

Intro to Python®

for Computer Science and Data Science



Intro to Python® for Computer Science and Data Science

Learning to Program with AI, Big Data and the Cloud

by Paul Deitel & Harvey Deitel

PART 1

CS: Python Fundamentals Quickstart

CS 1. Introduction to Computers and Python

DS Intro: AI—at the Intersection of CS and DS

CS 2. Introduction to Python Programming

DS Intro: Basic Descriptive Stats

CS 3. Control Statements and Program Development

DS Intro: Measures of Central Tendency—Mean, Median, Mode

CS 4. Functions

DS Intro: Basic Statistics—Measures of Dispersion

CS 5. Lists and Tuples

DS Intro: Simulation and Static Visualization

1. Chapters 1–11 marked CS are traditional Python programming and computer-science topics.

2. Light-tinted bottom boxes in Chapters 1–10 marked DS Intro are brief, friendly introductions to data-science topics.

PART 2

CS: Python Data Structures, Strings and Files

CS 6. Dictionaries and Sets

DS Intro: Simulation and Dynamic Visualization

CS 7. Array-Oriented Programming with NumPy

High-Performance NumPy Arrays

DS Intro:
Pandas Series and DataFrames

CS 8. Strings: A Deeper Look

Includes Regular Expressions

DS Intro: Pandas, Regular Expressions and Data Wrangling

CS 9. Files and Exceptions

DS Intro: Loading Datasets from CSV Files into Pandas DataFrames

3. Chapters 12–17 marked DS are Python-based, AI, big data and cloud chapters, each containing several full-implementation studies.

4. Functional-style programming is integrated book wide.

PART 3

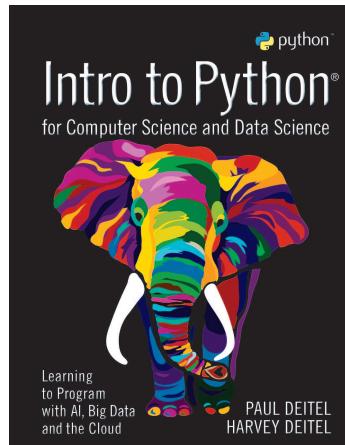
CS: Python High-End Topics

CS 10. Object-Oriented Programming

DS Intro: Time Series and Simple Linear Regression

CS 11. Computer Science Thinking: Recursion, Searching, Sorting and Big O

CS and DS Other Topics Blog



5. Preface explains the dependencies among the chapters.
6. Visualizations throughout.

PART 4

AI, Big Data and Cloud Case Studies

DS 12. Natural Language Processing (NLP)

Web Scraping in the Exercises

DS 13. Data Mining Twitter®

Sentiment Analysis, JSON and Web Services

DS 14. IBM Watson® and Cognitive Computing

DS 15. Machine Learning: Classification, Regression and Clustering

DS 16. Deep Learning

Convolutional and Recurrent Neural Networks; Reinforcement Learning in the Exercises

DS 17. Big Data: Hadoop®, Spark™, NoSQL and IoT

7. CS courses may cover more of the Python chapters and less of the DS content. Vice versa for Data Science courses.
8. We put Chapter 5 in Part 1. It's also a natural fit with Part 2.
Questions? deitel@deitel.com

Deitel® Series Page

How To Program Series

Android™ How to Program, 3/E
C++ How to Program, 10/E
C How to Program, 8/E
Java™ How to Program, Early Objects Version, 11/E
Java™ How to Program, Late Objects Version, 11/E
Internet & World Wide Web How to Program, 5/E
Visual Basic® 2012 How to Program, 6/E
Visual C#® How to Program, 6/E

REVEL™ Interactive Multimedia

REVEL™ for Deitel Java™

VitalSource Web Books

<http://bit.ly/DeitelOnVitalSource>
Android™ How to Program, 2/E and 3/E
C++ How to Program, 9/E and 10/E
Java™ How to Program, 10/E and 11/E
Simply C++: An App-Driven Tutorial Approach
Simply Visual Basic® 2010: An App-Driven Approach, 4/E
Visual Basic® 2012 How to Program, 6/E
Visual C#® How to Program, 6/E
Visual C#® 2012 How to Program, 5/E

Deitel® Developer Series

Android™ 6 for Programmers: An App-Driven Approach, 3/E
C for Programmers with an Introduction to C11
C++11 for Programmers
C# 6 for Programmers
Java™ for Programmers, 4/E
JavaScript for Programmers
Swift™ for Programmers

LiveLessons Video Training

<http://deitel.com/books/LiveLessons/>
Android™ 6 App Development Fundamentals, 3/E
C++ Fundamentals
Java SE 8™ Fundamentals, 2/E
Java SE 9™ Fundamentals, 3/E
C# 6 Fundamentals
C# 2012 Fundamentals
JavaScript Fundamentals
Swift™ Fundamentals

To receive updates on Deitel publications, Resource Centers, training courses, partner offers and more, please join the Deitel communities on

- Facebook®—<http://facebook.com/DeitelFan>
- Twitter®—@deitel
- LinkedIn®—<http://linkedin.com/company/deitel-&-associates>
- YouTube™—<http://youtube.com/DeitelTV>
- Instagram®—<http://instagram.com/DeitelFan>

and register for the free *Deitel® Buzz Online* e-mail newsletter at:

<http://www.deitel.com/newsletter/subscribe.html>

To communicate with the authors, send e-mail to:

deitel@deitel.com

For information on programming-languages corporate training seminars offered by Deitel & Associates, Inc. worldwide, write to deitel@deitel.com or visit:

<http://www.deitel.com/training/>

For continuing updates on Pearson/Deitel publications visit:

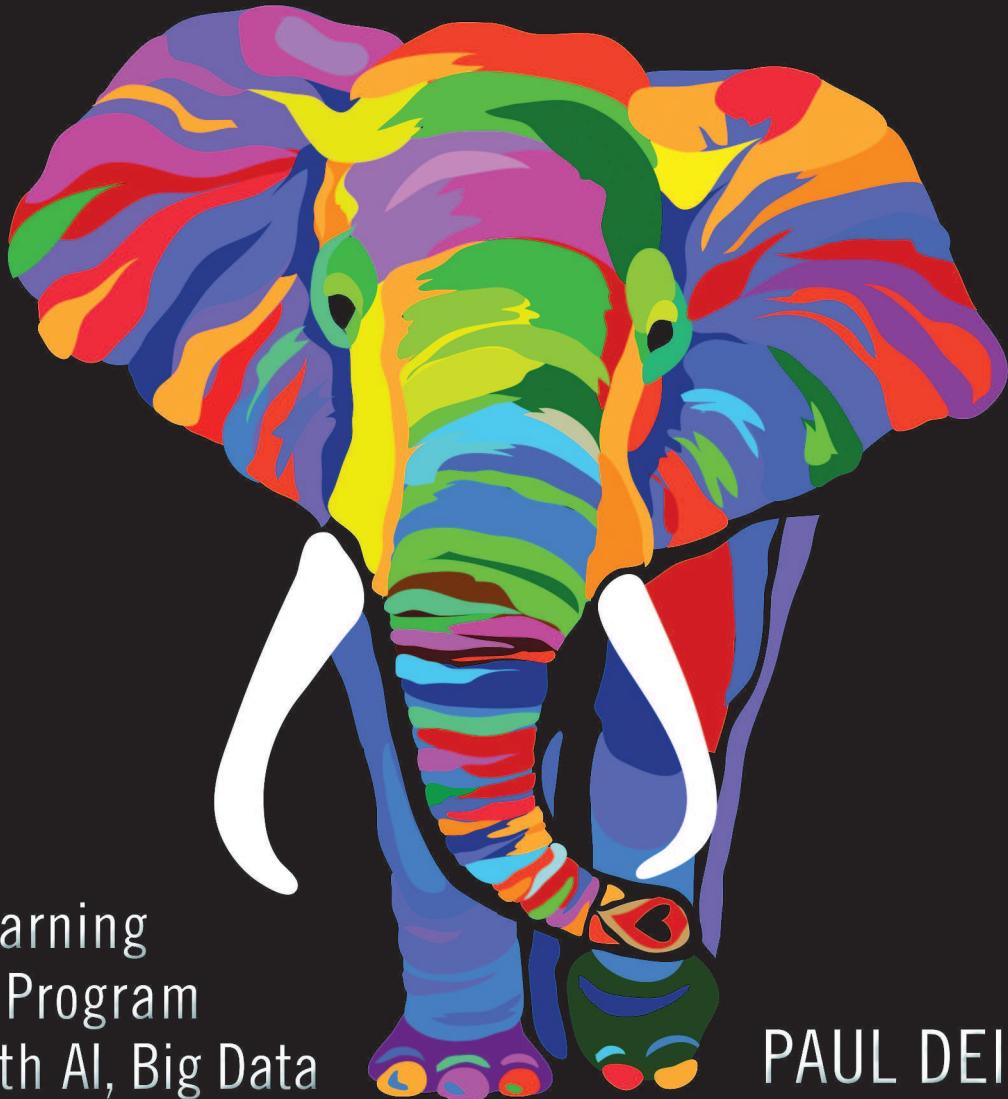
<http://www.deitel.com>

<http://www.pearson.com/deitel>



Intro to Python®

for Computer Science and Data Science



Learning
to Program
with AI, Big Data
and the Cloud

PAUL DEITEL
HARVEY DEITEL

Senior Vice President, Courseware Portfolio Management: *Marcia J. Horton*
Director, Portfolio Management: Engineering, Computer Science & Global Editions: *Julian Partridge*
Executive Higher Ed Portfolio Management: *Tracy Johnson (Dunkelberger)*
Portfolio Management Assistant: *Meghan Jacoby*
Managing Content Producer: *Scott Disanno*
Content Producer: *Carole Snyder*
Rights and Permissions Manager: *Ben Ferrini*
Inventory Manager: *Bruce Boundy*
Product Marketing Manager: *Yvonne Vannatta*
Field Marketing Manager: *Demetrius Hall*
Marketing Assistant: *Jon Bryant*
Cover Designer: *Paul Deitel, Harvey Deitel, Chuti Prasertsith*
Cover Art: ©Denel/Shutterstock

Copyright ©2020 Pearson Education, Inc. All rights reserved. Manufactured 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 <http://www.pearsoned.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.

The authors and publisher of this book have used their best efforts in preparing this book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The authors and publisher make no warranty of any kind, expressed or implied, with regard to these programs or to the documentation contained in this book. The authors and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

Deitel and the double-thumbs-up bug are registered trademarks of Deitel and Associates, Inc.

Library of Congress Cataloging-in-Publication Data
On file



ISBN-10: 0-13-540467-3
ISBN-13: 978-0-13-540467-6

*In Memory of Marvin Minsky,
a founding father of
artificial intelligence*

*It was a privilege to be your student in two
artificial-intelligence graduate courses at M.I.T.
You inspired your students to think beyond limits.*

Harvey Deitel



Contents

Preface	xix
----------------	------------

Before You Begin	xlv
-------------------------	------------

1 Introduction to Computers and Python	1
1.1 Introduction	2
1.2 Hardware and Software	3
1.2.1 Moore's Law	4
1.2.2 Computer Organization	4
1.3 Data Hierarchy	6
1.4 Machine Languages, Assembly Languages and High-Level Languages	9
1.5 Introduction to Object Technology	10
1.6 Operating Systems	13
1.7 Python	16
1.8 It's the Libraries!	18
1.8.1 Python Standard Library	18
1.8.2 Data-Science Libraries	18
1.9 Other Popular Programming Languages	20
1.10 Test-Drive: Using IPython and Jupyter Notebooks	21
1.10.1 Using IPython Interactive Mode as a Calculator	21
1.10.2 Executing a Python Program Using the IPython Interpreter	23
1.10.3 Writing and Executing Code in a Jupyter Notebook	24
1.11 Internet and World Wide Web	29
1.11.1 Internet: A Network of Networks	29
1.11.2 World Wide Web: Making the Internet User-Friendly	30
1.11.3 The Cloud	30
1.11.4 Internet of Things	31
1.12 Software Technologies	32
1.13 How Big Is Big Data?	33
1.13.1 Big Data Analytics	38
1.13.2 Data Science and Big Data Are Making a Difference: Use Cases	39
1.14 Intro to Data Science: Case Study—A Big-Data Mobile Application	40
2 Introduction to Python Programming	49
2.1 Introduction	50
2.2 Variables and Assignment Statements	50

2.3	Arithmetic	52
2.4	Function <code>print</code> and an Intro to Single- and Double-Quoted Strings	56
2.5	Triple-Quoted Strings	58
2.6	Getting Input from the User	59
2.7	Decision Making: The <code>if</code> Statement and Comparison Operators	61
2.8	Objects and Dynamic Typing	66
2.9	Intro to Data Science: Basic Descriptive Statistics	68
2.10	Wrap-Up	70

3 Control Statements and Program Development 73

3.1	Introduction	74
3.2	Algorithms	74
3.3	Pseudocode	75
3.4	Control Statements	75
3.5	<code>if</code> Statement	78
3.6	<code>if...else</code> and <code>if...elif...else</code> Statements	80
3.7	<code>while</code> Statement	85
3.8	<code>for</code> Statement	86
3.8.1	Iterables, Lists and Iterators	88
3.8.2	Built-In <code>range</code> Function	88
3.9	Augmented Assignments	89
3.10	Program Development: Sequence-Controlled Repetition	90
3.10.1	Requirements Statement	90
3.10.2	Pseudocode for the Algorithm	90
3.10.3	Coding the Algorithm in Python	91
3.10.4	Introduction to Formatted Strings	92
3.11	Program Development: Sentinel-Controlled Repetition	93
3.12	Program Development: Nested Control Statements	97
3.13	Built-In Function <code>range</code> : A Deeper Look	101
3.14	Using Type <code>Decimal</code> for Monetary Amounts	102
3.15	<code>break</code> and <code>continue</code> Statements	105
3.16	Boolean Operators <code>and</code> , <code>or</code> and <code>not</code>	106
3.17	Intro to Data Science: Measures of Central Tendency— Mean, Median and Mode	109
3.18	Wrap-Up	111

