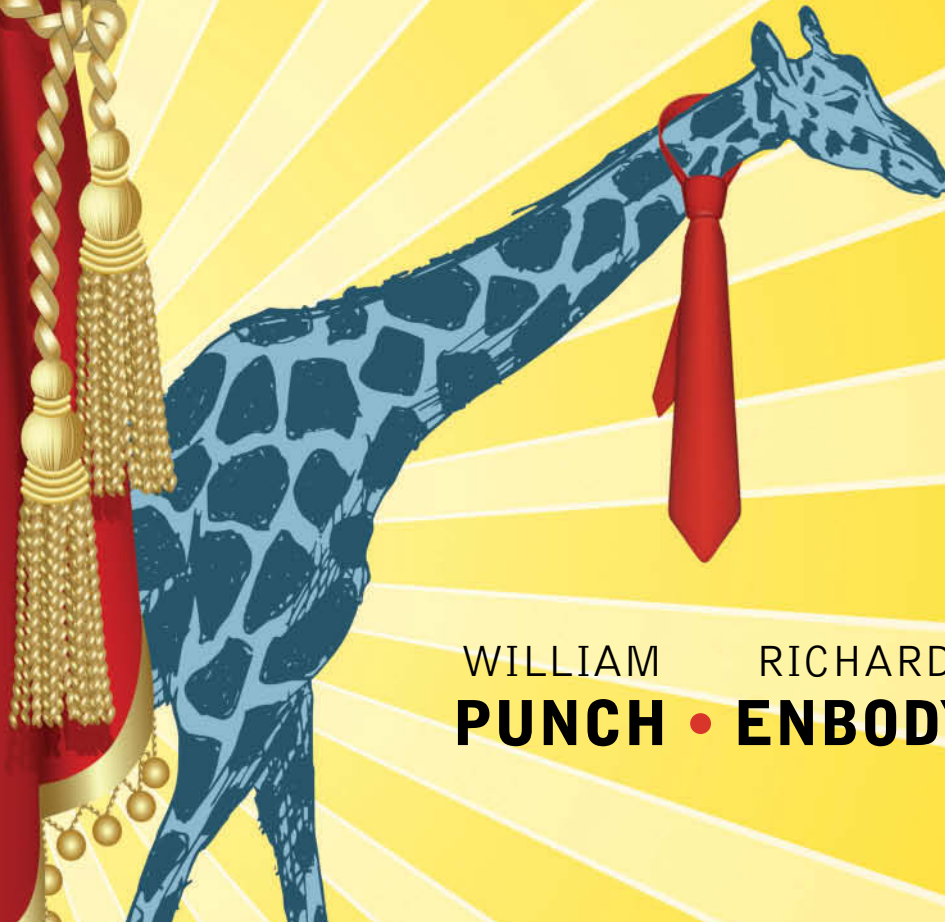




THE PRACTICE OF COMPUTING USING

# PYTHON

3RD EDITION



WILLIAM RICHARD  
**PUNCH • ENBODY**

THIRD  
EDITION

THE PRACTICE OF COMPUTING USING

# PYTHON

William Punch

Richard Enbody

**PEARSON**

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  
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

Senior Specialist, Program Planning and  
Support: Maura Zaldivar-Garcia  
Cover Designer: Joyce Wells  
Manager, Rights and Permissions: Rachel Youdelman  
Project Manager, Rights and Permissions: William Opaluch  
Inventory Manager: Meredith Maresca  
Media Project Manager: Dario Wong  
Full-Service Project Management: Jogender Taneja,  
iEnergizer Aptara®, Ltd.  
Composition: iEnergizer Aptara®, Ltd.  
Printer/Binder: Edwards Brothers Malloy, Inc.  
Cover and Insert Printer: Phoenix Color

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

MICROSOFT AND/OR ITS RESPECTIVE SUPPLIERS MAKE NO REPRESENTATIONS ABOUT THE SUITABILITY OF THE INFORMATION CONTAINED IN THE DOCUMENTS AND RELATED GRAPHICS PUBLISHED AS PART OF THE SERVICES FOR ANY PURPOSE. ALL SUCH DOCUMENTS AND RELATED GRAPHICS ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND. MICROSOFT AND/OR ITS RESPECTIVE SUPPLIERS HEREBY DISCLAIM ALL WARRANTIES AND CONDITIONS WITH REGARD TO THIS INFORMATION, INCLUDING ALL WARRANTIES AND CONDITIONS OF MERCHANTABILITY, WHETHER EXPRESS, IMPLIED OR STATUTORY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT SHALL MICROSOFT AND/OR ITS RESPECTIVE SUPPLIERS BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF INFORMATION AVAILABLE FROM THE SERVICES.

THE DOCUMENTS AND RELATED GRAPHICS CONTAINED HEREIN COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN. MICROSOFT AND/OR ITS RESPECTIVE SUPPLIERS MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED HEREIN AT ANY TIME. PARTIAL SCREEN SHOTS MAY BE VIEWED IN FULL WITHIN THE SOFTWARE VERSION SPECIFIED.

