE is different from lowercase. For example,
   - `my_name` is different than `my_Name` or `My_Name` or `My_name`.

### 1.4.7 Recommendations on Naming

Because naming is such an important part of programming, conventions are often developed that describe how to create names. Such conventions provide programmers with a common methodology to make clear what the program is doing to anyone who reads it. These conventions describe how to name various elements of a program based on their function, in particular when to use various techniques to name elements (capitalize, uppercase, leading underscore, etc.). It is a bit beyond us at this point to fully discuss these standards as we are not yet familiar with the different kinds of elements in a Python program, but we will describe the rules as we go along. However, for those that are interested, we will be using the standard that Google uses for its Python programmers, the *Google Style Guide for Python* which is similar to Python's PEP 8, the *Style Guide for Python Code*.[5] While we follow those guidelines, it is wise to not follow them to the point where readability is impinged. In that spirit we pull our next rule straight from PEP 8:

> **Rule 4:** A foolish consistency is the hobgoblin of little minds.

That rule is actually a quote from Ralph Waldo Emerson: "A foolish consistency is the hobgoblin of little minds, adored by little statesmen and philosophers and divines. With consistency a great soul has simply nothing to do." Emerson implies that rules are useful to follow, but sometimes not.

## 1.5 VARIABLES

A *variable* is a name you create in your program to represent "something" in your program. That "something" can be one of many types of entities: a value, another program to run, a set of data, a file. For starters, we will talk about a variable as something that represents a value: an integer, a string, a floating-point number, an so on. We create variables with meaningful names because the purpose of a good variable name is to make your code more readable. The variable name *pi* is a name most would recognize, and it is easier to both read and write than 3.1415926536. Once a variable is created, you can store, retrieve, or modify the data associated with that variable. Every time you use your variable in a program, its value is retrieved by Python and used in that variable's place. Thus, a variable is really a way to make it easier to read the program you are writing.

[5] http://google-styleguide.googlecode.com/svn/trunk/pyguide.html

PROGRAMMING TIP

It is often useful to describe your variable using a multiword phrase. The recommended way to do this, according to the *Google Style Guide*, is called "lower_with_under". That is, use lowercase letters to write your variable names, and connect the words together using an underline. Again, it is useful to avoid using a leading underline to your variable names for reasons that will become clear later. It is also useful to avoid capital letters, as the style guide will have something to say about when to use those later. Thus "radius_int" or "circumference_str" would be good variable names.

How does Python associate the value of the variable with its name? In essence, Python makes a list of names (variables, but other names as well) that are being used right now in the Python interpreter. The interpreter maintains a special structure called a *namespace* to keep this list of names and their associated values (see Figure 1.1). Each name in that list is associated with a value, and the Python interpreter updates both names and values during the course of its operation. The name associated with a value is an *alias* for the value, that is, another name for it. Whenever a new variable is created, its name is placed in the list along with an association to a value. If a variable name already exists in the table, its association is updated.

## 1.5.1 Variable Creation and Assignment

How is a name created? In Python, when a name is first used (assigned a value, defined as a function name, etc.) is when the name is created. Python updates the namespace list with the new name and its associated value.

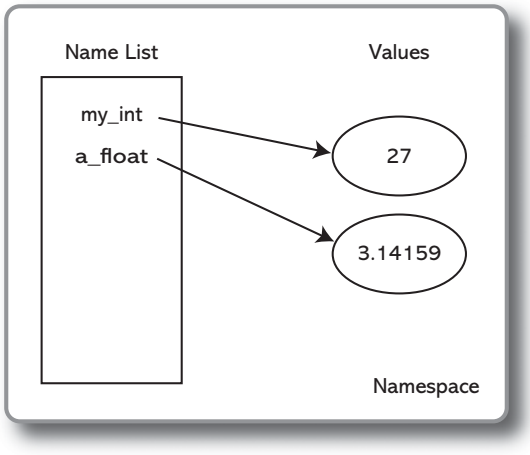



**FIGURE 1.1** Namespace containing variable names and associated values.

Assignment is one way to create a name such as a variable. The purpose of assignment is to associate a name with a value. The symbol of assignment for Python is the equal (=) sign. A simple example follows:

```
my_int = math.pi + 5
```

In this case, 5 is added to the value of the variable associated with `math.pi`, and the result of that expression is associated with the variable `my_int` in the Python namespace. It is important to note that the value `math.pi` is **not** modified by this operation. In fact, assignment does not change any values on the right-hand side. Only a new association is created by assignment: the association with the name on the left-hand side.