4 Functions 119

4.1	Introduction	120
4.2	Defining Functions	120
4.3	Functions with Multiple Parameters	123
4.4	Random-Number Generation	125
4.5	Case Study: A Game of Chance	128
4.6	Python Standard Library	131
4.7	<code>math</code> Module Functions	132
4.8	Using IPython Tab Completion for Discovery	133

4.9	Default Parameter Values	135
4.10	Keyword Arguments	136
4.11	Arbitrary Argument Lists	136
4.12	Methods: Functions That Belong to Objects	138
4.13	Scope Rules	138
4.14	<code>import</code> : A Deeper Look	140
4.15	Passing Arguments to Functions: A Deeper Look	142
4.16	Function-Call Stack	145
4.17	Functional-Style Programming	146
4.18	Intro to Data Science: Measures of Dispersion	148
4.19	Wrap-Up	150

5 Sequences: Lists and Tuples **155**

5.1	Introduction	156
5.2	Lists	156
5.3	Tuples	161
5.4	Unpacking Sequences	163
5.5	Sequence Slicing	166
5.6	<code>del</code> Statement	169
5.7	Passing Lists to Functions	171
5.8	Sorting Lists	172
5.9	Searching Sequences	174
5.10	Other List Methods	176
5.11	Simulating Stacks with Lists	178
5.12	List Comprehensions	179
5.13	Generator Expressions	181
5.14	Filter, Map and Reduce	182
5.15	Other Sequence Processing Functions	185
5.16	Two-Dimensional Lists	187
5.17	Intro to Data Science: Simulation and Static Visualizations	191
5.17.1	Sample Graphs for 600, 60,000 and 6,000,000 Die Rolls	191
5.17.2	Visualizing Die-Roll Frequencies and Percentages	193
5.18	Wrap-Up	199

6 Dictionaries and Sets **209**

6.1	Introduction	210
6.2	Dictionaries	210
6.2.1	Creating a Dictionary	210
6.2.2	Iterating through a Dictionary	212
6.2.3	Basic Dictionary Operations	212
6.2.4	Dictionary Methods <code>keys</code> and <code>values</code>	214
6.2.5	Dictionary Comparisons	216
6.2.6	Example: Dictionary of Student Grades	217
6.2.7	Example: Word Counts	218

x [Contents](#)

6.2.8	Dictionary Method <code>update</code>	220
6.2.9	Dictionary Comprehensions	220
6.3	Sets	221
6.3.1	Comparing Sets	223
6.3.2	Mathematical Set Operations	225
6.3.3	Mutable Set Operators and Methods	226
6.3.4	Set Comprehensions	228
6.4	Intro to Data Science: Dynamic Visualizations	228
6.4.1	How Dynamic Visualization Works	228
6.4.2	Implementing a Dynamic Visualization	231
6.5	Wrap-Up	234

7 [**Array-Oriented Programming with NumPy**](#) **239**

7.1	Introduction	240
7.2	Creating arrays from Existing Data	241
7.3	array Attributes	242
7.4	Filling arrays with Specific Values	244
7.5	Creating arrays from Ranges	244
7.6	List vs. array Performance: Introducing <code>%timeit</code>	246
7.7	array Operators	248
7.8	NumPy Calculation Methods	250
7.9	Universal Functions	252
7.10	Indexing and Slicing	254
7.11	Views: Shallow Copies	256
7.12	Deep Copies	258
7.13	Reshaping and Transposing	259
7.14	Intro to Data Science: pandas Series and DataFrames	262
7.14.1	pandas Series	262
7.14.2	DataFrames	267
7.15	Wrap-Up	275

8 [**Strings: A Deeper Look**](#) **283**

8.1	Introduction	284
8.2	Formatting Strings	285
8.2.1	Presentation Types	285
8.2.2	Field Widths and Alignment	286
8.2.3	Numeric Formatting	287
8.2.4	String's <code>format</code> Method	288
8.3	Concatenating and Repeating Strings	289
8.4	Stripping Whitespace from Strings	290
8.5	Changing Character Case	291
8.6	Comparison Operators for Strings	292
8.7	Searching for Substrings	292
8.8	Replacing Substrings	294

8.9	Splitting and Joining Strings	294
8.10	Characters and Character-Testing Methods	297
8.11	Raw Strings	298
8.12	Introduction to Regular Expressions	299
8.12.1	re Module and Function <code>fullmatch</code>	300
8.12.2	Replacing Substrings and Splitting Strings	303
8.12.3	Other Search Functions; Accessing Matches	304
8.13	Intro to Data Science: Pandas, Regular Expressions and Data Munging	307
8.14	Wrap-Up	312

9 Files and Exceptions **319**

9.1	Introduction	320
9.2	Files	321
9.3	Text-File Processing	321
9.3.1	Writing to a Text File: Introducing the <code>with</code> Statement	322
9.3.2	Reading Data from a Text File	323
9.4	Updating Text Files	325
9.5	Serialization with JSON	327
9.6	Focus on Security: <code>pickle</code> Serialization and Deserialization	330
9.7	Additional Notes Regarding Files	330
9.8	Handling Exceptions	331
9.8.1	Division by Zero and Invalid Input	332
9.8.2	<code>try</code> Statements	332
9.8.3	Catching Multiple Exceptions in One <code>except</code> Clause	335
9.8.4	What Exceptions Does a Function or Method Raise?	336
9.8.5	What Code Should Be Placed in a <code>try</code> Suite?	336
9.9	<code>finally</code> Clause	336
9.10	Explicitly Raising an Exception	339
9.11	(Optional) Stack Unwinding and Tracebacks	339
9.12	Intro to Data Science: Working with CSV Files	342
9.12.1	Python Standard Library Module <code>csv</code>	342
9.12.2	Reading CSV Files into Pandas <code>DataFrames</code>	344
9.12.3	Reading the Titanic Disaster Dataset	346
9.12.4	Simple Data Analysis with the Titanic Disaster Dataset	347
9.12.5	Passenger Age Histogram	348
9.13	Wrap-Up	349

10 Object-Oriented Programming **355**

10.1	Introduction	356
10.2	Custom Class <code>Account</code>	358
10.2.1	Test-Driving Class <code>Account</code>	358
10.2.2	<code>Account</code> Class Definition	360
10.2.3	Composition: Object References as Members of Classes	361
10.3	Controlling Access to Attributes	363

10.4	Properties for Data Access	364
10.4.1	Test-Driving Class <code>Time</code>	364
10.4.2	Class <code>Time</code> Definition	366
10.4.3	Class <code>Time</code> Definition Design Notes	370
10.5	Simulating “Private” Attributes	371
10.6	Case Study: Card Shuffling and Dealing Simulation	373
10.6.1	Test-Driving Classes <code>Card</code> and <code>DeckOfCards</code>	373
10.6.2	Class <code>Card</code> —Introducing Class Attributes	375
10.6.3	Class <code>DeckOfCards</code>	377
10.6.4	Displaying Card Images with Matplotlib	378
10.7	Inheritance: Base Classes and Subclasses	382
10.8	Building an Inheritance Hierarchy; Introducing Polymorphism	384
10.8.1	Base Class <code>CommissionEmployee</code>	384
10.8.2	Subclass <code>SalariedCommissionEmployee</code>	387
10.8.3	Processing <code>CommissionEmployee</code> s and <code>SalariedCommissionEmployee</code> s Polymorphically	391
10.8.4	A Note About Object-Based and Object-Oriented Programming	391
10.9	Duck Typing and Polymorphism	392
10.10	Operator Overloading	393
10.10.1	Test-Driving Class <code>Complex</code>	394
10.10.2	Class <code>Complex</code> Definition	395
10.11	Exception Class Hierarchy and Custom Exceptions	397
10.12	Named Tuples	399
10.13	A Brief Intro to Python 3.7’s New Data Classes	400
10.13.1	Creating a <code>Card</code> Data Class	401
10.13.2	Using the <code>Card</code> Data Class	403
10.13.3	Data Class Advantages over Named Tuples	405
10.13.4	Data Class Advantages over Traditional Classes	406
10.14	Unit Testing with Docstrings and <code>doctest</code>	406
10.15	Namespaces and Scopes	411
10.16	Intro to Data Science: Time Series and Simple Linear Regression	414
10.17	Wrap-Up	423

11	Computer Science Thinking: Recursion, Searching, Sorting and Big O	431
11.1	Introduction	432
11.2	Factorials	433
11.3	Recursive Factorial Example	433
11.4	Recursive Fibonacci Series Example	436
11.5	Recursion vs. Iteration	439
11.6	Searching and Sorting	440
11.7	Linear Search	440
11.8	Efficiency of Algorithms: Big O	442
11.9	Binary Search	444
11.9.1	Binary Search Implementation	445

11.9.2	Big O of the Binary Search	447
11.10	Sorting Algorithms	448
11.11	Selection Sort	448
11.11.1	Selection Sort Implementation	449
11.11.2	Utility Function <code>print_pass</code>	450
11.11.3	Big O of the Selection Sort	451
11.12	Insertion Sort	451
11.12.1	Insertion Sort Implementation	452
11.12.2	Big O of the Insertion Sort	453
11.13	Merge Sort	454
11.13.1	Merge Sort Implementation	454
11.13.2	Big O of the Merge Sort	459
11.14	Big O Summary for This Chapter's Searching and Sorting Algorithms	459
11.15	Visualizing Algorithms	460
11.15.1	Generator Functions	462
11.15.2	Implementing the Selection Sort Animation	463
11.16	Wrap-Up	468

12 Natural Language Processing (NLP) 477

12.1	Introduction	478
12.2	TextBlob	479
12.2.1	Create a TextBlob	481
12.2.2	Tokenizing Text into Sentences and Words	482
12.2.3	Parts-of-Speech Tagging	482
12.2.4	Extracting Noun Phrases	483
12.2.5	Sentiment Analysis with TextBlob's Default Sentiment Analyzer	484
12.2.6	Sentiment Analysis with the <code>NaiveBayesAnalyzer</code>	486
12.2.7	Language Detection and Translation	487
12.2.8	Inflection: Pluralization and Singularization	489
12.2.9	Spell Checking and Correction	489
12.2.10	Normalization: Stemming and Lemmatization	490
12.2.11	Word Frequencies	491
12.2.12	Getting Definitions, Synonyms and Antonyms from WordNet	492
12.2.13	Deleting Stop Words	494
12.2.14	n-grams	496
12.3	Visualizing Word Frequencies with Bar Charts and Word Clouds	497
12.3.1	Visualizing Word Frequencies with Pandas	497
12.3.2	Visualizing Word Frequencies with Word Clouds	500
12.4	Readability Assessment with Textstatistic	503
12.5	Named Entity Recognition with spaCy	505
12.6	Similarity Detection with spaCy	507
12.7	Other NLP Libraries and Tools	509
12.8	Machine Learning and Deep Learning Natural Language Applications	509
12.9	Natural Language Datasets	510
12.10	Wrap-Up	510

13 Data Mining Twitter	515
13.1 Introduction	516
13.2 Overview of the Twitter APIs	518
13.3 Creating a Twitter Account	519
13.4 Getting Twitter Credentials—Creating an App	520
13.5 What's in a Tweet?	521
13.6 Tweepy	525
13.7 Authenticating with Twitter Via Tweepy	525
13.8 Getting Information About a Twitter Account	527
13.9 Introduction to Tweepy Cursors: Getting an Account's Followers and Friends	529
13.9.1 Determining an Account's Followers	529
13.9.2 Determining Whom an Account Follows	532
13.9.3 Getting a User's Recent Tweets	532
13.10 Searching Recent Tweets	534
13.11 Spotting Trends: Twitter Trends API	536
13.11.1 Places with Trending Topics	536
13.11.2 Getting a List of Trending Topics	537
13.11.3 Create a Word Cloud from Trending Topics	539
13.12 Cleaning/Preprocessing Tweets for Analysis	541
13.13 Twitter Streaming API	542
13.13.1 Creating a Subclass of <code>StreamListener</code>	543
13.13.2 Initiating Stream Processing	545
13.14 Tweet Sentiment Analysis	547
13.15 Geocoding and Mapping	551
13.15.1 Getting and Mapping the Tweets	552
13.15.2 Utility Functions in <code>tweetutilities.py</code>	556
13.15.3 Class <code>LocationListener</code>	558
13.16 Ways to Store Tweets	559
13.17 Twitter and Time Series	560
13.18 Wrap-Up	560
14 IBM Watson and Cognitive Computing	565
14.1 Introduction: IBM Watson and Cognitive Computing	566
14.2 IBM Cloud Account and Cloud Console	568
14.3 Watson Services	568
14.4 Additional Services and Tools	572
14.5 Watson Developer Cloud Python SDK	573
14.6 Case Study: Traveler's Companion Translation App	574
14.6.1 Before You Run the App	575
14.6.2 Test-Driving the App	576
14.6.3 <code>SimpleLanguageTranslator.py</code> Script Walkthrough	577
14.7 Watson Resources	587
14.8 Wrap-Up	589