MICROSOFT® WINDOWS®, AND MICROSOFT OFFICE® ARE REGISTERED TRADEMARKS OF THE MICROSOFT CORPORATION IN THE U.S.A AND OTHER COUNTRIES. THIS BOOK IS NOT SPONSORED OR ENDORSED BY OR AFFILIATED WITH THE MICROSOFT CORPORATION.

Cover Photo Credit: Unorobus/Fotolia, Ifong/123RF, Deposit Photos/Glow Images, Onot/Shutterstock, Natalia Natykach/123RF, Vitezslav Valka/123RF

The programs and applications presented in this book have been included for their instructional value. They have been tested with care but are not guaranteed for any particular purpose. The publisher does not offer any warranty or representation, nor does it accept any liabilities with respect to the programs or applications.

---

Copyright © 2017, 2013, 2011 Pearson Education, Inc. All rights reserved. Printed in the United States of America. This publication is protected by Copyright, and permission should be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms and the appropriate contacts within the Pearson Education Global Rights & Permissions department, please visit [www.pearsonhighed.com/permissions/](http://www.pearsonhighed.com/permissions/).

Many of the designations by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed in initial caps or all caps.

**Library of Congress Cataloging-in-Publication Data available upon request.**

Names: Punch, W. F. (William F.), author. | Enbody, Richard J., author.

Title: The practice of computing using Python / W.F. Punch and R.J. Enbody,

Department of Computer Science and Engineering, Michigan State University.

Description: 3rd edition. | Boston : Pearson, 2016. | Includes bibliographical references and index.

Identifiers: LCCN 2015050451 | ISBN 9780134379760 | ISBN 0134379764

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

Classification: LCC QA76.73.P98 P92 2016 | DDC 005/13/3—dc23 LC record available at <http://lccn.loc.gov/2015050451>

10 9 8 7 6 5 4 3 2 1

**PEARSON**

ISBN 10: 0-13-437976-4

ISBN 13: 978-0-13-437976-0

*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** xxiv

**PREFACE** xxv

**PREFACE TO THE SECOND EDITION** xxix

**PART 1 THINKING ABOUT COMPUTING** 1

**Chapter 0** The Study of Computer Science 3

**PART 2 STARTING TO PROGRAM** 35

**Chapter 1** Beginnings 37

**Chapter 2** Control 87

**Chapter 3** Algorithms and Program Development 161

**PART 3 DATA STRUCTURES AND FUNCTIONS** 187

**Chapter 4** Working with Strings 189

**Chapter 5** Functions—QuickStart 245

**Chapter 6** Files and Exceptions I 271

**Chapter 7** Lists and Tuples 311

**Chapter 8** More on Functions 395

**Chapter 9** Dictionaries and Sets 423

**Chapter 10** More Program Development 483

**PART 4 CLASSES, MAKING YOUR OWN DATA STRUCTURES  
AND ALGORITHMS** 527

**Chapter 11** Introduction to Classes 529

**Chapter 12** More on Classes 571

**Chapter 13** Program Development with Classes 615

**PART 5 BEING A BETTER PROGRAMMER 643**

- Chapter 14** Files and Exceptions II 645
- Chapter 15** Recursion: Another Control Mechanism 687
- Chapter 16** Other Fun Stuff with Python 709
- Chapter 17** The End, or Perhaps the Beginning 751

**APPENDICES 753**

- Appendix A** Getting and Using Python 753
- Appendix B** Simple Drawing with Turtle Graphics 773
- Appendix C** What's Wrong with My Code? 785
- Appendix D** Pylab: A Plotting and Numeric Tool 817
- Appendix E** Quick Introduction to Web-based User Interfaces 829
- Appendix F** Table of UTF-8 One Byte Encodings 859
- Appendix G** Precedence 861
- Appendix H** Naming Conventions 863
- Appendix I** Check Yourself Solutions 867

**INDEX 873**

# C O N T E N T S

**VIDEONOTES** xxiv

**PREFACE** xxv

**PREFACE TO THE SECOND EDITION** xxix

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

## **PART 1** THINKING ABOUT COMPUTING 1

### **Chapter 0** The Study of Computer Science 3

- 0.1 Why Computer Science? 3
  - 0.1.1 Importance of Computer Science 3
  - 0.1.2 Computer Science Around You 4
  - 0.1.3 Computer “Science” 4
  - 0.1.4 Computer Science Through Computer Programming 6
- 0.2 The Difficulty and Promise of Programming 6
  - 0.2.1 Difficulty 1: Two Things at Once 6
  - 0.2.2 Difficulty 2: What Is a Good Program? 9
  - 0.2.3 The Promise of a Computer Program 10
- 0.3 Choosing a Computer Language 11
  - 0.3.1 Different Computer Languages 11
  - 0.3.2 Why Python? 11
  - 0.3.3 Is Python the Best Language? 13
- 0.4 What Is Computation? 13
- 0.5 What Is a Computer? 13