The general form of the assignment statement is the same as in mathematics:

left-hand side = right-hand side

Although we use the familiar equal sign (=) from mathematics, its meaning in programming languages is different! In math, the equal (=) indicates equality: what is on the left hand side of the equal sign (=) has the same value as what is on the right-hand side. In Python, the equal sign (=) represents *assignment*. Assignment is the operation to associate a value with a variable. The left-hand side represents the variable name, and the right hand side represents the value. If the variable does not yet exist, then the variable is created and placed in the namespace table; otherwise, the variable's value is updated to the value on the right-hand side. This notation can lead to some odd expressions that would not make much sense mathematically, but make good sense from Python's point of view, such as:

```
my_var = my_var + 1
```

We interpret the previous statement as follows: get the value referred to by `my_var`, add 1 to it, and then associate the resulting sum with the variable `my_var`. That is, if `my_var`'s value was 4, then after assignment its value will be updated to be 5. More generally, the process is to evaluate everything in the expression on the right-hand side first, get the value of that expression, and associate that value with the name on the left-hand side.

Evaluation of an assignment statement is a two-step process:

1. Evaluate the expression on the right-hand side.
2. Take the resulting value from the right-hand expression and associate it with the variable on the left-hand side. (Create the variable if it doesn't exist; otherwise update it.)

For example:

```
my_var = 2 + 3 * 5
```

First evaluate the expression on the right-hand side, resulting in the value 17 (remember, multiplication before addition), then associate 17 with `my_var`; that is, update the namespace so that `my_var` is an alias for 17 in the namespace. Notice how Python uses the standard mathematical rules for the order of operations: multiplication and division before addition or subtraction.

In the earlier assignment statement

```
my_var = my_var + 1
```

notice that `my_var` has two roles. On the right-hand side, it represents a value: "get this value." On the left-hand side, it represents a name that we will associate with value: "a name we will associate with the value."

With that in mind, we can examine some assignment statements that do not make sense and are thus not allowed in Python:

- `7 = my_var + 1` is illegal because 7, like any other integer, is a literal, not a legal variable name, and cannot be changed. (You wouldn't want the value 7 to somehow become 125.)
- `my_var + 7 = 14` is illegal because `my_var + 7` is an expression, not a legal variable name in the namespace.
- Assignment cannot be used in a statement or expression where a value is expected. This is because assignment is a statement; it does not return a value. The statement `print(my_var = 7)` is illegal because print requires a value to print, but the assignment statement does not return a value.

## Check Yourself: Variables and Assignment

**1.** Which of the following are acceptable variable names for Python?
(a) xyzzy
(b) 2ndVar
(c) rich&bill
(d) long_name
(e) good2go

**2.** Which of the following statement(s) best describe a Python namespace?
(a) A list of acceptable names to use with Python
(b) A place where objects are stored in Python
(c) A list of Python names and the values with which they are associated
(d) All of the above
(e) None of the above

**3.** Give the values printed by the following program for each of the labeled lines.

```
int_a = 27
int_b = 5
int_a = 6

print(int_a)      # Line 1
print(int_b + 5)  # Line 2
print(int_b)      # Line 3
```

(a) What is printed by `Line 1`?
(b) What is printed by `Line 2`?
(c) What is printed by `Line 3`?

## 1.6 OBJECTS AND TYPES

In assignment, we associate a value with a variable. What exactly is that value? What information is important and useful to know about that value?

In Python, every "thing" in the system is considered to be an *object*. In Python, though, the word "object" has a very particular meaning. An object in Python has

- an *identity*
- some *attributes*
- zero or more names

Whenever an object is created by Python, it receives an identification number. If you are ever interested in the number of any object, you can use the `id` function to discover its ID number. In general, the ID number isn't very interesting to us, but that number is used by Python to distinguish one object from another. We will take a brief look at the ID here because it helps explain how Python manages objects.

Notice that in addition to the ID number, an object can also have a name or even multiple names. This name is not part of the object's ID but is used by us, the programmers, to make the code more readable. Python uses the namespace to associate a name (such as a variable name) with an object. Interestingly, multiple namespaces may associate different names with the same object!

Finally, every object has a set of attributes associated with it. Attributes are essentially information about the object. We will offer more insight into object attributes later, but the one that we are most interested in right now is an object's type.

In Python, and in many other languages, each object is considered an example of a *type*. For example, 1, 27, and 365 are objects, each of the same type, called `int` (integer). Also, 3.1415926, $6.022141 \times 10^{23}$, $6.67428 \times 10^{-11}$ are all objects that are examples of the type called floating-point numbers (real numbers), which is the type called `float` in Python. Finally, `"spam"`, `'ham'`, `"'fred"'` are objects of the type called strings, called `str` in Python (more in Chapter 4).

Knowing the type of an object informs Python (and us, the programmers) of two things:

- *Attributes* of the object tell us something about its "content". For example, there are no decimal points in an integer object, and there are no letters in either an integer object or a float object.
- *Operations* tell us something about actions that can be done with the object. For example, we can divide two integer objects or two float objects, but the division operation makes no sense on a string object.