15 Machine Learning: Classification, Regression and Clustering	593
15.1 Introduction to Machine Learning	594
15.1.1 Scikit-Learn	595
15.1.2 Types of Machine Learning	596
15.1.3 Datasets Bundled with Scikit-Learn	598
15.1.4 Steps in a Typical Data Science Study	599
15.2 Case Study: Classification with k-Nearest Neighbors and the Digits Dataset, Part 1	599
15.2.1 k-Nearest Neighbors Algorithm	601
15.2.2 Loading the Dataset	602
15.2.3 Visualizing the Data	606
15.2.4 Splitting the Data for Training and Testing	608
15.2.5 Creating the Model	609
15.2.6 Training the Model	610
15.2.7 Predicting Digit Classes	610
15.3 Case Study: Classification with k-Nearest Neighbors and the Digits Dataset, Part 2	612
15.3.1 Metrics for Model Accuracy	612
15.3.2 K-Fold Cross-Validation	616
15.3.3 Running Multiple Models to Find the Best One	617
15.3.4 Hyperparameter Tuning	619
15.4 Case Study: Time Series and Simple Linear Regression	620
15.5 Case Study: Multiple Linear Regression with the California Housing Dataset	625
15.5.1 Loading the Dataset	626
15.5.2 Exploring the Data with Pandas	628
15.5.3 Visualizing the Features	630
15.5.4 Splitting the Data for Training and Testing	634
15.5.5 Training the Model	634
15.5.6 Testing the Model	635
15.5.7 Visualizing the Expected vs. Predicted Prices	636
15.5.8 Regression Model Metrics	637
15.5.9 Choosing the Best Model	638
15.6 Case Study: Unsupervised Machine Learning, Part 1—Dimensionality Reduction	639
15.7 Case Study: Unsupervised Machine Learning, Part 2—k-Means Clustering	642
15.7.1 Loading the Iris Dataset	644
15.7.2 Exploring the Iris Dataset: Descriptive Statistics with Pandas	646
15.7.3 Visualizing the Dataset with a Seaborn <code>pairplot</code>	647
15.7.4 Using a <code>KMeans</code> Estimator	650
15.7.5 Dimensionality Reduction with Principal Component Analysis	652
15.7.6 Choosing the Best Clustering Estimator	655
15.8 Wrap-Up	656

16 Deep Learning	665
16.1 Introduction	666
16.1.1 Deep Learning Applications	668
16.1.2 Deep Learning Demos	669
16.1.3 Keras Resources	669
16.2 Keras Built-In Datasets	669
16.3 Custom Anaconda Environments	670
16.4 Neural Networks	672
16.5 Tensors	674
16.6 Convolutional Neural Networks for Vision; Multi-Classification with the MNIST Dataset	676
16.6.1 Loading the MNIST Dataset	677
16.6.2 Data Exploration	678
16.6.3 Data Preparation	680
16.6.4 Creating the Neural Network	682
16.6.5 Training and Evaluating the Model	691
16.6.6 Saving and Loading a Model	696
16.7 Visualizing Neural Network Training with TensorBoard	697
16.8 ConvnetJS: Browser-Based Deep-Learning Training and Visualization	700
16.9 Recurrent Neural Networks for Sequences; Sentiment Analysis with the IMDb Dataset	701
16.9.1 Loading the IMDb Movie Reviews Dataset	702
16.9.2 Data Exploration	703
16.9.3 Data Preparation	705
16.9.4 Creating the Neural Network	706
16.9.5 Training and Evaluating the Model	709
16.10 Tuning Deep Learning Models	710
16.11 Convnet Models Pretrained on ImageNet	711
16.12 Reinforcement Learning	712
16.12.1 Deep Q-Learning	713
16.12.2 OpenAI Gym	713
16.13 Wrap-Up	714
17 Big Data: Hadoop, Spark, NoSQL and IoT	723
17.1 Introduction	724
17.2 Relational Databases and Structured Query Language (SQL)	728
17.2.1 A books Database	730
17.2.2 SELECT Queries	734
17.2.3 WHERE Clause	734
17.2.4 ORDER BY Clause	736
17.2.5 Merging Data from Multiple Tables: INNER JOIN	737
17.2.6 INSERT INTO Statement	738
17.2.7 UPDATE Statement	739
17.2.8 DELETE FROM Statement	739

17.3	NoSQL and NewSQL Big-Data Databases: A Brief Tour	741
17.3.1	NoSQL Key–Value Databases	741
17.3.2	NoSQL Document Databases	742
17.3.3	NoSQL Columnar Databases	742
17.3.4	NoSQL Graph Databases	743
17.3.5	NewSQL Databases	743
17.4	Case Study: A MongoDB JSON Document Database	744
17.4.1	Creating the MongoDB Atlas Cluster	745
17.4.2	Streaming Tweets into MongoDB	746
17.5	Hadoop	755
17.5.1	Hadoop Overview	755
17.5.2	Summarizing Word Lengths in <i>Romeo and Juliet</i> via MapReduce	758
17.5.3	Creating an Apache Hadoop Cluster in Microsoft Azure HDInsight	758
17.5.4	Hadoop Streaming	760
17.5.5	Implementing the Mapper	760
17.5.6	Implementing the Reducer	761
17.5.7	Preparing to Run the MapReduce Example	762
17.5.8	Running the MapReduce Job	763
17.6	Spark	766
17.6.1	Spark Overview	766
17.6.2	Docker and the Jupyter Docker Stacks	767
17.6.3	Word Count with Spark	770
17.6.4	Spark Word Count on Microsoft Azure	773
17.7	Spark Streaming: Counting Twitter Hashtags Using the pyspark-notebook Docker Stack	777
17.7.1	Streaming Tweets to a Socket	777
17.7.2	Summarizing Tweet Hashtags; Introducing Spark SQL	780
17.8	Internet of Things and Dashboards	786
17.8.1	Publish and Subscribe	788
17.8.2	Visualizing a PubNub Sample Live Stream with a Freeboard Dashboard	788
17.8.3	Simulating an Internet-Connected Thermostat in Python	790
17.8.4	Creating the Dashboard with Freeboard.io	792
17.8.5	Creating a Python PubNub Subscriber	794
17.9	Wrap-Up	798



Preface

“There’s gold in them thar hills!”¹

For many decades, some powerful trends have been in place. Computer hardware has rapidly been getting faster, cheaper and smaller. Internet bandwidth (that is, its information carrying capacity) has rapidly been getting larger and cheaper. And quality computer software has become ever more abundant and essentially free or nearly free through the “open source” movement. Soon, the “Internet of Things” will connect tens of billions of devices of every imaginable type. These will generate enormous volumes of data at rapidly increasing speeds and quantities.

Not so many years ago, if people had told us that we’d write a college-level introductory programming textbook with words like “Big Data” and “Cloud” in the title and a graphic of a multicolored elephant (emblematic of “big”) on the cover, our reaction might have been, “Huh?” And, if they’d told us we’d include AI (for artificial intelligence) in the title, we might have said, “Really? Isn’t that pretty advanced stuff for novice programmers?”

If people had said, we’d include “Data Science” in the title, we might have responded, “Isn’t data already included in the domain of ‘Computer Science’? Why would we need a separate academic discipline for it?” Well, in programming today, the latest innovations are “all about the data”—*data science*, *data analytics*, big *data*, relational *databases* (SQL), and NoSQL and NewSQL *databases*.

So, here we are! Welcome to *Intro to Python for Computer Science and Data Science: Learning to Program with AI, Big Data and the Cloud*.

In this book, you’ll learn hands-on with today’s most compelling, leading-edge computing technologies—and, as you’ll see, with an easily tunable mix of computer science and data science appropriate for introductory courses in those and related disciplines. And, you’ll program in Python—one of the world’s most popular languages and the fastest growing among them. In this Preface, we present the “soul of the book.”

Professional programmers often quickly discover that they like Python. They appreciate its expressive power, readability, conciseness and interactivity. They like the world of open-source software development that’s generating an ever-growing base of reusable software for an enormous range of application areas.

Whether you’re an instructor, a novice student or an experienced professional programmer, this book has much to offer you. Python is an excellent first programming language for novices and is equally appropriate for developing industrial-strength applications. For the novice, the early chapters establish a solid programming foundation.

We hope you’ll find *Intro to Python for Computer Science and Data Science* educational, entertaining and challenging. It has been a joy to work on this project.

1. Source unknown, frequently misattributed to Mark Twain.

Python for Computer Science and Data Science Education

Many top U.S. universities have switched to Python as their language of choice for teaching introductory computer science, with “eight of the top 10 CS departments (80%), and 27 of the top 39 (69%)” using Python.² It’s now particularly popular for educational and scientific computing,³ and it recently surpassed R as the most popular data science programming language.^{4,5,6}

Modular Architecture

We anticipate that the computer science undergraduate curriculum will evolve to include a data science component—this book is designed to facilitate that and to meet the needs of introductory data science courses with a Python programming component.

The book’s **modular architecture** (please see the **Table of Contents graphic** on the book’s first page) helps us meet the diverse needs of computer science, data science and related audiences. Instructors can adapt it conveniently to a wide range of courses offered to students drawn from many majors.

Chapters 1–11 cover traditional introductory computer science programming topics. Chapters 1–10 each include an *optional* brief **Intro to Data Science** section introducing artificial intelligence, basic descriptive statistics, measures of central tendency and dispersion, simulation, static and dynamic visualization, working with CSV files, pandas for data exploration and data wrangling, time series and simple linear regression. These help you prepare for the data science, AI, big data and cloud case studies in Chapters 12–17, which present opportunities for you to use **real-world datasets** in complete case studies.

After covering Python Chapters 1–5 and a few key parts of Chapters 6–7, you’ll be able to handle significant portions of the **data science, AI and big data case studies** in Chapters 12–17, which are appropriate for all contemporary programming courses:

- Computer science courses will likely work through more of Chapters 1–11 and fewer of the **Intro to Data Science** sections in Chapters 1–10. CS instructors will want to cover some or all of the case-study Chapters 12–17.
- Data science courses will likely work through fewer of Chapters 1–11, most or all of the **Intro to Data Science** sections in Chapters 1–10, and most or all of the case-study Chapters 12–17.

The “Chapter Dependencies” section of this Preface will help instructors plan their syllabi in the context of the book’s unique architecture.

Chapters 12–17 are loaded with cool, powerful, contemporary content. They present hands-on implementation case studies on topics such as supervised machine learning, unsupervised machine learning, deep learning, reinforcement learning (in the exercises), natural

-
2. Guo, Philip., “Python Is Now the Most Popular Introductory Teaching Language at Top U.S. Universities,” ACM, July 07, 2014, <https://cacm.acm.org/blogs/blog-cacm/176450-python-is-now-the-most-popular-introductory-teaching-language-at-top-u-s-universities/fulltext>.
 3. <https://www.oreilly.com/ideas/5-things-to-watch-in-python-in-2017>.
 4. <https://www.kdnuggets.com/2017/08/python-overtakes-r-leader-analytics-data-science.html>.
 5. <https://www.r-bloggers.com/data-science-job-report-2017-r-passes-sas-but-python-leaves-them-both-behind/>.
 6. <https://www.oreilly.com/ideas/5-things-to-watch-in-python-in-2017>.

language processing, data mining Twitter, cognitive computing with IBM’s Watson, big data and more. Along the way, you’ll acquire a **broad literacy** of data science terms and concepts, ranging from briefly defining terms to using concepts in small, medium and large programs. Browsing the book’s detailed index will give you a sense of the breadth of coverage.

Audiences for the Book

The modular architecture makes this book appropriate for several audiences:

- **All standard Python computer science and related majors.** First and foremost, our book is a solid contemporary Python CS 1 entry. The computing curriculum recommendations from the ACM/IEEE list five types of computing programs: Computer Engineering, Computer Science, Information Systems, Information Technology and Software Engineering.⁷ The book is appropriate for each of these.
- **Undergraduate courses for data science majors**—Our book is useful in many data science courses. It follows the curriculum recommendations for **integration of all the key areas in all courses**, as appropriate for intro courses. In the proposed data science curriculum, the book can be the primary textbook for the first computer science course or the first data science course, then be used as a Python reference throughout the upper curriculum.
- **Service courses** for students who are not computer science or data science majors.
- **Graduate courses in data science**—The book can be used as the primary textbook in the first course, then as a Python reference in other graduate-level data science courses.
- **Two-year colleges**—These schools will increasingly offer courses that prepare students for data science programs in the four-year colleges—the book is an appropriate option for that purpose.
- **High schools**—Just as they began teaching computer classes in response to strong interest, many are already teaching Python programming and data science classes.⁸ According to a recent article on LinkedIn, “data science should be taught in high school,” where the “curriculum should mirror the types of careers that our children will go into, focused directly on where jobs and technology are going.”⁹ We believe that data science could soon become a popular college advanced-placement course and that eventually there will be a data science AP exam.
- **Professional industry training courses.**

Key Features

KIS (Keep It Simple), KIS (Keep it Small), KIT (Keep it Topical)

- **Keep it simple**—In every aspect of the book and its instructor and student supplements, we strive for **simplicity and clarity**. For example, when we present nat-

7. <https://www.acm.org/education/curricula-recommendations>.

8. <http://datascience.la/introduction-to-data-science-for-high-school-students/>.

9. <https://www.linkedin.com/pulse/data-science-should-taught-high-school-rebecca-croucher/>.

ural language processing, we use the simple and intuitive TextBlob library rather than the more complex NLTK. In general, when multiple libraries could be used to perform similar tasks, we use the simplest one.

- **Keep it small**—Most of the book’s 538 examples are small—often just a few lines of code, with immediate interactive IPython feedback. We use large examples as appropriate in approximately 40 larger scripts and complete case studies.
- **Keep it topical**—We read scores of recent Python-programming and data science textbooks and professional books. In all we browsed, read or watched about 15,000 current articles, research papers, white papers, videos, blog posts, forum posts and documentation pieces. This enabled us to “take the pulse” of the Python, computer science, data science, AI, big data and cloud communities to create 1566 up-to-the-minute examples, exercises and projects (EEPs).

IPython’s Immediate-Feedback, Explore, Discover and Experiment Pedagogy

- The ideal way to learn from this book is to read it and run the code examples in parallel. Throughout the book, we use the **IPython interpreter**, which provides a friendly, immediate-feedback, interactive mode for quickly exploring, discovering and experimenting with Python and its extensive libraries.
- Most of the code is presented in **small, interactive IPython sessions** (which we call **IIs**). For each code snippet you write, IPython immediately reads it, evaluates it and prints the results. This **instant feedback** keeps your attention, boosts learning, facilitates rapid prototyping and speeds the software-development process.
- Our books always emphasize the **live-code teaching approach**, focusing on *complete, working programs* with *sample inputs and outputs*. IPython’s “magic” is that it turns snippets into live code that “comes alive” as you enter each line. This promotes learning and encourages experimentation.
- IPython is a great way to learn the error messages associated with common errors. We’ll intentionally make errors to show you what happens. When we say something is an error, try it to see what happens.
- We use this same immediate-feedback philosophy in the book’s 557 **Self-Check Exercises** (ideal for “flipped classrooms”—we’ll soon say more about that phenomenon) and many of the 471 end-of-chapter exercises and projects.