0.5.1	Computation in Nature	14
0.5.2	The Human Computer	17
0.6	The Modern, Electronic Computer	18
0.6.1	It's the Switch!	18
0.6.2	The Transistor	19
0.7	A High-Level Look at a Modern Computer	24
0.8	Representing Data	26
0.8.1	Binary Data	26
0.8.2	Working with Binary	27
0.8.3	Limits	28
0.8.4	Representing Letters	29
0.8.5	Representing Other Data	30
0.8.6	What Does a Number Represent?	31
0.8.7	How to Talk About Quantities of Data	32
0.8.8	How Much Data Is That?	32
0.9	Overview of Coming Chapters	34
<b>PART 2</b>	<b>STARTING TO PROGRAM</b>	<b>35</b>
<b>Chapter 1</b>	<b>Beginnings</b>	<b>37</b>
1.1	Practice, Practice, Practice	37
1.2	QUICKSTART, the Circumference Program	38
1.2.1	Examining the Code	40
1.3	An Interactive Session	42
1.4	Parts of a Program	43
1.4.1	Modules	43
1.4.2	Statements and Expressions	43
1.4.3	Whitespace	45
1.4.4	Comments	46
1.4.5	Special Python Elements: Tokens	46
1.4.6	Naming Objects	48
1.4.7	Recommendations on Naming	49
1.5	Variables	49
1.5.1	Variable Creation and Assignment	50
1.6	Objects and Types	53
1.6.1	Numbers	55
1.6.2	Other Built-In Types	57
1.6.3	Object Types: Not Variable Types	58
1.6.4	Constructing New Values	60

1.7	Operators	61
1.7.1	Integer Operators	61
1.7.2	Floating-Point Operators	64
1.7.3	Mixed Operations	64
1.7.4	Order of Operations and Parentheses	65
1.7.5	Augmented Assignment Operators: A Shortcut!	66
1.8	Your First Module, Math	68
1.9	Developing an Algorithm	69
1.9.1	New Rule—Testing	73
1.10	Visual Vignette: Turtle Graphics	74
1.11	What’s Wrong with My Code?	75
<b>Chapter 2</b>	<b>Control</b>	<b>87</b>
2.1	QUICKSTART Control	87
2.1.1	Selection	87
2.1.2	Booleans for Decisions	89
2.1.3	The <i>if</i> Statement	89
2.1.4	Example: What Lead Is Safe in Basketball?	92
2.1.5	Repetition	96
2.1.6	Example: Finding Perfect Numbers	100
2.1.7	Example: Classifying Numbers	105
2.2	In-Depth Control	109
2.2.1	<i>True</i> and <i>False</i> : Booleans	109
2.2.2	Boolean Variables	110
2.2.3	Relational Operators	110
2.2.4	Boolean Operators	115
2.2.5	Precedence	116
2.2.6	Boolean Operators Example	117
2.2.7	Another Word on Assignments	120
2.2.8	The Selection Statement for Decisions	122
2.2.9	More on Python Decision Statements	122
2.2.10	Repetition: the <i>while</i> Statement	126
2.2.11	Sentinel Loop	136
2.2.12	Summary of Repetition	136
2.2.13	More on the <i>for</i> Statement	137
2.2.14	Nesting	140
2.2.15	Hailstone Sequence Example	142
2.3	Visual Vignette: Plotting Data with Pylab	143
2.3.1	First Plot and Using a List	144
2.3.2	More Interesting Plot: A Sine Wave	145

2.4	Computer Science Perspectives: Minimal Universal Computing	147
2.4.1	Minimal Universal Computing	147
2.5	What's Wrong with My Code?	148
<b>Chapter 3</b>	<b>Algorithms and Program Development</b>	<b>161</b>
3.1	What Is an Algorithm?	161
3.1.1	Example Algorithms	162
3.2	Algorithm Features	163
3.2.1	Algorithm versus Program	163
3.2.2	Qualities of an Algorithm	165
3.2.3	Can We Really Do All That?	167
3.3	What Is a Program?	167
3.3.1	Readability	167
3.3.2	Robust	171
3.3.3	Correctness	172
3.4	Strategies for Program Design	173
3.4.1	Engage and Commit	173
3.4.2	Understand, Then Visualize	174
3.4.3	Think Before You Program	175
3.4.4	Experiment	175
3.4.5	Simplify	175
3.4.6	Stop and Think	177
3.4.7	Relax: Give Yourself a Break	177
3.5	A Simple Example	177
3.5.1	Build the Skeleton	178
3.5.2	Output	178
3.5.3	Input	179
3.5.4	Doing the Calculation	181
<b>PART 3</b>	<b>DATA STRUCTURES AND FUNCTIONS</b>	<b>187</b>
<b>Chapter 4</b>	<b>Working with Strings</b>	<b>189</b>
4.1	The String Type	190
4.1.1	The Triple-Quote String	190
4.1.2	Nonprinting Characters	191
4.1.3	String Representation	191
4.1.4	Strings as a Sequence	192
4.1.5	More Indexing and Slicing	193
4.1.6	Strings Are Iterable	198