If you are unsure of the type of an object in Python, you can ask Python to tell you its type. The function `type` returns the type of any object. The following session shows some interaction with Python and its objects:

```
In  [1]: a_int = 7
In  [2]: id(a_int)
Out [2]: 16790848
```

```
In  [3]: type(a_int) # because 7 is an integer, a_nt is of type int
Out [3]: int

In  [4]: b_float = 2.5
In  [5]: id(b_float) # note that b_float's ID is different than a_int's ID
Out [5]: 17397652

In  [6]: type(b_float) # because 2.5 is Real, b_float is of type float
Out [6]: float

In  [7]: a_int = b_float # associate a_int with value of b_float
In  [8]: type(a_int)
Out [8]: float      # a_int is now of type float

In  [9]: id(a_int) # a_int's now has the same ID as b_float
Out [9]: 17397652
```

Notice a few interesting things in the example session:

- When we ask for the ID (using `id`), we get a system dependent ID number. That number will be different from session to session or machine to machine. To us, the ID number is hard to remember (though not for Python). That's the point! We create variable names so that we can remember an object we have created so we don't *have* to remember weird ID numbers. We show the ID here to help explain Figure 1.2.
- The information returned by a call to the `type` function indicates the type of the value. An integer returns *`int`*, a floating-point object returns *`float`*, and so on.
- Most important, the type of an object has *nothing* to do with the variable name; instead it specifies only the *object* with which it is associated. In the previous Python session, an int object (`a_int = 7`) and a floating-point object (`b_float = 2.5`) associated (respectively) with `a_int` and `b_float`. We then assign the floating-point object to the variable named `a_int` using `a_int = b_float`. The type of object associated with `a_int` is now float. The name of the variable has nothing to do with the type of object with which it is associated. However, it can be useful if the name *does* say something about the associated object for those who must read the code. In any case, Python does not enforce this naming convention. We, as good programmers, must enforce it. Note that the ID of the object associated with `a_int` is now the same as the ID of the variable `b_float`. Both names are now associated with the same object.

Figure 1.2 shows the namespace with two assignments on the left side and a third assignment on the right.

Python (and therefore we) must pay particular attention to an object's type because that type determines what is "reasonable" or "permissible" to do to that object, and if the operation can be done, what the result will be. As you progress in Python, you will learn

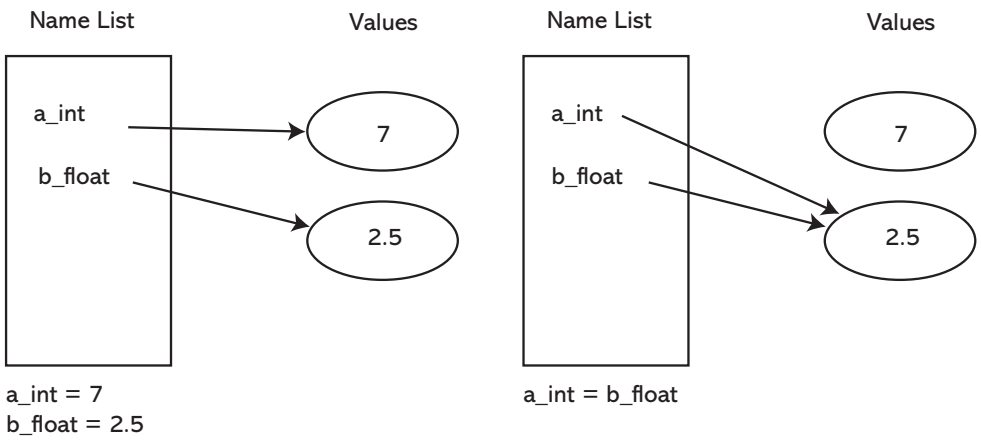



**FIGURE 1.2** Namespace before and after the final assignment.

more about predefined types in Python, as well as how we can define our own types (or, said in a different way, our own *class*. See Chapter 11).

The following sections describe a few of the basic types in Python.

## 1.6.1 Numbers

Python provides several numeric types. We will work a lot with these types during these early chapters because they relate to numeric concepts that we are familiar with. You will add more types as we move through the book.

### Integers

The integer type is designated in Python as type `int`. The integer type corresponds to our mathematical notion of integer. The operations we can perform are those that we would expect: + (addition), − (subtraction), * (multiplication), and / (division, though there are some complications about division /—[see Section 1.7]) as well as a few others. In Python, integers can grow to be as large as needed to store a value, and that value will be exact, but really big integers can be slower to operate on.

How big an integer can you make? Give it a try.