Python Programming Fundamentals

- First and foremost, this is an introductory Python textbook. We provide rich coverage of Python and general programming fundamentals.
- We discuss Python’s programming models—**procedural programming**, **functional-style programming** and **object-oriented programming**.
- We emphasize **problem-solving** and **algorithm development**.
- We use best practices to **prepare students for industry**.
- **Functional-style programming** is used throughout the book as appropriate. A chart in Chapter 4 lists most of Python’s key functional-style programming capabilities and the chapters in which we initially cover many of them.

538 Examples, and 471 Exercises and Projects (EEPs)

- Students use a hands-on applied approach to learn from a broad selection of **real-world examples, exercises and projects** (EEPs) drawn from computer science, data science and many other fields.
- The **538 examples** range from individual code snippets to complete computer science, data science, artificial intelligence and big data case studies.
- The **471 exercises and projects** naturally extend the chapter examples. Each chapter concludes with a substantial set of exercises covering a wide variety of topics. This helps instructors tailor their courses to the unique requirements of their audiences and to vary course assignments each semester.
- The EEPs give you an engaging, challenging and entertaining introduction to Python programming, including hands-on AI, computer science and data science.
- Students attack exciting and entertaining challenges with AI, **big data and cloud** technologies like **natural language processing**, **data mining** Twitter, machine learning, deep learning, Hadoop, MapReduce, Spark, IBM Watson, key data science libraries (NumPy, pandas, SciPy, NLTK, TextBlob, spaCy, BeautifulSoup, Textastic, Tweepy, Scikit-learn, Keras), key visualization libraries (Matplotlib, Seaborn, Folium) and more.
- Our EEPs encourage you to think into the future. We had the following idea as we wrote this Preface—although it’s not in the text, many similar thought-provoking projects are: With **deep learning**, the **Internet of Things** and large numbers of TV cameras trained on sporting events, it will become possible to keep *automatic statistics*, review the details of every play and resolve instant-replay reviews immediately. So, fans won’t have to endure the bad calls and delays common in today’s sporting events. Here’s a thought—we can use these technologies to eliminate referees. Why not? We’re increasingly entrusting our lives to other deep-learning-based technologies like **robotic surgeons** and **self-driving cars**!
- The **project exercises** encourage you to go deeper into what you’ve learned and research technologies we have not covered. Projects are often larger in scope and may require significant Internet research and implementation effort.
- In the **instructor supplements**, we provide solutions to many exercises, including most in the core Python Chapters 1–11. **Solutions are available only to instructors**—see the section “Instructor Supplements on Pearson’s Instructor Resource Center” later in this Preface for details. **We do not provide solutions to the project and research exercises.**
- We encourage you to look at lots of **demos** and free **open-source** code examples (available on sites such as **GitHub**) for inspiration on additional class **projects**, term **projects**, directed-study projects, capstone-course projects and thesis research.

557 Self-Check Exercises and Answers

- Most sections end with an average of three **Self-Check Exercises**.
- **Fill-in-the-blank, true/false and discussion Self Checks** enable you to test your understanding of the concepts you just studied.

- IPython interactive Self Checks give you a chance to try out and reinforce the programming techniques you just learned.
- For rapid learning, answers immediately follow all Self-Check Exercises.

Avoid Heavy Math in Favor of English Explanations

- Data science topics can be highly mathematical. This book will be used in first computer science and data science courses where students may not have deep mathematical backgrounds, so we avoid heavy math, leaving it to upper-level courses.
- We capture the conceptual essence of the mathematics and put it to work in our examples, exercises and projects. We do this by using Python libraries such as `statistics`, `NumPy`, `SciPy`, `pandas` and many others, which hide the mathematical complexity. So, it's straightforward for students to get many of the benefits of mathematical techniques like `linear regression` without having to know the mathematics behind them. In the `machine-learning` and `deep-learning` examples, we focus on creating objects that do the math for you "behind the scenes." This is one of the keys to **object-based programming**. It's like driving a car safely to your destination without knowing all the math, engineering and science that goes into building engines, transmissions, power steering and anti-skid braking systems.

Visualizations

- 67 full-color static, dynamic, animated and interactive two-dimensional and three-dimensional visualizations (charts, graphs, pictures, animations etc.) help you understand concepts.
- We focus on high-level visualizations produced by `Matplotlib`, `Seaborn`, `pandas` and `Folium` (for `interactive maps`).
- We use visualizations as a pedagogic tool. For example, we make the **law of large numbers** "come alive" in a dynamic `die-rolling simulation` and bar chart. As the number of rolls increases, you'll see each face's percentage of the total rolls gradually approach 16.667% (1/6th) and the sizes of the bars representing the percentages equalize.
- You need to get to know your data. One way is simply to look at the raw data. For even modest amounts of data, you could rapidly get lost in the detail. Visualizations are especially crucial in big data for `data exploration` and `communicating reproducible research results`, where the data items can number in the millions, billions or more. A common saying is that a picture is worth a thousand words¹⁰—in **big data**, a visualization could be worth billions or more items in a database.
- Sometimes, you need to "fly 40,000 feet above the data" to see it "in the large." **Descriptive statistics** help but can be misleading. Anscombe's quartet, which you'll investigate in the exercises, demonstrates through visualizations that significantly *different* datasets can have *nearly identical* descriptive statistics.
- We show the visualization and animation code so you can implement your own. We also provide the animations in source-code files and as Jupyter Notebooks, so

10. https://en.wikipedia.org/wiki/A_picture_is_worth_a_thousand_words.

you can conveniently customize the code and animation parameters, re-execute the animations and see the effects of the changes.

- Many exercises ask you to create your own visualizations.

Data Experiences

- The undergraduate data science curriculum proposal says “**Data experiences** need to play a central role in all courses.”¹¹
- In the book’s examples, exercises and projects (EEPs), you’ll work with many **real-world datasets and data sources**. There’s a wide variety of **free open datasets** available online for you to experiment with. Some of the sites we reference list hundreds or thousands of datasets. We encourage you to explore these.
- We collected hundreds of syllabi, tracked down **instructor dataset preferences** and researched the most popular datasets for **supervised machine learning, unsupervised machine learning** and **deep learning** studies. Many of the libraries you’ll use come bundled with popular datasets for experimentation.
- You’ll learn the steps required to obtain data and prepare it for analysis, analyze that data using many techniques, tune your models and communicate your results effectively, especially through visualization.

Thinking Like a Developer

- You’ll work with a **developer focus**, using such popular sites as **GitHub** and **StackOverflow**, and doing lots of Internet research. Our **Intro to Data Science** sections and case studies in Chapters 12–17 provide rich data experiences.
- **GitHub** is an excellent venue for **finding open-source code** to incorporate into your projects (and to contribute your code to the **open-source community**). It’s also a crucial element of the software developer’s arsenal with **version control tools** that help teams of developers manage open-source (and private) projects.
- We encourage you to study developers’ code on sites like GitHub.
- To get ready for career work in computer science and data science, you’ll use an extraordinary range of free and open-source Python and data science **libraries**, free and open **real-world datasets** from government, industry and academia, and free, free-trial and **freemium** offerings of software and cloud services.

Hands-On Cloud Computing

- Much of big data analytics occurs in the cloud, where it’s easy to scale *dynamically* the amount of hardware and software your applications need. You’ll work with various cloud-based services (some directly and some indirectly), including Twitter, Google Translate, IBM Watson, Microsoft Azure, OpenMapQuest, geopy, Dweet.io and PubNub. You’ll explore more in the exercises and projects.
- We encourage you to use free, free trial or freemium services from various cloud vendors. We prefer those that don’t require a credit card because you don’t want

11. “Curriculum Guidelines for Undergraduate Programs in Data Science,” <http://www.annualreviews.org/doi/full/10.1146/annurev-statistics-060116-053930> (p. 18).

to risk accidentally running up big bills. If you decide to use a service that requires a credit card, ensure that the tier you’re using for free will not automatically jump to a paid tier.

Database, Big Data and Big Data Infrastructure

- According to IBM (Nov. 2016), 90% of the world’s data was created in the last two years.¹² Evidence indicates that the speed of data creation is accelerating.
- According to a March 2016 *AnalyticsWeek* article, within five years there will be over 50 billion devices connected to the Internet and by 2020 we’ll be producing 1.7 megabytes of new data every second *for every person on the planet!*¹³
- We include an optional treatment of **relational databases** and SQL with SQLite.
- Databases are critical big data infrastructure for storing and manipulating the massive amounts of data you’ll process. Relational databases process *structured data*—they’re not geared to the *unstructured* and *semi-structured data* in big data applications. So, as big data evolved, NoSQL and NewSQL databases were created to handle such data efficiently. We include a **NoSQL** and **NewSQL** overview and a hands-on case study with a **MongoDB JSON document database**.
- We include a solid treatment of **big data hardware and software infrastructure** in Chapter 17, “Big Data: Hadoop, Spark, NoSQL and IoT (Internet of Things).”

Artificial Intelligence Case Studies

- Why doesn’t this book have an artificial intelligence chapter? After all, AI is on the cover. In the case study Chapters 12–16, we present **artificial intelligence** topics (a key intersection between computer science and data science), including **natural language processing**, **data mining** Twitter to perform sentiment analysis, **cognitive computing** with IBM Watson, **supervised machine learning**, **unsupervised machine learning**, **deep learning** and **reinforcement learning** (in the exercises). Chapter 17 presents the big data hardware and software infrastructure that enables computer scientists and data scientists to implement leading-edge AI-based solutions.

Computer Science

- The Python fundamentals treatment in Chapters 1–10 will get you thinking like a computer scientist. Chapter 11, “Computer Science Thinking: Recursion, Searching, Sorting and Big O,” gives you a more advanced perspective—these are classic computer science topics. Chapter 11 emphasizes performance issues.

Built-In Collections: Lists, Tuples, Sets, Dictionaries

- There’s little reason today for most application developers to build *custom* data structures. This is a subject for CS2 courses—our scope is *strictly* CS1 and the corresponding data science course(s). The book features a solid **two-chapter**

12. <https://public.dhe.ibm.com/common/ssi/ecm/wr/en/wrl12345usen/watson-customer-engagement-watson-marketing-wr-other-papers-and-reports-wrl12345usen-20170719.pdf>.
 13. <https://analyticsweek.com/content/big-data-facts/>.

treatment of Python’s built-in data structures—lists, tuples, dictionaries and sets—with which most data-structuring tasks can be accomplished.

Array-Oriented Programming with NumPy Arrays and Pandas Series/DataFrames

- We take an innovative approach in this book by focusing on three key data structures from open-source libraries—NumPy arrays, pandas Series and pandas DataFrames. These libraries are used extensively in data science, computer science, artificial intelligence and big data. NumPy offers as much as two orders of magnitude higher performance than built-in Python lists.
- We include in Chapter 7 a rich treatment of NumPy arrays. Many libraries, such as pandas, are built on NumPy. The **Intro to Data Science** sections in Chapters 7–9 introduce pandas Series and DataFrames, which along with NumPy arrays are then used throughout the remaining chapters.

File Processing and Serialization

- Chapter 9 presents **text-file processing**, then demonstrates how to serialize objects using the popular **JSON (JavaScript Object Notation)** format. JSON is a commonly used data-interchange format that you’ll frequently see used in the data science chapters—often with libraries that hide the JSON details for simplicity.
- Many data science libraries provide built-in file-processing capabilities for loading datasets into your Python programs. In addition to plain text files, we process files in the popular **CSV (comma-separated values) format** using the Python Standard Library’s csv module and capabilities of the pandas data science library.

Object-Based Programming

- In all the Python code we studied during our research for this book, we rarely encountered *custom classes*. These are common in the powerful libraries *used* by Python programmers.
- We emphasize using the enormous number of valuable classes that the Python **open-source community** has packaged into industry standard class libraries. You’ll focus on knowing what libraries are out there, choosing the ones you’ll need for your app, creating objects from existing classes (usually in one or two lines of code) and making them “jump, dance and sing.” This is called **object-based programming**—it enables you to **build impressive applications concisely**, which is a significant part of Python’s appeal.
- With this approach, you’ll be able to use machine learning, deep learning, reinforcement learning (in the exercises) and other AI technologies to solve a wide range of intriguing problems, including **cognitive computing** challenges like **speech recognition** and **computer vision**. In the past, with just an introductory programming course, you never would have been able to tackle such tasks.

Object-Oriented Programming

- For computer science students, developing *custom classes* is a crucial **object-oriented programming** skill, along with inheritance, polymorphism and duck typing. We discuss these in Chapter 10.

- The object-oriented programming treatment is modular, so instructors can present basic or intermediate coverage.
- Chapter 10 includes a discussion of unit testing with `doctest` and a fun card-shuffling-and-dealing simulation.
- The six data science, AI, big data and cloud chapters require only a few straightforward custom class definitions. Instructors who do not wish to cover Chapter 10 can have students simply mimic our class definitions.

Privacy

- In the exercises, you'll research ever-stricter privacy laws such as **HIPAA** (**Health Insurance Portability and Accountability Act**) in the United States and **GDPR** (**General Data Protection Regulation**) for the European Union. A key aspect of privacy is protecting users' **personally identifiable information (PII)**, and a key challenge with big data is that it's easy to cross-reference facts about individuals among databases. We mention privacy issues in several places throughout the book.

Security

- Security is crucial to privacy. We deal with some Python-specific security issues.
- AI and big data present unique privacy, security and ethical challenges. In the exercises, students will research the **OWASP Python Security Project** (<http://www.pythonsecurity.org/>), **anomaly detection**, **blockchain** (the technology behind cryptocurrencies like BitCoin and Ethereum) and more.