4.2	String Operations	199
4.2.1	Concatenation (+) and Repetition (*)	199
4.2.2	Determining When + Indicates Addition or Concatenation?	200
4.2.3	Comparison Operators	201
4.2.4	The <i>in</i> Operator	202
4.2.5	String Collections Are Immutable	203
4.3	A Preview of Functions and Methods	205
4.3.1	A String Method	205
4.3.2	Determining Method Names and Method Arguments	208
4.3.3	String Methods	210
4.3.4	String Functions	210
4.4	Formatted Output for Strings	211
4.4.1	Descriptor Codes	212
4.4.2	Width and Alignment Descriptors	213
4.4.3	Floating-Point Precision Descriptor	214
4.5	Control and Strings	215
4.6	Working with Strings	218
4.6.1	Example: Reordering a Person's Name	218
4.6.2	Palindromes	220
4.7	More String Formatting	223
4.8	Unicode	226
4.9	A GUI to Check a Palindrome	228
4.10	What's Wrong with My Code?	232
<b>Chapter 5</b>	<b>Functions—QuickStart</b>	<b>245</b>
5.1	What Is a Function?	245
5.1.1	Why Have Functions?	246
5.2	Python Functions	247
5.3	Flow of Control with Functions	250
5.3.1	Function Flow in Detail	251
5.3.2	Parameter Passing	251
5.3.3	Another Function Example	253
5.3.4	Function Example: Area of a Triangle	254
5.3.5	Functions Calling Functions	258
5.3.6	When to Use a Function	259
5.3.7	What If There Is No Return Statement?	260
5.3.8	What If There Are Multiple Return Statements?	260

	5.4	Visual Vignette: Turtle Flag	261
	5.5	What's Wrong with My Code?	262
<b>Chapter 6</b>		<b>Files and Exceptions I</b>	<b>271</b>
	6.1	What Is a File?	271
	6.2	Accessing Files: Reading Text Files	271
	6.2.1	What's Really Happening?	272
	6.3	Accessing Files: Writing Text Files	273
	6.4	Reading and Writing Text Files in a Program	274
	6.5	File Creation and Overwriting	275
	6.5.1	Files and Functions Example: Word Puzzle	276
	6.6	First Cut, Handling Errors	282
	6.6.1	Error Names	283
	6.6.2	The try-except Construct	283
	6.6.3	try-except Flow of Control	284
	6.6.4	Exception Example	285
	6.7	Example: Counting Poker Hands	288
	6.7.1	Program to Count Poker Hands	291
	6.8	GUI to Count Poker Hands	299
	6.8.1	Count Hands Function	300
	6.8.2	The Rest of the GUI Code	302
	6.9	Error Check Float Input	304
	6.10	What's Wrong with My Code?	304
<b>Chapter 7</b>		<b>Lists and Tuples</b>	<b>311</b>
	7.1	What Is a List?	311
	7.2	What You Already Know How To Do With Lists	313
	7.2.1	Indexing and Slicing	314
	7.2.2	Operators	315
	7.2.3	Functions	317
	7.2.4	List Iteration	318
	7.3	Lists Are Different than Strings	319
	7.3.1	Lists Are Mutable	319
	7.3.2	List Methods	320
	7.4	Old and New Friends: Split and Other Functions and Methods	325
	7.4.1	Split and Multiple Assignment	325
	7.4.2	List to String and Back Again, Using join	326
	7.4.3	The Sorted Function	327

7.5	Working with Some Examples	328
7.5.1	Anagrams	328
7.5.2	Example: File Analysis	334
7.6	Mutable Objects and References	340
7.6.1	Shallow versus Deep Copy	345
7.6.2	Mutable versus Immutable	349
7.7	Tuples	350
7.7.1	Tuples from Lists	352
7.7.2	Why Tuples?	353
7.8	Lists: The Data Structure	353
7.8.1	Example Data Structure	354
7.8.2	Other Example Data Structures	355
7.9	Algorithm Example: U.S. EPA Automobile Mileage Data	355
7.9.1	CSV Module	365
7.10	Visual Vignette: Plotting EPA Data	366
7.11	List Comprehension	368
7.11.1	Comprehensions, Expressions, and the Ternary Operator	370
7.12	Visual Vignette: More Plotting	370
7.12.1	Pylab Arrays	371
7.12.2	Plotting Trigonometric Functions	373
7.13	GUI to Find Anagrams	374
7.13.1	Function Model	374
7.13.2	Controller	375
7.14	What's Wrong with My Code?	377
<b>Chapter 8</b>	<b>More on Functions</b>	<b>395</b>
8.1	Scope	395
8.1.1	Arguments, Parameters, and Namespaces	397
8.1.2	Passing Mutable Objects	399
8.1.3	Returning a Complex Object	401
8.1.4	Refactoring evens	403
8.2	Default Values and Parameters as Keywords	404
8.2.1	Example: Default Values and Parameter Keywords	405
8.3	Functions as Objects	407
8.3.1	Function Annotations	408
8.3.2	Docstrings	409