Integers can be written in normal base 10 form or in other base formats—in particular base 8 (called *octal*) and base 16 (called *hexadecimal*). We note this because of an oddity you might run into with integers: leading zeros are not allowed. Python assumes that if you precede a number with a 0 (zero), you mean to encode it in a base other than 10. A letter following the 0 indicates the base. If it is "o", base 8 (octal) is specified. An "x" specifies base 16 (hexadecimal) and a "b" specifies base 2 (binary). We illustrate this in a session:

```
In  [1]: 012 # leading zero without letter is invalid
Out [1]: SyntaxError: invalid token
```

```
In  [2]: 0o12 # "o" indicates octal, base 8
Out [2]: 10

In  [3]: 0x12 # "x" indicates hexadecimal, base 16
Out [3]: 18

In  [4]: 0b101 # "b" indicates binary, base 2
Out [4]: 5
```

## Floating-Point Numbers

Floating-point or real numbers are designated in Python as type `float`. The floating-point type refers to non-integer numbers, numbers with decimal points. Floats are created either by typing the value, such as 25.678 or by using exponential notation, with the exponent represented by an "e". Thus, $2.99 \times 10^8$ is written as `2.99e8` and $9.109 \times 10^{-31}$ can be written as `9.109e-31`. The operators are, like integers, +, −, *, and / (see Section 1.7 for more detail). Floats represent real numbers, but only approximately. For example, what is the exact decimal representation of the operation 2.0/3.0? As you know, there *is* no exact decimal equivalent because the result is an infinite series; that is $2.0/3.0 = 0.666\ldots$. Since a computer has a finite amount of memory, real numbers must be represented with approximations of their actual value. Look at the following session.

```
In  [1]: 2.0 / 3.0
Out [1]: 0.6666666666666666

In  [2]: 1.1 + 2.2
Out [2]: 3.3000000000000003

In  [3]: 0.1 + 0.1 + 0.1 - 0.3
Out [3]: 5.551115123125783e-17
```

If you were to do the calculations yourself, you would find that 1.1 + 2.2 is equal to 3.3. However, the session shows it is 3.3000000000000003. The same applies for the last addition. The result should be 0 but Python returns a *very* small value instead. Approximations like this, if carried through multiple evaluations, can lead to significant differences than what is expected. Python does provide a module called the `decimal` module that provides more predictable, and controllable, results for floating-point numbers.

It is important to remember that floating-point values are approximate values, not exact, and that operations using floating-point values yield approximate values. Integers are exact, and operations on integers yield exact values.

Finally, unlike integers in Python, a leading 0 on a floating-point number carries no significance: 012. (notice the decimal point) is equivalent to 12.0.

## Fractions

Python also provides the type *fraction* for rational numbers. A fraction consists of the obvious two parts: the numerator and the denominator. Fractions do not suffer from the conversion

of a rational to a real number as discussed previously, and can be operated on without loss of precision using addition, subtraction, multiplication, and division. See Section 16.1.1 for more information.

## 1.6.2 Other Built-In Types

Python has more types that we will introduce in the coming chapters. We mention them briefly here as a preview.

### Boolean

A Boolean value has a Python type *bool*. The Boolean type refers to the values `True` or `False` (note the capitalization). If an object is of type Boolean, it can be only one of those two values. In fact, the two Boolean objects are represented as integers: 0 is `False` and 1 is `True`. There are a number of Boolean operators, which we will examine in Chapter 2.

### String

A string in Python has the type *str*. A string is our first *collection* type. A collection type contains multiple objects organized as a single object type. The string type is a *sequence*. It consists of a collection of characters in a sequence (order matters), delimited by single quotes (' ') or double quotes (" "). For example, "This is a string!" or 'here is another string' or even a very short string as "x". Some languages such as C and its derivatives consider single characters and strings as different types, but Python only has strings. Operations on strings are described in Chapter 4.

### List

A list in Python is of type *list*. A list is also a sequence type, like a string, though it can have elements other than characters in the sequence. Because sequences are collections, a list is also a collection. Lists are indicated with square brackets ([ and ]), and its contents are separated by commas. Lists will be covered in Chapter 7. Here is a list:

```
[4, 3.57, 'abc']
```

### Dictionary

A dictionary in Python is of type *dict*. A dictionary is a *map* type, a collection though not a sequence. A map type consists of a set of element pairs. The first element in the pair is the *key* and the second is the *value*. The key can be used to search for a value, much like a dictionary or phone book. If you want to look up the phone number (the value) or a person (the key), you can efficiently search for the name and find the number. Curly braces ({ and }) indicate dictionaries; a colon separates the key and value pair. Dictionaries will be covered in Chapter 9. Here is a dictionary:

```
{'Jones':3471124, 'Larson':3472289, 'Smith':3471288}
```