Ethics

- Ethics conundrum: Suppose big data analytics with AI predicts that a person with no criminal record has a significant chance of committing a serious crime. Should that person be arrested? In the exercises, you'll research this and other ethical issues, including *deep fakes* (AI-generated images and videos that appear to be real), *bias* in machine learning and *CRISPR gene editing*. Students also investigate privacy and ethical issues surrounding AIs and **intelligent assistants**, such as **IBM Watson**, **Amazon Alexa**, **Apple Siri**, **Google Assistant** and **Microsoft Cortana**. For example, just recently, a judge ordered Amazon to turn over Alexa recordings for use in a criminal case.¹⁴

Reproducibility

- In the sciences in general, and data science in particular, there's a need to reproduce the results of experiments and studies, and to communicate those results effectively. **Jupyter Notebooks** are a preferred means for doing this.
- We provide you with a Jupyter Notebooks experience to help meet the reproducibility recommendations of the data science undergraduate curriculum proposal.
- We discuss *reproducibility* throughout the book in the context of programming techniques and software such as Jupyter Notebooks and Docker.

14. <https://techcrunch.com/2018/11/14/amazon-echo-recordings-judge-murder-case/>.

Transparency

- The data science curriculum proposal mentions data transparency. One aspect of data transparency is the availability of data. Many governments and other organization now adhere to **open-data** principles, enabling anyone to access their data.¹⁵ We point you to a wide range of datasets that are made available by such entities.
- Other aspects of data transparency include determining that data is correct and knowing its origin (think, for example, of “fake news”). Many of the datasets we use are bundled with key libraries we present, such as **Scikit-learn** for machine learning and **Keras** for deep learning. We also point you to various curated **dataset repositories** such as the **University of California Irvine (UCI) Machine Learning Repository** (with 450+ datasets)¹⁶ and **Carnegie Mellon University’s StatLib Datasets Archive** (with 100+ datasets).¹⁷

Performance

- We use the **timeit** profiling tool in several examples and exercises to compare the performance of different approaches to performing the same tasks. Other performance-related discussions include generator expressions, NumPy arrays vs. Python lists, performance of machine-learning and deep-learning models, and Hadoop and Spark distributed-computing performance.

Big Data and Parallelism

- Computer applications have generally been good at doing one thing at a time. Today’s more sophisticated applications need to be able to do many things in parallel. The human brain is believed to have the equivalent of 100 billion parallel processors.¹⁸ For years we’ve written about parallelism at the program level, which is complex and error-prone.
- In this book, rather than writing your own parallelization code, you’ll let libraries like Keras running over TensorFlow, and big data tools like Hadoop and Spark parallelize operations for you. In this big data/AI era, the sheer processing requirements of massive data apps demand taking advantage of true parallelism provided by **multicore processors**, **graphics processing units (GPUs)**, **tensor processing units (TPUs)** and huge **clusters of computers in the cloud**. Some big data tasks could have thousands of processors working in parallel to analyze massive amounts of data in reasonable time. Sequentializing such processing is typically not an option, because it would take too long.

15. https://www.mckinsey.com/~media/McKinsey/Business%20Functions/McKinsey%20Digital/Our%20Insights/Big%20data%20The%20next%20frontier%20for%20innovation/MGI_big_data_full_report.ashx (page 56).

16. <https://archive.ics.uci.edu/ml/datasets.html>.

17. <http://lib.stat.cmu.edu/datasets/>.

18. <https://www.technologyreview.com/s/532291/fmri-data-reveals-the-number-of-parallel-processes-running-in-the-brain/>.

Chapter Dependencies

If you’re an instructor planning your course syllabus or a professional deciding which chapters to read, this section will help you make the best decisions. Please read the one-page **Table of Contents** on the first page of the book—this will quickly familiarize you with the book’s unique architecture. Teaching or reading the chapters in order is easiest. However, much of the content in the Intro to Data Science sections at the ends of Chapters 1–10 and the case studies in Chapters 12–17 requires only Chapters 1–5 and small portions of Chapters 6–10 as discussed below.

Part 1: Python Fundamentals Quickstart

We recommend that all courses cover Python Chapters 1–5:

- **Chapter 1, Introduction to Computers and Python**, introduces concepts that lay the groundwork for the Python programming in Chapters 2–11 and the big data, artificial-intelligence and cloud-based case studies in Chapters 12–17. The chapter also includes **test-drives of IPython and Jupyter Notebooks**.
- **Chapter 2, Introduction to Python Programming**, presents Python programming fundamentals with code examples illustrating key language features.
- **Chapter 3, Control Statements and Program Development**, presents Python’s control statements, focuses on **problem-solving and algorithm development**, and introduces **basic list processing**.
- **Chapter 4, Functions**, introduces program construction using existing functions and custom functions as building blocks, presents **simulation techniques** with **random-number generation** and introduces **tuple fundamentals**.
- **Chapter 5, Sequences: Lists and Tuples**, presents Python’s built-in list and tuple collections in more detail and begins our introduction to **functional-style programming**.

Part 2: Python Data Structures, Strings and Files¹⁹

The following summarizes inter-chapter dependencies for Python Chapters 6–9 and assumes that you’ve read Chapters 1–5.

- **Chapter 6, Dictionaries and Sets**—The Intro to Data Science section is not dependent on Chapter 6’s contents.
- **Chapter 7, Array-Oriented Programming with NumPy**—The Intro to Data Science section requires dictionaries (Chapter 6) and arrays (Chapter 7).
- **Chapter 8, Strings: A Deeper Look**—The Intro to Data Science section requires raw strings and regular expressions (Sections 8.11–8.12), and pandas **Series** and **DataFrame** features from Section 7.14’s Intro to Data Science.
- **Chapter 9, Files and Exceptions**—For **JSON serialization**, it’s useful to understand dictionary fundamentals (Section 6.2). Also, the Intro to Data Science section requires the built-in **open** function and the **with** statement (Section 9.3), and pandas **DataFrame** features from Section 7.14’s Intro to Data Science.

19. We could have included Chapter 5 in Part 2. We placed it in Part 1 because that’s the group of chapters all courses should cover.

Part 3: Python High-End Topics

The following summarizes inter-chapter dependencies for Python Chapters 10–11 and assumes that you've read Chapters 1–5.

- **Chapter 10, Object-Oriented Programming**—The Intro to Data Science requires pandas DataFrame features from the Intro to Data Science Section 7.14. Instructors wanting to cover only **classes and objects** can present Sections 10.1–10.6. Instructors wanting to cover more advanced topics like **inheritance, polymorphism and duck typing**, can present Sections 10.7–10.9. Sections 10.10–10.15 provide additional advanced perspectives.
- **Chapter 11, Computer Science Thinking: Recursion, Searching, Sorting and Big O**—Requires creating and accessing the elements of arrays (Chapter 7), the %timeit magic (Section 7.6), string method join (Section 8.9) and Matplotlib FuncAnimation from Section 6.4's Intro to Data Science.

Part 4: AI, Cloud and Big Data Case Studies

The following summary of inter-chapter dependencies for Chapters 12–17 assumes that you've read Chapters 1–5. Most of Chapters 12–17 also require dictionary fundamentals from Section 6.2.

- **Chapter 12, Natural Language Processing (NLP)**, uses pandas DataFrame features from Section 7.14's Intro to Data Science.
- **Chapter 13, Data Mining Twitter**, uses pandas DataFrame features from Section 7.14's Intro to Data Science, string method join (Section 8.9), JSON fundamentals (Section 9.5), TextBlob (Section 12.2) and Word clouds (Section 12.3). Several examples require defining a class via inheritance (Chapter 10), but readers can simply mimic the class definitions we provide without reading Chapter 10.
- **Chapter 14, IBM Watson and Cognitive Computing**, uses built-in function open and the with statement (Section 9.3).
- **Chapter 15, Machine Learning: Classification, Regression and Clustering**, uses NumPy array fundamentals and method unique (Chapter 7), pandas DataFrame features from Section 7.14's Intro to Data Science and Matplotlib function subplots (Section 10.6).
- **Chapter 16, Deep Learning**, requires NumPy array fundamentals (Chapter 7), string method join (Section 8.9), general machine-learning concepts from Chapter 15 and features from Chapter 15's Case Study: Classification with k-Nearest Neighbors and the Digits Dataset.
- **Chapter 17, Big Data: Hadoop, Spark, NoSQL and IoT**, uses string method split (Section 6.2.7), Matplotlib FuncAnimation from Section 6.4's Intro to Data Science, pandas Series and DataFrame features from Section 7.14's Intro to Data Science, string method join (Section 8.9), the json module (Section 9.5), NLTK stop words (Section 12.2.13) and from Chapter 13 Twitter authentication, Tweepy's StreamListener class for streaming tweets, and the geopy and folium libraries. A few examples require defining a class via inheritance (Chapter 10), but readers can simply mimic the class definitions we provide without reading Chapter 10.

Computing and Data Science Curricula

We read the following ACM/IEEE CS-and-related curriculum documents in preparation for writing this book:

- Computer Science Curricula 2013,²⁰
- CC2020: A Vision on Computing Curricula,²¹
- Information Technology Curricula 2017,²²
- Cybersecurity Curricula 2017,²³

and the 2016 data science initiative “**Curriculum Guidelines for Undergraduate Programs in Data Science**²⁴ from the faculty group sponsored by the NSF and the Institute for Advanced Study.

Computing Curricula

- According to “CC2020: A Vision on Computing Curricula,” the curriculum “needs to be reviewed and updated to include the new and emerging areas of computing such as **cybersecurity** and **data science**.²⁵
- Data science includes key topics (besides general-purpose programming) such as machine learning, deep learning, natural language processing, speech synthesis and recognition and others that are classic artificial intelligence (AI)—and hence CS topics as well.

Data Science Curriculum

- Graduate-level data science is well established and the undergraduate level is growing rapidly to meet strong industry demand. Our hands-on, nonmathematical, project-oriented, programming-intensive approach facilitates moving data science into the undergraduate curriculum, based on the proposed new curriculum.
- There already are lots of undergraduate data science and data analytics programs, but they’re not uniform. That was part of the motivation for the 25 faculty members on the data science curriculum committee to get together in 2016 and develop the proposed 10-course undergraduate major in data science, “Curriculum Guidelines for Undergraduate Programs in Data Science.”
- The curriculum committee says that “many of the courses traditionally found in computer science, statistics, and mathematics offerings should be redesigned for

-
- 20. ACM/IEEE (Assoc. Comput. Mach./Inst. Electr. Eng.). 2013. *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science* (New York: ACM), <http://ai.stanford.edu/users/sahami/CS2013/final-draft/CS2013-final-report.pdf>.
 - 21. A. Clear, A. Parrish, G. van der Veer and M. Zhang “CC2020: A Vision on Computing Curricula,” <https://dl.acm.org/citation.cfm?id=3017690>.
 - 22. *Information Technology Curricula 2017*, <http://www.acm.org/binaries/content/assets/education/it2017.pdf>.
 - 23. *Cybersecurity Curricula 2017*, https://cybered.hosting.acm.org/wp-content/uploads/2018/02/newcover_csec2017.pdf.
 - 24. “Curriculum Guidelines for Undergraduate Programs in Data Science,” <http://www.annualreviews.org/doi/full/10.1146/annurev-statistics-060116-053930>.
 - 25. <http://delivery.acm.org/10.1145/3020000/3017690/p647-clear.pdf>.

the data science major in the interests of efficiency and the potential synergy that integrated courses would offer.”²⁶

- The committee recommends integrating these areas with computational and statistical thinking in all courses, and indicates that *new textbooks will be essential*²⁷—this book is designed with the committee’s recommendations in mind.
- Python has rapidly become one of the world’s most popular general-purpose programming languages. For schools that want to **cover only one language** in their data science major, it’s reasonable that Python be that language.

Data Science Overlaps with Computer Science²⁸

The undergraduate data science curriculum proposal includes algorithm development, programming, computational thinking, data structures, database, mathematics, statistical thinking, machine learning, data science and more—a significant overlap with computer science, especially given that the data science courses include some key AI topics. Even though ours is a Python programming textbook, it touches each of these areas (except for heavy mathematics) from the recommended data science 10-course curriculum, as we efficiently work data science into various examples, exercises, projects and full-implementation case studies.

Key Points from the Data Science Curriculum Proposal

In this section, we call out some key points from the data science undergraduate curriculum proposal²⁹ or its detailed course descriptions appendix.³⁰ We worked hard to incorporate these and many other objectives:

- learn **programming fundamentals** commonly presented in **computer science** courses, including working with **data structures**.
- be able to **solve problems by creating algorithms**.
- work with **procedural, functional and object-oriented programming**.
- receive an integrated presentation of **computational and statistical thinking**, including **exploring concepts via simulations**.
- use **development environments** (we use IPython and Jupyter Notebooks).
- work with **real-world data in practical case studies and projects in every course**.
- **obtain, explore and transform (wrangle) data for analysis**.
- **create static, dynamic and interactive data visualizations**.

26. “Curriculum Guidelines for Undergraduate Programs in Data Science,” <http://www.annualreviews.org/doi/full/10.1146/annurev-statistics-060116-053930> (pp. 16–17).

27. “Curriculum Guidelines for Undergraduate Programs in Data Science,” <http://www.annualreviews.org/doi/full/10.1146/annurev-statistics-060116-053930> (pp. 16–17).

28. This section is intended primarily for data science instructors. Given that the emerging 2020 Computing Curricula for computer science and related disciplines is likely to include some key data science topics, this section includes important information for computer science instructors as well.

29. “Curriculum Guidelines for Undergraduate Programs in Data Science,” <http://www.annualreviews.org/doi/full/10.1146/annurev-statistics-060116-053930>.

30. “Appendix—Detailed Courses for a Proposed Data Science Major,” http://www.annualreviews.org/doi/suppl/10.1146/annurev-statistics-060116-053930/suppl_file/st04_de_veaux_supmat.pdf.

- communicate reproducible results.
- work with existing software and cloud-based tools.
- work with statistical and machine-learning models.
- work with high-performance tools (Hadoop, Spark, MapReduce and NoSQL).
- focus on data's ethics, security, privacy, reproducibility and transparency issues.