8.4	Example: Determining a Final Grade	410
8.4.1	The Data	410
8.4.2	The Design	410
8.4.3	Function: <i>weighted_grade</i>	411
8.4.4	Function: <i>parse_line</i>	411
8.4.5	Function: <i>main</i>	412
8.4.6	Example Use	413
8.5	Pass “by Value” or “by Reference”	413
8.6	What’s Wrong with My Code?	414
<b>Chapter 9</b>	<b>Dictionaries and Sets</b>	<b>423</b>
9.1	Dictionaries	423
9.1.1	Dictionary Example	424
9.1.2	Python Dictionaries	425
9.1.3	Dictionary Indexing and Assignment	425
9.1.4	Operators	426
9.1.5	Ordered Dictionaries	431
9.2	Word Count Example	432
9.2.1	Count Words in a String	432
9.2.2	Word Frequency for Gettysburg Address	433
9.2.3	Output and Comments	437
9.3	Periodic Table Example	438
9.3.1	Working with CSV Files	439
9.3.2	Algorithm Overview	441
9.3.3	Functions for Divide and Conquer	441
9.4	Sets	445
9.4.1	History	445
9.4.2	What’s in a Set?	445
9.4.3	Python Sets	446
9.4.4	Methods, Operators, and Functions for Python Sets	447
9.4.5	Set Methods	447
9.5	Set Applications	452
9.5.1	Relationship between Words of Different	452
9.5.2	Output and Comments	456
9.6	Scope: The Full Story	456
9.6.1	Namespaces and Scope	457
9.6.2	Search Rule for Scope	457
9.6.3	Local	457
9.6.4	Global	458
9.6.5	Built-Ins	462
9.6.6	Enclosed	463

9.7	Using zip to Create Dictionaries	464
9.8	Dictionary and Set Comprehensions	465
9.9	Visual Vignette: Bar Graph of Word Frequency	466
9.9.1	Getting the Data Right	466
9.9.2	Labels and the xticks Command	467
9.9.3	Plotting	467
9.10	GUI to Compare Files	468
9.10.1	Controller and View	469
9.10.2	Function Model	471
9.11	What's Wrong with My Code?	473
<b>Chapter 10</b>	<b>More Program Development</b>	<b>483</b>
10.1	Introduction	483
10.2	Divide and Conquer	483
10.2.1	Top-Down Refinement	484
10.3	The Breast Cancer Classifier	484
10.3.1	The Problem	484
10.3.2	The Approach: Classification	485
10.3.3	Training and Testing the Classifier	485
10.3.4	Building the Classifier	485
10.4	Designing the Classifier Algorithm	487
10.4.1	Divided, now Conquer	490
10.4.2	Data Structures	491
10.4.3	File Format	491
10.4.4	The make_training_set Function	492
10.4.5	The make_test_set Function	496
10.4.6	The train_classifier Function	497
10.4.7	train_classifier, Round 2	499
10.4.8	Testing the Classifier on New Data	502
10.4.9	The report_results Function	506
10.5	Running the Classifier on Full Data	508
10.5.1	Training versus Testing	508
10.6	Other Interesting Problems	512
10.6.1	Tag Clouds	512
10.6.2	S&P 500 Predictions	514
10.6.3	Predicting Religion with Flags	517
10.7	GUI to Plot the Stock Market	519
10.7.1	Function Model	519
10.7.2	Controller and View	521



## PART 4 CLASSES, MAKING YOUR OWN DATA STRUCTURES AND ALGORITHMS 527

### Chapter 11 Introduction to Classes 529

- 11.1 QUICKSTART: Simple Student Class 529
- 11.2 Object-Oriented Programming 530
  - 11.2.1 Python Is Object-Oriented! 530
  - 11.2.2 Characteristics of OOP 531
- 11.3 Working with OOP 531
  - 11.3.1 Class and Instance 531
- 11.4 Working with Classes and Instances 532
  - 11.4.1 Built-In Class and Instance 532
  - 11.4.2 Our First Class 534
  - 11.4.3 Changing Attributes 536
  - 11.4.4 The Special Relationship Between an Instance and Class: instance-of 537
- 11.5 Object Methods 540
  - 11.5.1 Using Object Methods 540
  - 11.5.2 Writing Methods 541
  - 11.5.3 The Special Argument `self` 542
  - 11.5.4 Methods Are the Interface to a Class Instance 544
- 11.6 Fitting into the Python Class Model 545
  - 11.6.1 Making Programmer-Defined Classes 545
  - 11.6.2 A Student Class 545
  - 11.6.3 Python Standard Methods 546
  - 11.6.4 Now There Are Three: Class Designer, Programmer, and User 550
- 11.7 Example: Point Class 551
  - 11.7.1 Construction 553
  - 11.7.2 Distance 553
  - 11.7.3 Summing Two Points 553
  - 11.7.4 Improving the Point Class 554
- 11.8 Python and OOP 558
  - 11.8.1 Encapsulation 558
  - 11.8.2 Inheritance 559
  - 11.8.3 Polymorphism 559
- 11.9 Python and Other OOP Languages 559
  - 11.9.1 Public versus Private 559
  - 11.9.2 Indicating Privacy Using Double Underscores (`__`) 560

	11.9.3	Python's Philosophy	561
	11.9.4	Modifying an Instance	562
	11.10	What's Wrong with My Code?	562
<b>Chapter 12</b>		<b>More on Classes</b>	<b>571</b>
	12.1	More About Class Properties	571
	12.1.1	Rational Number (Fraction) Class Example	572
	12.2	How Does Python Know?	574
	12.2.1	Classes, Types, and Introspection	574
	12.2.2	Remember Operator Overloading	577
	12.3	Creating Your Own Operator Overloading	577
	12.3.1	Mapping Operators to Special Methods	578
	12.4	Building the Rational Number Class	581
	12.4.1	Making the Class	581
	12.4.2	Review Fraction Addition	583
	12.4.3	Back to Adding Fractions	586
	12.4.4	Equality and Reducing Rationals	590
	12.4.5	Divide and Conquer at Work	593
	12.5	What Doesn't Work (Yet)	593
	12.5.1	Introspection	594
	12.5.2	Repairing <code>int + Rational</code> Errors	596
	12.6	Inheritance	598
	12.6.1	The "Find the Attribute" Game	599
	12.6.2	Using Inheritance	602
	12.6.3	Example: The Standard Model	603
	12.7	What's Wrong with My Code?	608
<b>Chapter 13</b>		<b>Program Development with Classes</b>	<b>615</b>
	13.1	Predator–Prey Problem	615
	13.1.1	The Rules	616
	13.1.2	Simulation Using Object-Oriented Programming	617
	13.2	Classes	617
	13.2.1	<i>Island</i> Class	617
	13.2.2	Predator and Prey, Kinds of Animals	619
	13.2.3	Predator and Prey Classes	622
	13.2.4	Object Diagram	623
	13.2.5	Filling the Island	623
	13.3	Adding Behavior	626
	13.3.1	Refinement: Add Movement	626
	13.3.2	Refinement: Time Simulation Loop	629

- 13.4 Refinement: Eating, Breeding, and Keeping Time 630
  - 13.4.1 Improved Time Loop 631
  - 13.4.2 Breeding 634
  - 13.4.3 Eating 636
  - 13.4.4 The Tick of the Clock 637
- 13.5 Refinement: How Many Times to Move? 638
- 13.6 Visual Vignette: Graphing Population Size 639

## PART 5 BEING A BETTER PROGRAMMER 643

### Chapter 14 Files and Exceptions II 645

- 14.1 More Details on Files 645
  - 14.1.1 Other File Access Methods, Reading 647
  - 14.1.2 Other File Access Methods, Writing 649
  - 14.1.3 Universal New Line Format 651
  - 14.1.4 Moving Around in a File 652
  - 14.1.5 Closing a File 654
  - 14.1.6 The *with* Statement 654
  - 14.1.7 Text File Encodings; Unicode 655
- 14.2 CSV Files 656
  - 14.2.1 CSV Module 657
  - 14.2.2 CSV Reader 658
  - 14.2.3 CSV Writer 659
  - 14.2.4 Example: Update Some Grades 659
- 14.3 Module: *os* 661
  - 14.3.1 Directory (Folder) Structure 662
  - 14.3.2 *os* Module Functions 663
  - 14.3.3 *os* Module Example 665
- 14.4 More on Exceptions 667
  - 14.4.1 Basic Exception Handling 668
  - 14.4.2 A Simple Example 669
  - 14.4.3 Events 671
  - 14.4.4 A Philosophy Concerning Exceptions 672
- 14.5 Exception: *else* and *finally* 673
  - 14.5.1 *finally* and *with* 673
  - 14.5.2 Example: Refactoring the Reprompting of a File Name 673
- 14.6 More on Exceptions 675
  - 14.6.1 *Raise* 675
  - 14.6.2 Create Your Own 676
- 14.7 Example: Password Manager 677

<b>Chapter 15</b>	<b>Recursion: Another Control Mechanism</b>	<b>687</b>
15.1	What Is Recursion?	687
15.2	Mathematics and Rabbits	689
15.3	Let's Write Our Own: Reversing a String	692
15.4	How Does Recursion Actually Work?	694
15.4.1	Stack Data Structure	695
15.4.2	Stacks and Function Calls	697
15.4.3	A Better Fibonacci	699
15.5	Recursion in Figures	700
15.5.1	Recursive Tree	700
15.5.2	Sierpinski Triangles	702
15.6	Recursion to Non-recursion	703
15.7	GUI for Turtle Drawing	704
15.7.1	Using Turtle Graphics to Draw	704
15.7.2	Function Model	705
15.7.3	Controller and View	706
<b>Chapter 16</b>	<b>Other Fun Stuff with Python</b>	<b>709</b>
16.1	Numbers	709
16.1.1	Fractions	710
16.1.2	Decimal	714
16.1.3	Complex Numbers	718
16.1.4	Statistics Module	720
16.1.5	Random Numbers	722
16.2	Even More on Functions	724
16.2.1	Having a Varying Number of Parameters	725
16.2.2	Iterators and Generators	728
16.2.3	Other Functional Programming Ideas	733
16.2.4	Some Functional Programming Tools	734
16.2.5	Decorators: Functions Calling Functions	736
16.3	Classes	741
16.3.1	Properties	742
16.3.2	Serializing an Instance: pickle	745
16.4	Other Things in Python	748
16.4.1	Data Types	748
16.4.2	Built-in Modules	748
16.4.3	Modules on the Internet	749
<b>Chapter 17</b>	<b>The End, or Perhaps the Beginning</b>	<b>751</b>