Jobs Requiring Data Science Skills

In 2011, McKinsey Global Institute produced their report, “Big data: The next frontier for innovation, competition and productivity.” In it, they said, “The United States alone faces a shortage of 140,000 to 190,000 people with deep analytical skills as well as 1.5 million managers and analysts to analyze big data and make decisions based on their findings.”³¹ This continues to be the case. The August 2018 “LinkedIn Workforce Report” says the United States has a shortage of over 150,000 people with data science skills.³² A 2017 report from IBM, Burning Glass Technologies and the Business-Higher Education Forum, says that by 2020 in the United States there will be hundreds of thousands of new jobs requiring data science skills.³³

Jupyter Notebooks

For your convenience, we provide the book’s examples in **Python source code (.py)** files for use with the command-line IPython interpreter *and* as **Jupyter Notebooks (.ipynb)** files that you can load into your web browser and execute. You can use whichever method of executing code examples you prefer.

Jupyter Notebooks is a free, open-source project that enables authors to combine text, graphics, audio, video, and interactive coding functionality for entering, editing, executing, debugging, and modifying code quickly and conveniently in a web browser. According to the article, “What Is Jupyter?”:

Jupyter has become a standard for scientific research and data analysis. It packages computation and argument together, letting you build “computational narratives”; ... and it simplifies the problem of distributing working software to teammates and associates.³⁴

In our experience, it’s a wonderful learning environment and **rapid prototyping tool** for novices and experienced developers alike. For this reason, we use **Jupyter Notebooks** rather than a traditional **integrated development environment (IDE)**, such as Eclipse, Visual Studio, PyCharm or Spyder. Academics and professionals already use Jupyter extensively for sharing research results. Jupyter Notebooks support is provided through the traditional open-source community mechanisms³⁵ (see “Getting Your Questions Answered” later in this Preface).

31. https://www.mckinsey.com/~media/McKinsey/Business%20Functions/McKinsey%20Digital/Our%20Insights/Big%20data%20The%20next%20frontier%20for%20innovation/MGI_big_data_full_report.ashx (page 3).

32. <https://economicgraph.linkedin.com/resources/linkedin-workforce-report-august-2018>.

33. https://www.burning-glass.com/wp-content/uploads/The_Quant_Crunch.pdf (page 3).

34. <https://www.oreilly.com/ideas/what-is-jupyter>.

35. <https://jupyter.org/community>.

We believe Jupyter Notebooks are a compelling way to teach and learn Python and that most instructors will choose to use Jupyter. The notebooks include:

- examples,
- Self Check exercises,
- all end-of-chapter exercises containing code, such as “What does this code do?” and “What’s wrong with this code?” exercises.
- **Visualizations and animations**, which are a crucial part of the book’s pedagogy. We provide the code in Jupyter Notebooks so students can conveniently **reproduce our results**.

See the Before You Begin section that follows this Preface for software installation details and see the test-drives in Section 1.10 for information on running the book’s examples.

Collaboration and Sharing Results

Working in teams and communicating research results are both emphasized in the proposed undergraduate data science curriculum³⁶ and are important for students moving into data-analytics positions in industry, government or academia:

- The notebooks you create are **easy to share** among team members simply by copying the files or via GitHub.
- Research results, including code and insights, can be shared as static web pages via tools like nbviewer (<https://nbviewer.jupyter.org>) and GitHub—both automatically render notebooks as web pages.

Reproducibility: A Strong Case for Jupyter Notebooks

In data science, and in the sciences in general, experiments and studies should be **reproducible**. This has been written about in the literature for many years, including

- Donald Knuth’s 1992 computer science publication—*Literate Programming*.³⁷
- The article “Language-Agnostic Reproducible Data Analysis Using Literate Programming,”³⁸ which says, “Lir (literate, reproducible computing) is based on the idea of literate programming as proposed by Donald Knuth.”

Essentially, reproducibility captures the complete environment used to produce results—hardware, software, communications, algorithms (especially code), data and the **data’s provenance** (origin and lineage).

The undergraduate data science curriculum proposal mentions reproducibility as a goal in four places. The article “50 Years of Data Science” says, “teaching students to work reproducibly enables easier and deeper evaluation of their work; having them reproduce parts of analyses by others allows them to learn skills like **Exploratory Data Analysis** that are commonly practiced but not yet systematically taught; and training them to work reproducibly will make their post-graduation work more reliable.”³⁹

36. “Curriculum Guidelines for Undergraduate Programs in Data Science,” <http://www.annualreviews.org/doi/full/10.1146/annurev-statistics-060116-053930> (pp. 18–19).

37. Knuth, D., “Literate Programming” (PDF), *The Computer Journal*, British Computer Society, 1992.

38. <http://journals.plos.org/plosone/article?id=10.1371/journal.pone.0164023>.

Docker

In Chapter 17, we'll introduce **Docker**—a tool for packaging software into *containers* that bundle *everything* required to execute that software conveniently, reproducibly and portably across platforms. Some software packages we use in Chapter 17 require complicated setup and configuration. For many of these, you can download free preexisting **Docker containers**. These enable you to avoid complex installation issues and execute software locally on your desktop or notebook computers, making Docker a great way to help you get started with new technologies quickly and conveniently.

Docker also helps with **reproducibility**. You can create custom Docker containers that are configured with the versions of every piece of software and every library you used in your study. This would enable others to recreate the environment you used, then reproduce your work, and will help you reproduce your own results. In Chapter 17, you'll use Docker to download and execute a container that's preconfigured for you to code and run big data Spark applications using Jupyter Notebooks.

Class Tested

While the book was under development, one of our academic reviewers—Dr. Alison Sanchez, Assistant Professor in Economics, University of San Diego—class tested it in a new course, “Business Analytics Strategy.” She commented: “Wonderful for first-time Python learners from all educational backgrounds and majors. My business analytics students had little to no coding experience when they began the course. In addition to loving the material, it was easy for them to follow along with the example exercises and by the end of the course were able to mine and analyze Twitter data using techniques learned from the book. The chapters are clearly written with detailed explanations of the example code—which makes it easy for students without a computer science background to understand. The modular structure, wide range of contemporary data science topics, and companion Jupyter notebooks make this a fantastic resource for instructors and students of a variety of Data Science, Business Analytics and Computer Science courses.”

“Flipped Classroom”

Many instructors are now using “flipped classrooms.”^{40,41} Students learn the content on their own before coming to class (typically via video lectures), and class time is used for tasks such as hands-on coding, working in groups and discussions. Our book and supplements are appropriate for flipped classrooms:

- We provide extensive VideoNotes in which co-author Paul Deitel teaches the concepts in the core Python chapters. See “Student and Instructor Supplements” later in this Preface for details on accessing the videos.
- Some students learn best by—and video is *not* hands-on. **One of the most compelling features of the book** is its interactive approach with 538 Python code

39. “50 Years of Data Science,” <http://courses.csail.mit.edu/18.337/2015/docs/50YearsData-Science.pdf>, p. 33.

40. https://en.wikipedia.org/wiki/Flipped_classroom.

41. <https://www.edsurge.com/news/2018-05-24-a-case-for-flipping-learning-without-videos>.

examples—many with just one or a few snippets—and 557 Self Check exercises with answers. These enable students to learn in small pieces with immediate feedback—perfect for active self-paced learning. Students can easily modify the “hot” code and see the effects of their changes.

- Our Jupyter Notebooks supplements provide a convenient mechanism for students to work with the code.
- We provide 471 exercises and projects, which students can work on at home and/or in class. Many of these are appropriate for group projects.
- We provide lots of probing questions on ethics, privacy, security and more in the exercises and projects. These are appropriate for in-class discussions and group work.

Special Feature: IBM Watson Analytics and Cognitive Computing

Early in our research for this book, we recognized the rapidly growing interest in IBM’s **Watson**. We investigated competitive services and found Watson’s “no credit card required” policy for its “free tiers” to be among the most friendly for our readers.

IBM Watson is a cognitive-computing platform being employed across a wide range of real-world scenarios. Cognitive-computing systems simulate the pattern-recognition and decision-making capabilities of the human brain to “learn” as they consume more data.^{42,43,44} We include a significant hands-on Watson treatment. We use the free **Watson Developer Cloud: Python SDK**, which provides application programming interfaces (APIs) that enable you to interact with Watson’s services programmatically. Watson is fun to use and a great platform for letting your creative juices flow. You’ll demo or use the following Watson APIs: **Conversation**, **Discovery**, **Language Translator**, **Natural Language Classifier**, **Natural Language Understanding**, **Personality Insights**, **Speech to Text**, **Text to Speech**, **Tone Analyzer** and **Visual Recognition**.

Watson’s Lite Tier Services and Watson Case Study

IBM encourages learning and experimentation by providing *free lite tiers* for many of their APIs.⁴⁵ In Chapter 14, you’ll try demos of many Watson services.⁴⁶ Then, you’ll use the lite tiers of Watson’s **Text to Speech**, **Speech to Text** and **Translate** services to implement a “traveler’s assistant” translation app. You’ll speak a question in English, then the app will transcribe your speech to English text, translate the text to Spanish and speak the Spanish text. Next, you’ll speak a Spanish response (in case you don’t speak Spanish, we provide an audio file you can use). Then, the app will quickly transcribe the speech to Spanish text, translate the text to English and speak the English response. Cool stuff!

42. <http://whatis.techtarget.com/definition/cognitive-computing>.

43. https://en.wikipedia.org/wiki/Cognitive_computing.

44. <https://www.forbes.com/sites/bernardmarr/2016/03/23/what-everyone-should-know-about-cognitive-computing>.

45. Always check the latest terms on IBM’s website, as the terms and services may change.

46. <https://console.bluemix.net/catalog/>.

Teaching Approach

Intro to Python for Computer Science and Data Science contains a rich collection of examples, exercises and projects drawn from many fields. Students solve interesting, real-world problems working with **real-world datasets**. The book concentrates on the principles of good **software engineering** and stresses **program clarity**.

Using Fonts for Emphasis

We place the key terms and the index's page reference for each defining occurrence in **bold** text for easier reference. We place on-screen components in the **bold Helvetica** font (for example, the **File** menu) and use the **Lucida** font for Python code (for example, `x = 5`).

Syntax Coloring

The book is in full color. For readability, we syntax color all the code. Our syntax-coloring conventions are as follows:

```
comments appear in green
keywords appear in dark blue
constants and literal values appear in light blue
errors appear in red
all other code appears in black
```

Objectives and Outline

Each chapter begins with objectives that tell you what to expect and give you an opportunity, after reading the chapter, to determine whether it has met the intended goals. The chapter outline enables students to approach the material in top-down fashion.

538 Examples

The book's **538 examples** contain approximately **4000 lines of code**. This is a relatively small amount of code for a book this size and is due to the fact that Python is such an expressive language. Also, our coding style is to use powerful class libraries to do most of the work wherever possible.

160 Tables/Illustrations/Visualizations

Abundant tables, line drawings, and visualizations are included. The visualizations are in color and include some 2D and 3D, some static and dynamic and some interactive.

Programming Wisdom

We integrate into the discussions programming wisdom from the authors' combined nine decades of programming and teaching experience, including:

- **Good programming practices** and preferred Python idioms that help you produce clearer, more understandable and more maintainable programs.
- **Common programming errors** to reduce the likelihood that you'll make them.
- **Error-prevention tips** with suggestions for exposing bugs and removing them from your programs. Many of these tips describe techniques for preventing bugs from getting into your programs in the first place.
- **Performance tips** that highlight opportunities to make your programs run faster or minimize the amount of memory they occupy.

- Software engineering observations that highlight architectural and design issues for proper software construction, especially for larger systems.

Wrap-Up

Chapters 2–17 end with Wrap-Up sections summarizing what you've learned.

Index

We have included an extensive index. The defining occurrences of key terms are highlighted with a **bold** page number.

Software Used in the Book

All the software you'll need for this book is available for Windows, macOS and Linux and is free for download from the Internet. We wrote the book's examples using the free **Anaconda Python distribution**. It includes most of the Python, visualization and data science libraries you'll need, as well as Python, the IPython interpreter, Jupyter Notebooks and Spyder, considered one of the best Python data science integrated development environments (IDEs)—we use only IPython and Jupyter Notebooks for program development in the book. The Before You Begin section discusses installing Anaconda and other items you'll need for working with our examples.

Python Documentation

You'll find the following documentation especially helpful as you work through the book:

- The Python Standard Library:
<https://docs.python.org/3/library/index.html>
- The Python Language Reference:
<https://docs.python.org/3/reference/index.html>
- Python documentation list:
<https://docs.python.org/3/>

Getting Your Questions Answered

Online forums enable you to interact with other Python programmers and get your Python questions answered. Popular Python and general programming forums include:

- python-forum.io
- StackOverflow.com
- <https://www.dreamincode.net/forums/forum/29-python/>

Also, many vendors provide forums for their tools and libraries. Most of the libraries you'll use in this book are managed and maintained at github.com. Some library maintainers provide support through the **Issues** tab on a given library's GitHub page. If you cannot find an answer to your questions online, please see our web page for the book at

<http://www.deitel.com>⁴⁷

47. Our website is undergoing a major upgrade. If you do not find something you need, please write to us directly at deitel@deitel.com.

Getting Jupyter Help

Jupyter Notebooks support is provided through:

- Project Jupyter Google Group:
<https://groups.google.com/forum/#!forum/jupyter>
- Jupyter real-time chat room:
<https://gitter.im/jupyter/jupyter>
- GitHub
<https://github.com/jupyter/help>
- StackOverflow:
<https://stackoverflow.com/questions/tagged/jupyter>
- Jupyter for Education Google Group (for instructors teaching with Jupyter):
<https://groups.google.com/forum/#!forum/jupyter-education>

Student and Instructor Supplements

The following supplements are available to students and instructors.

Code Examples and Getting Started Videos

To get the most out of the presentation, you should execute each code example in parallel with reading the corresponding discussion. On the book's web page at

<http://www.deitel.com>

we provide:

- Downloadable Python source code (.py files) and Jupyter Notebooks (.ipynb files) for the book's **code examples**, for code-based Self-Check Exercises and for **end-of-chapter exercises** that have code as part of the exercise description.
- **Getting Started** videos showing how to use the code examples with IPython and Jupyter Notebooks. We also introduce these tools in Section 1.10.
- **Blog posts and book updates.**

For download instructions, see the *Before You Begin* section that follows this Preface.