## APPENDICES 753

- Appendix A** Getting and Using Python 753
  - A.1** About Python 753
    - A.1.1** History 753
    - A.1.2** Python 3 753
    - A.1.3** Python Is Free and Portable 754
    - A.1.4** Installing Anaconda 756
    - A.1.5** Starting Our Python IDE: SPYDER 756
    - A.1.6** Working with Python 757
    - A.1.7** Making a Program 760
  - A.2** The IPython Console 762
    - A.2.1** Anatomy of an iPython Session 763
    - A.2.2** Your Top Three iPython Tips 764
    - A.2.3** Completion and the Tab Key 764
    - A.2.4** The ? Character 766
    - A.2.5** More iPython Tips 766
  - A.3** Some Conventions for This Book 769
    - A.3.1** Interactive Code 770
    - A.3.2** Program: Written Code 770
    - A.3.3** Combined Program and Output 770
  - A.4** Summary 771
- Appendix B** Simple Drawing with Turtle Graphics 773
  - B.0.1** What Is a Turtle? 773
  - B.0.2** Motion 775
  - B.0.3** Drawing 775
  - B.0.4** Color 777
  - B.0.5** Drawing with Color 779
  - B.0.6** Other Commands 781
  - B.1** Tidbits 783
    - B.1.1** Reset/Close the Turtle Window 783
- Appendix C** What's Wrong with My Code? 785
  - C.1** It's Your Fault! 785
    - C.1.1** Kinds of Errors 785
    - C.1.2** "Bugs" and Debugging 787
  - C.2** Debugging 789
    - C.2.1** Testing for Correctness 789
    - C.2.2** Probes 789
    - C.2.3** Debugging with SPYDER Example 1 789
    - C.2.4** Debugging Example 1 Using `print()` 793

	<b>C.2.5</b>	Debugging with SPYDER Example 2	<b>794</b>
	<b>C.2.6</b>	More Debugging Tips	<b>802</b>
<b>C.3</b>		More about Testing	<b>803</b>
	<b>C.3.1</b>	Testing Is Hard!	<b>804</b>
	<b>C.3.2</b>	Importance of Testing	<b>805</b>
	<b>C.3.3</b>	Other Kinds of Testing	<b>805</b>
<b>C.4</b>		What's Wrong with My Code?	<b>805</b>
	<b>C.4.1</b>	Chapter 1: Beginnings	<b>805</b>
	<b>C.4.2</b>	Chapter 2: Control	<b>807</b>
	<b>C.4.3</b>	Chapter 4: Strings	<b>808</b>
	<b>C.4.4</b>	Chapter 5: Functions	<b>809</b>
	<b>C.4.5</b>	Chapter 6: Files and Exceptions	<b>810</b>
	<b>C.4.6</b>	Chapter 7: Lists and Tuples	<b>811</b>
	<b>C.4.7</b>	Chapter 8: More Functions	<b>812</b>
	<b>C.4.8</b>	Chapter 9: Dictionaries	<b>813</b>
	<b>C.4.9</b>	Chapter 11: Classes I	<b>814</b>
	<b>C.4.10</b>	Chapter 12: Classes II	<b>815</b>
<b>Appendix D</b>		Pylab: A Plotting and Numeric Tool	<b>817</b>
	<b>D.1</b>	Plotting	<b>817</b>
	<b>D.2</b>	Working with pylab	<b>818</b>
	<b>D.2.1</b>	Plot Command	<b>818</b>
	<b>D.2.2</b>	Colors, Marks, and Lines	<b>819</b>
	<b>D.2.3</b>	Generating X-Values	<b>819</b>
	<b>D.2.4</b>	Plot Properties	<b>820</b>
	<b>D.2.5</b>	Tick Labels	<b>821</b>
	<b>D.2.6</b>	Legend	<b>822</b>
	<b>D.2.7</b>	Bar Graphs	<b>824</b>
	<b>D.2.8</b>	Histograms	<b>824</b>
	<b>D.2.9</b>	Pie Charts	<b>825</b>
	<b>D.2.10</b>	How Powerful Is pylab?	<b>826</b>
<b>Appendix E</b>		Quick Introduction to Web-based User Interfaces	<b>829</b>
	<b>E.0.1</b>	MVC Architecture	<b>830</b>
<b>E.1</b>		Flask	<b>830</b>
<b>E.2</b>		QuickStart Flask, Hello World	<b>831</b>
	<b>E.2.1</b>	What Just Happened?	<b>832</b>
	<b>E.2.2</b>	Multiple Routes	<b>833</b>
	<b>E.2.3</b>	Stacked Routes, Passing Address Arguments	<b>835</b>
<b>E.3</b>		Serving Up Real HTML Pages	<b>836</b>
	<b>E.3.1</b>	A Little Bit of HTML	<b>836</b>
	<b>E.3.2</b>	HTML Tags	<b>836</b>