Companion Website

The book's Companion Website at

<https://www.pearson.com/deitel>

contains the same code downloads described above in "Code Examples and Getting Started Videos" as well as extensive **VideoNotes** in which co-author Paul Deitel explains most of the examples in the book's core Python chapters.

New copies of this book come with a **Companion Website access code** on the book's inside front cover. If the access code is already visible or there isn't one, you purchased a used book or an edition that does not come with an access code. In this case, you can purchase access directly from the Companion Website.

Instructor Supplements on Pearson's Instructor Resource Center

The following supplements are available to qualified instructors only through Pearson Education's IRC (Instructor Resource Center) at <http://www.pearsonhighered.com/irc>:

- **PowerPoint slides.**
- **Instructor Solutions Manual** with solutions to many of the exercises. Solutions are *not* provided for “project” and “research” exercises—many of which are substantial and appropriate for term projects, directed-study projects, capstone-course projects and thesis topics. **Before assigning a particular exercise for homework, instructors should check the IRC to be sure the solution is available.**
- **Test Item File** with multiple-choice, short-answer questions and answers. These are easy to use in automated assessment tools.

Please do not write to us requesting access to the Pearson Instructor's Resource Center which contains the book's instructor supplements, including exercise solutions. Access is strictly limited to college instructors teaching from the book. Instructors may obtain access through their Pearson representatives. If you're not a registered faculty member, contact your Pearson representative or visit

<https://www.pearson.com/replocator>

Instructor Examination Copies

Instructors can request an **examination copy** of the book from their Pearson representative:

<https://www.pearson.com/replocator>

Keeping in Touch with the Authors

For answers to questions, syllabus assistance or to report an error, send an e-mail to us at
deitel@deitel.com

or interact with us via **social media**:

- **Facebook®** (<http://www.deitel.com/deitelfan>)
- **Twitter®** (@deitel)
- **LinkedIn®** (<http://linkedin.com/company/deitel-&-associates>)
- **YouTube®** (<http://youtube.com/DeitelTV>)

Acknowledgments

We'd like to thank Barbara Deitel for long hours devoted to Internet research on this project. We're fortunate to have worked with the dedicated team of publishing professionals at Pearson. We appreciate the guidance, wisdom and energy of Tracy Johnson (Executive Portfolio Manager, Higher Ed Courseware, Computer Science)—she challenged us at every step of the process to “get it right.” Carole Snyder managed the book's production and interacted with Pearson's permissions team, promptly clearing our graphics and citations. We selected the cover art, and Chuti Prasertsith designed the cover.

We wish to acknowledge the efforts of our academic and professional reviewers. Meghan Jacoby and Patricia Byron-Kimball recruited the reviewers and managed the review process.

Adhering to a tight schedule, the reviewers scrutinized our work, providing countless suggestions for improving the accuracy, completeness and timeliness of the presentation.

Reviewers

Proposal Reviewers

Dr. Irene Bruno, Associate Professor in the Department of Information Sciences and Technology, George Mason University
 Lance Bryant, Associate Professor, Department of Mathematics, Shippensburg University
 Daniel Chen, Data Scientist, Lander Analytics
 Garrett Dancik, Associate Professor of Computer Science/Bioinformatics Department of Computer Science, Eastern Connecticut State University
 Dr. Marsha Davis, Department Chair of Mathematical Sciences, Eastern Connecticut State University
 Roland DePratti, Adjunct Professor of Computer Science, Eastern Connecticut State University
 Shyamal Mitra, Senior Lecturer, Computer Science, University of Texas at Austin
 Dr. Mark Pauley, Senior Research Fellow, Bioinformatics, School of Interdisciplinary Informatics, University of Nebraska at Omaha
 Sean Raleigh, Associate Professor of Mathematics, Chair of Data Science, Westminster College
 Alison Sanchez, Assistant Professor in Economics, University of San Diego

Dr. Harvey Siy, Associate Professor of Computer Science, Information Science and Technology, University of Nebraska at Omaha
 Jamie Whitacre, Independent Data Science Consultant

Book Reviewers

Daniel Chen, Data Scientist, Lander Analytics
 Garrett Dancik, Associate Professor of Computer Science/Bioinformatics, Eastern Connecticut State University
 Pranshu Gupta, Assistant Professor, Computer Science, DeSales University
 David Koop, Assistant Professor, Data Science Program Co-Director, U-Mass Dartmouth
 Ramon Mata-Toledo, Professor, Computer Science, James Madison University
 Shyamal Mitra, Senior Lecturer, Computer Science, University of Texas at Austin
 Alison Sanchez, Assistant Professor in Economics, University of San Diego
 José Antonio González Seco, IT Consultant
 Jamie Whitacre, Independent Data Science Consultant
 Elizabeth Wickes, Lecturer, School of Information Sciences, University of Illinois

A Special Thank You

Our thanks to Prof. Alison Sanchez for class-testing the book prepublication in her new “Business Analytics Strategy” class at the University of San Diego. She reviewed the lengthy proposal, adopting the book sight unseen and signed on as a full-book reviewer in parallel with using the book in her class. Her guidance (and courage) throughout the entire book-development process are sincerely appreciated.

Well, there you have it! As you read the book, we’d appreciate your comments, criticisms, corrections and suggestions for improvement. Please send all correspondence to:

deitel@deitel.com

We’ll respond promptly.

Welcome again to the exciting open-source world of Python programming. We hope you enjoy this look at leading-edge computer-applications development with Python, IPython, Jupyter Notebooks, AI, big data and the cloud. We wish you great success!

Paul and Harvey Deitel

About the Authors

Paul J. Deitel, CEO and Chief Technical Officer of Deitel & Associates, Inc., is an MIT graduate with 38 years of experience in computing. Paul is one of the world's most experienced programming-languages trainers, having taught professional courses to software developers since 1992. He has delivered hundreds of programming courses to industry clients internationally, including Cisco, IBM, Siemens, Sun Microsystems (now Oracle), Dell, Fidelity, NASA at the Kennedy Space Center, the National Severe Storm Laboratory, White Sands Missile Range, Rogue Wave Software, Boeing, Nortel Networks, Puma, iRobot and many more. He and his co-author, Dr. Harvey M. Deitel, are the world's best-selling programming-language textbook/professional book/video authors.

Dr. Harvey M. Deitel, Chairman and Chief Strategy Officer of Deitel & Associates, Inc., has 58 years of experience in computing. Dr. Deitel earned B.S. and M.S. degrees in Electrical Engineering from MIT and a Ph.D. in Mathematics from Boston University—he studied computing in each of these programs before they spun off Computer Science programs. He has extensive college teaching experience, including earning tenure and serving as the Chairman of the Computer Science Department at Boston College before founding Deitel & Associates, Inc., in 1991 with his son, Paul. The Deitels' publications have earned international recognition, with more than 100 translations published in Japanese, German, Russian, Spanish, French, Polish, Italian, Simplified Chinese, Traditional Chinese, Korean, Portuguese, Greek, Urdu and Turkish. Dr. Deitel has delivered hundreds of programming courses to academic, corporate, government and military clients.

About Deitel® & Associates, Inc.

Deitel & Associates, Inc., founded by Paul Deitel and Harvey Deitel, is an internationally recognized authoring and corporate training organization, specializing in computer programming languages, object technology, mobile app development and Internet and web software technology. The company's training clients include some of the world's largest companies, government agencies, branches of the military, and academic institutions. The company offers instructor-led training courses delivered at client sites worldwide on major programming languages and platforms.

Through its 44-year publishing partnership with Pearson/Prentice Hall, Deitel & Associates, Inc., publishes leading-edge programming textbooks and professional books in print and e-book formats, **LiveLessons** video courses, Safari-Live online seminars and **Revel™** interactive multimedia courses. To contact Deitel & Associates, Inc. and the authors, or to request a proposal on-site, instructor-led training, write to:

`deitel@deitel.com`

To learn more about Deitel on-site corporate training, visit

`http://www.deitel.com/training`

Individuals wishing to purchase Deitel books can do so at

`https://www.amazon.com`

Bulk orders by corporations, the government, the military and academic institutions should be placed directly with Pearson. For more information, visit

`https://www.informit.com/store/sales.aspx`



Before You Begin

This section contains information you should review before using this book. We'll post updates at: <http://www.deitel.com>.

Font and Naming Conventions

We show Python code and commands and file and folder names in a **sans-serif** font, and on-screen components, such as menu names, in a **bold sans-serif** font. We use *italics* for emphasis and **bold** occasionally for strong emphasis.

Getting the Code Examples

You can download the `examples.zip` file containing the book's examples from our *Intro to Python for Computer Science and Data Science* web page at:

<http://www.deitel.com>

Click the **Download Examples** link to save the file to your local computer. Most web browsers place the file in your user account's **Downloads** folder. The examples are also available from Pearson's Companion Website for the book at:

<https://pearson.com/deitel>

When the download completes, locate it on your system, and extract its `examples` folder into your user account's **Documents** folder:

- Windows: `C:\Users\YourAccount\Documents\examples`
- macOS or Linux: `~/Documents/examples`

Most operating systems have a built-in extraction tool. You also may use an archive tool such as 7-Zip (www.7-zip.org) or WinZip (www.winzip.com).

Structure of the `examples` Folder

You'll execute three kinds of examples in this book:

- Individual code snippets in the IPython interactive environment.
- Complete applications, which are known as scripts.
- Jupyter Notebooks—a convenient interactive, web-browser-based environment in which you can write and execute code and intermix the code with text, images and video.

We demonstrate each in Section 1.10's test drives.

The `examples` folder contains one subfolder per chapter. These are named `ch##`, where `##` is the two-digit chapter number 01 to 17—for example, `ch01`. Except for Chapters 14, 16 and 17, each chapter’s folder contains the following items:

- `snippets_ipynb`—A folder containing the chapter’s Jupyter Notebook files.
- `snippets_py`—A folder containing Python source code files in which each code snippet we present is separated from the next by a blank line. You can copy and paste these snippets into IPython or into new Jupyter Notebooks that you create.
- Script files and their supporting files.

Chapter 14 contains one application. Chapters 16 and 17 explain where to find the files you need in the `ch16` and `ch17` folders, respectively.

Installing Anaconda

We use the easy-to-install Anaconda Python distribution with this book. It comes with almost everything you’ll need to work with our examples, including:

- the IPython interpreter,
- most of the Python and data science libraries we use,
- a local Jupyter Notebooks server so you can load and execute our notebooks, and
- various other software packages, such as the Spyder Integrated Development Environment (IDE)—we use only IPython and Jupyter Notebooks in this book.

Download the Python 3.x Anaconda installer for Windows, macOS or Linux from:

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

When the download completes, run the installer and follow the on-screen instructions. To ensure that Anaconda runs correctly, do not move its files after you install it.

Updating Anaconda

Next, ensure that Anaconda is up to date. Open a command-line window on your system as follows:

- On macOS, open a **Terminal** from the **Applications** folder’s **Utilities** subfolder.
- On Windows, open the **Anaconda Prompt** from the start menu. When doing this to update Anaconda (as you’ll do here) or to install new packages (discussed momentarily), execute the **Anaconda Prompt** as an *administrator* by right-clicking, then selecting **More > Run as administrator**. (If you cannot find the Anaconda Prompt in the start menu, simply search for it in the **Type here to search** field at the bottom of your screen.)
- On Linux, open your system’s **Terminal** or shell (this varies by Linux distribution).

In your system’s command-line window, execute the following commands to update Anaconda’s installed packages to their latest versions:

1. `conda update conda`
2. `conda update --all`

Package Managers

The `conda` command used above invokes the **conda package manager**—one of the two key Python package managers you’ll use in this book. The other is **pip**. Packages contain the files required to install a given Python library or tool. Throughout the book, you’ll use `conda` to install additional packages, unless those packages are not available through `conda`, in which case you’ll use `pip`. Some people prefer to use `pip` exclusively as it currently supports more packages. If you ever have trouble installing a package with `conda`, try `pip` instead.

Installing the Prospector Static Code Analysis Tool

In the book’s exercises, we ask you to analyze Python code using the Prospector analysis tool, which checks your Python code for common errors and helps you improve your code. To install Prospector and the Python libraries it uses, run the following command in the command-line window:

```
pip install prospector
```

Installing jupyter-matplotlib

We implement several animations using a visualization library called Matplotlib. To use them in Jupyter Notebooks, you must install a tool called `ipymp1`. In the **Terminal**, **Anaconda Command Prompt** or shell you opened previously, execute the following commands¹ one at a time:

```
conda install -c conda-forge ipymp1
conda install nodejs
jupyter labextension install @jupyter-widgets/jupyterlab-manager
jupyter labextension install jupyter-matplotlib
```

Installing the Other Packages

Anaconda comes with approximately 300 popular Python and data science packages for you, such as NumPy, Matplotlib, pandas, Regex, BeautifulSoup, requests, Bokeh, SciPy, SciKit-Learn, Seaborn, Spacy, sqlite, statsmodels and many more. The number of additional packages you’ll need to install throughout the book will be small and we’ll provide installation instructions as necessary. As you discover new packages, their documentation will explain how to install them.

Get a Twitter Developer Account

If you intend to use our “Data Mining Twitter” chapter and any Twitter-based examples in subsequent chapters, apply for a Twitter developer account. Twitter now requires registration for access to their APIs. To apply, fill out and submit the application at

```
https://developer.twitter.com/en/apply-for-access
```

Twitter reviews every application. At the time of this writing, personal developer accounts were being approved immediately and company-account applications were taking from several days to several weeks. Approval is not guaranteed.

1. <https://github.com/matplotlib/jupyter-matplotlib>.