	<b>E.3.3</b>	Flask Returning Web Pages	<b>838</b>
	<b>E.3.4</b>	Getting Arguments into Our Web Pages	<b>839</b>
<b>E.4</b>		Active Web Pages	<b>841</b>
	<b>E.4.1</b>	Forms in <code>wtforms</code>	<b>841</b>
	<b>E.4.2</b>	A Good Example Goes a Long Way	<b>842</b>
	<b>E.4.3</b>	Many Fields Example	<b>847</b>
<b>E.5</b>		Displaying and Updating Images	<b>852</b>
<b>E.6</b>		Odds and Ends	<b>857</b>
<b>Appendix F</b>		Table of UTF-8 One Byte Encodings	<b>859</b>
<b>Appendix G</b>		Precedence	<b>861</b>
<b>Appendix H</b>		Naming Conventions	<b>863</b>
	<b>H.1</b>	Python Style Elements	<b>864</b>
	<b>H.2</b>	Naming Conventions	<b>864</b>
	<b>H.2.1</b>	Our Added Naming Conventions	<b>864</b>
	<b>H.3</b>	Other Python Conventions	<b>865</b>
<b>Appendix I</b>		Check Yourself Solutions	<b>867</b>
	<b>I.1</b>	Chapter 1	<b>867</b>
		Variables and Assignment	<b>867</b>
		Types and Operators	<b>867</b>
	<b>I.2</b>	Chapter 2	<b>868</b>
		Basic Control Check	<b>868</b>
		Loop Control Check	<b>868</b>
		More Control Check	<b>868</b>
		<i>for</i> and <i>range</i> Check	<b>868</b>
	<b>I.3</b>	Chapter 4	<b>869</b>
		Slicing Check	<b>869</b>
		String Comparison Check	<b>869</b>
	<b>I.4</b>	Chapter 5	<b>869</b>
		Simple Functions Check	<b>869</b>
	<b>I.5</b>	Chapter 6	<b>869</b>
		Exception Check	<b>869</b>
		Function Practice with Strings	<b>870</b>
	<b>I.6</b>	Chapter 7	<b>870</b>
		Basic Lists Check	<b>870</b>
		Lists and Strings Check	<b>870</b>
		Mutable List Check	<b>870</b>

- I.7** Chapter 8 **870**
  - Passing Mutables Check **870**
  - More on Functions Check **871**
- I.8** Chapter 9 **871**
  - Dictionary Check **871**
  - Set Check **871**
- I.9** Chapter 11 **871**
  - Basic Classes Check **871**
  - Defining Special Methods **871**
- I.10** Chapter 12 **872**
  - Check Defining Your Own Operators **872**
- I.11** Chapter 14 **872**
  - Basic File Operations **872**
  - Basic Exception Control **872**

**INDEX 873**



# VIDEONOTES

<b>VideoNote 0.1</b>	Getting Python	13
<b>VideoNote 1.1</b>	Simple Arithmetic	64
<b>VideoNote 1.2</b>	Solving Your First Problem	73
<b>VideoNote 2.1</b>	Simple Control	96
<b>VideoNote 2.2</b>	Nested Control	140
<b>VideoNote 3.1</b>	Algorithm Decomposition	177
<b>VideoNote 3.2</b>	Algorithm Development	185
<b>VideoNote 4.1</b>	Playing with Strings	210
<b>VideoNote 4.2</b>	String Formatting	214
<b>VideoNote 5.1</b>	Simple Functions	251
<b>VideoNote 5.2</b>	Problem Design Using Functions	261
<b>VideoNote 6.1</b>	Reading Files	272
<b>VideoNote 6.2</b>	Simple Exception Handling	285
<b>VideoNote 7.1</b>	List Operations	327
<b>VideoNote 7.2</b>	List Application	349
<b>VideoNote 8.1</b>	More on Parameters	405
<b>VideoNote 9.1</b>	Using a Dictionary	437
<b>VideoNote 9.2</b>	More Dictionaries	465
<b>VideoNote 10.1</b>	Program Development: Tag Cloud	512
<b>VideoNote 11.1</b>	Designing a Class	545
<b>VideoNote 11.2</b>	Improving a Class	554
<b>VideoNote 12.1</b>	Augmenting a Class	593
<b>VideoNote 12.2</b>	Create a Class	596
<b>VideoNote 13.1</b>	Improve Simulation	623
<b>VideoNote 14.1</b>	Dictionary Exceptions	669
<b>VideoNote 15.1</b>	Recursion	692
<b>VideoNote 16.1</b>	Properties	742



## P R E F A C E

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 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.



# 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 2 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.pearsonhighered.com/cs-resources/](http://www.pearsonhighered.com/cs-resources/)

- 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.pearsonhighered.com/irc](http://www.pearsonhighered.com/irc). Register at the site for access. You may also contact your local Pearson Education sales representative

W. F. PUNCH  
R. J. ENBODY



# 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**.

Copyright © 2017 Pearson Education, Inc. or its affiliate(s). All rights reserved. HELO88173 • 11/15



PART

1

# Thinking About Computing

**Chapter 0** The Study of Computer Science

This page intentionally left blank



# 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,”<sup>1</sup> 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<sup>(TM)</sup> 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?

---

<sup>1</sup> <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 Damoiselle 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  
The good day (i.e., a  
hello, i.e., greetings).  
The stay/sojourn/  
visit (i.e., quarantine)  
{It} is prison.

### A une Damoiselle malade

Ma mignonne,  
Je vous donne  
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

My sweet dear,  
I send cheer –  
All the best!  
Your forced rest  
Is like jail.

### A une Damoiselle malade

Ma mignonne,  
Je vous donne  
Le bon jour;  
Le séjour  
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.