Internet Connection Required in Some Chapters

While using this book, you'll need an Internet connection to install various additional Python libraries. In some chapters, you'll register for accounts with cloud-based services, mostly to use their free tiers. Some services require credit cards to verify your identity. In a few cases, you'll use services that are not free. In these cases, you'll take advantage of monetary credits provided by the vendors so you can try their services without incurring charges. **Caution:** Some cloud-based services incur costs once you set them up. When you complete our case studies using such services, be sure to promptly delete the resources you allocated.

Slight Differences in Program Outputs

When you execute our examples, you might notice some differences between the results we show and your own results:

- Due to differences in how calculations are performed with floating-point numbers (like -123.45 , 7.5 or 0.0236937) across operating systems, you might see minor variations in outputs—especially in digits far to the right of the decimal point.
- When we show outputs that appear in separate windows, we crop the windows to remove their borders.

Getting Your Questions Answered

Online forums enable you to interact with other Python programmers and get your Python questions answered. Popular Python and general programming forums include:

- python-forum.io
- StackOverflow.com
- <https://www.dreamincode.net/forums/forum/29-python/>

Also, many vendors provide forums for their tools and libraries. Most of the libraries you'll use in this book are managed and maintained at github.com. Some library maintainers provide support through the **Issues** tab on a given library's GitHub page. If you cannot find an answer to your questions online, please see our web page for the book at

<http://www.deitel.com>¹

You're now ready to begin reading *Intro to Python for Computer Science and Data Sciences: Learning to Program with AI, Big Data and the Cloud*. We hope you enjoy the book!

1. Our website is undergoing a major upgrade. If you do not find something you need, please write to us directly at deitel@deitel.com.



Intro to Python®

for Computer Science and Data Science



Learning
to Program
with AI, Big Data
and the Cloud

PAUL DEITEL
HARVEY DEITEL

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, 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 ***Intro to Python for Computer Science and Data Science: Learning to Program with AI, Big Data and the Cloud*** by Paul Deitel and Harvey Deitel.

- 1 Go to www.pearson.com/cs-resources
- 2 Enter the title of your textbook or browse by author name.
- 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 student access code.
Do not use a 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 access code can only be used once. This subscription is valid for 12 months upon activation and is not transferable. If the access code has already been revealed it may no longer be valid. If this is the case you can purchase a subscription on the login page for the Companion Website.

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

REVIEWER COMMENTS

“Wonderful for first-time Python learners from all educational backgrounds and majors. My business analytics students had little to no coding experience when they began the course. In addition to loving the material, it was easy for them to follow along with the example exercises and by the end of the course were able to mine and analyze Twitter data using techniques learned from the book. The chapters are clearly written with detailed explanations of the example code, which makes it easy for students without a computer science background to understand. The modular structure, wide range of contemporary data science topics, and companion Jupyter notebooks make this a fantastic resource for instructors and students of a variety of Data Science, Business Analytics, and Computer Science courses. The “Self Checks” are great for students. Fabulous Big Data chapter—it covers all of the relevant programs and platforms. Great Watson chapter! This is the type of material that I look for as someone who teaches Business Analytics. The chapter provided a great overview of the Watson applications. Also, your translation examples are great for students because they provide an “instant reward”—it’s very satisfying for students to implement a task and receive results so quickly. Machine Learning is a huge topic and this chapter serves as a great introduction. I loved the housing data example—very relevant for business analytics students. The chapter was visually stunning.”

—**Alison Sanchez, Assistant Professor in Economics, University of San Diego**

“I like the new combination of topics from computer science, data science, and stats. A compelling feature is the integration of content that is typically considered in separate courses. This is important for building data science programs that are more than just cobbling together math and computer science courses. A book like this may help facilitate expanding our offerings and using Python as a bridge for computer and data science topics. For a data science program that focuses on a single language (mostly), I think Python is probably the way to go.” —**Lance Bryant, Shippensburg University**

“The end-of-the-chapter problems are a real strength of this book (and of Deitel & Deitel books in general). I would likely use this book. The most compelling feature is that it could, theoretically, be used for both computer science and data science programs.”

—**Dr. Mark Pauley, University of Nebraska at Omaha**

“I agree with the authors that CS curricula should include data science—the authors do an excellent job of combining programming and data science topics into an introductory text. The material is presented in digestible sections accompanied by engaging interactive examples. This book should appeal to both computer science students interested in high-level Python programming topics and data science applications, and to data science students who have little or no prior programming experience. Nearly all concepts are accompanied by a worked-out example. A comprehensive overview of object-oriented programming in Python—the use of graphics is sure to engage the reader. A great introduction to Big Data concepts, notably Hadoop, Spark, and IoT. The examples are extremely realistic and practical.” —**Garrett Dancik, Eastern Connecticut State University**

“I can see students feeling really excited about playing with the animations. Covers some of the most modern Python syntax approaches and introduces community standards for style and documentation. The breadth of each chapter and modular design of this book ensure that instructors can select sections tailored to a variety of programming skill levels and domain knowledge. The sorting visualization program is neat. The machine learning chapter does a great job of walking people through the boilerplate code needed for ML in Python. The case studies accomplish this really well. The later examples are so visual. Many of the model evaluation tasks make for really good programming practice.”

—**Elizabeth Wickes, Lecturer, School of Information Sciences, University of Illinois at Urbana-Champaign**

“An engaging, highly-accessible book that will foster curiosity and motivate beginning data scientists to develop essential foundations in Python programming, statistics, data manipulation, working with APIs, data visualization, machine learning, cloud computing, and more. Great walkthrough of the Twitter APIs—sentiment analysis piece is very useful. I’ve taken several classes that cover natural language processing and this is the first time the tools and concepts have been explained so clearly. I appreciate the discussion of serialization with JSON and pickling and when to use one or the other—with an emphasis on using JSON over pickle—good to know there’s a better, safer way! Very clear and engaging coverage of recursion, searching, sorting, and especially Big O—several “Aha” moments. The sorting animation is illustrative, useful, and fun. I look forward to seeing the textbook in use by instructors, students, and aspiring data scientists very soon.” —**Jamie Whitacre, Data Science Consultant**

Introduction to Computers and Python

I



Objectives

In this chapter you'll:

- Learn about exciting recent developments in computing.
- Learn computer hardware, software and Internet basics.
- Understand the data hierarchy from bits to databases.
- Understand the different types of programming languages.
- Understand object-oriented programming basics.
- Understand the strengths of Python and other leading programming languages.
- Understand the importance of libraries.
- Be introduced to key Python and data-science libraries you'll use in this book.
- Test-drive the IPython interpreter's interactive mode for executing Python code.
- Execute a Python script that animates a bar chart.
- Create and test-drive a web-browser-based Jupyter Notebook for executing Python code.
- Learn how big "big data" is and how quickly it's getting even bigger.
- Study a big-data case study on a mobile navigation app.
- Be introduced to artificial intelligence—the intersection between computer science and data science.

Outline

2 Introduction to Computers and Python

1.1	Introduction	
1.2	Hardware and Software	
1.2.1	Moore's Law	
1.2.2	Computer Organization	
1.3	Data Hierarchy	
1.4	Machine Languages, Assembly Languages and High-Level Languages	
1.5	Introduction to Object Technology	
1.6	Operating Systems	
1.7	Python	
1.8	It's the Libraries!	
1.8.1	Python Standard Library	
1.8.2	Data-Science Libraries	
1.9	Other Popular Programming Languages	
1.10	Test-Drive: Using IPython and Jupyter Notebooks	
1.10.1	Using IPython Interactive Mode as a Calculator	
		1.10.2 Executing a Python Program Using the IPython Interpreter
		1.10.3 Writing and Executing Code in a Jupyter Notebook
1.11	Internet and World Wide Web	
1.11.1	Internet: A Network of Networks	
1.11.2	World Wide Web: Making the Internet User-Friendly	
1.11.3	The Cloud	
1.11.4	Internet of Things	
1.12	Software Technologies	
1.13	How Big Is Big Data?	
1.13.1	Big Data Analytics	
1.13.2	Data Science and Big Data Are Making a Difference: Use Cases	
1.14	Case Study—A Big-Data Mobile Application	
1.15	Intro to Data Science: Artificial Intelligence—at the Intersection of CS and Data Science Exercises	

1.1 Introduction

Welcome to Python—one of the world’s most widely used computer programming languages and, according to the *Popularity of Programming Languages (PYPL) Index*, the world’s most popular.¹ You’re probably familiar with many of the powerful tasks computers perform. In this textbook, you’ll get intensive, hands-on experience writing Python instructions that command computers to perform those and other tasks. **Software** (that is, the Python instructions you write, which are also called **code**) controls **hardware** (that is, computers and related devices).

Here, we introduce terminology and concepts that lay the groundwork for the Python programming you’ll learn in Chapters 2–11 and the big-data, artificial-intelligence and cloud-based case studies we present in Chapters 12–17. We’ll introduce hardware and software concepts and overview the data hierarchy—from individual bits to databases, which store the massive amounts of data companies need to implement contemporary applications such as Google Search, Waze, Uber, Airbnb and a myriad of others.

We’ll discuss the types of programming languages and introduce *object-oriented programming* terminology and concepts. You’ll learn why Python has become so popular. We’ll introduce the Python Standard Library and various data-science libraries that help you avoid “reinventing the wheel.” You’ll use these libraries to create *software objects* that you’ll interact with to perform significant tasks with modest numbers of instructions. We’ll introduce additional software technologies that you’re likely to use as you develop software.

Next, you’ll work through three test-drives showing how to execute Python code:

- In the first, you’ll use IPython to execute Python instructions interactively and immediately see their results.

1. <https://pypl.github.io/PYPL.html> (as of January 2019).

- In the second, you'll execute a substantial Python application that will display an animated bar chart summarizing rolls of a six-sided die as they occur. You'll see the "Law of Large Numbers" in action. In Chapter 6, you'll build this application with the Matplotlib visualization library.
- In the last, we'll introduce Jupyter Notebooks using JupyterLab—an interactive, web-browser-based tool in which you can conveniently write and execute Python instructions. Jupyter Notebooks enable you to include text, images, audios, videos, animations and code.

In the past, most computer applications ran on "standalone" computers (that is, not networked together). Today's applications can be written with the aim of communicating among the world's computers via the Internet. We'll introduce the Internet, the World Wide Web, the Cloud and the Internet of Things (IoT), laying the groundwork for the contemporary applications you'll develop in Chapters 12–17.

You'll learn just how big "big data" is and how quickly it's getting even bigger. Next, we'll present a big-data case study on the Waze mobile navigation app, which uses many current technologies to provide dynamic driving directions that get you to your destination as quickly and as safely as possible. As we walk through those technologies, we'll mention where you'll use many of them in this book. The chapter closes with our first Intro to Data Science section in which we discuss a key intersection between computer science and data science—artificial intelligence.

1.2 Hardware and Software

Computers can perform calculations and make logical decisions phenomenally faster than human beings can. Many of today's personal computers can perform billions of calculations in one second—more than a human can perform in a lifetime. *Supercomputers* are already performing *thousands of trillions (quadrillions)* of instructions per second! IBM has developed the IBM Summit supercomputer, which can perform over 122 quadrillion calculations per second (122 *petaflops*)² To put that in perspective, *the IBM Summit supercomputer can perform in one second almost 16 million calculations for every person on the planet!*³ And supercomputing upper limits are growing quickly.

Computers process data under the control of sequences of instructions called **computer programs** (or simply **programs**). These software programs guide the computer through ordered actions specified by people called **computer programmers**.

A computer consists of various physical devices referred to as hardware (such as the keyboard, screen, mouse, solid-state disks, hard disks, memory, DVD drives and processing units). Computing costs are *dropping dramatically*, due to rapid developments in hardware and software technologies. Computers that might have filled large rooms and cost millions of dollars decades ago are now inscribed on computer chips smaller than a fingernail, costing perhaps a few dollars each. Ironically, silicon is one of the most abundant materials on Earth—it's an ingredient in common sand. Silicon-chip technology has made computing so economical that computers have become a commodity.

-
2. <https://en.wikipedia.org/wiki/FLOPS>.
 3. For perspective on how far computing performance has come, consider this: In his early computing days, Harvey Deitel used the Digital Equipment Corporation PDP-1 (<https://en.wikipedia.org/wiki/PDP-1>), which was capable of performing only 93,458 operations per second.

1.2.1 Moore's Law

Every year, you probably expect to pay at least a little more for most products and services. The opposite has been the case in the computer and communications fields, especially with regard to the hardware supporting these technologies. For many decades, hardware costs have fallen rapidly.

Every year or two, the capacities of computers have approximately *doubled* inexpensively. This remarkable trend often is called **Moore's Law**, named for the person who identified it in the 1960s, Gordon Moore, co-founder of Intel—one of the leading manufacturers of the processors in today's computers and embedded systems. Moore's Law and related observations apply especially to

- the amount of memory that computers have for programs,
- the amount of secondary storage (such as solid-state drive storage) they have to hold programs and data over longer periods of time, and
- their processor speeds—the speeds at which they *execute* their programs (that is, do their work).

Similar growth has occurred in the communications field—costs have plummeted as enormous demand for communications *bandwidth* (that is, information-carrying capacity) has attracted intense competition. We know of no other fields in which technology improves so quickly and costs fall so rapidly. Such phenomenal improvement is truly fostering the *Information Revolution*.

1.2.2 Computer Organization

Regardless of differences in *physical* appearance, computers can be envisioned as divided into various **logical units** or sections:

Input Unit

This “receiving” section obtains information (data and computer programs) from **input devices** and places it at the disposal of the other units for processing. Most user input is entered into computers through keyboards, touch screens and mouse devices. Other forms of input include receiving voice commands, scanning images and barcodes, reading from secondary storage devices (such as hard drives, Blu-ray Disc™ drives and USB flash drives—also called “thumb drives” or “memory sticks”), receiving video from a webcam and having your computer receive information from the Internet (such as when you stream videos from YouTube® or download e-books from Amazon). Newer forms of input include position data from a GPS device, motion and orientation information from an *accelerometer* (a device that responds to up/down, left/right and forward/backward acceleration) in a smartphone or wireless game controller (such as those for Microsoft® Xbox®, Nintendo Switch™ and Sony® PlayStation®) and voice input from intelligent assistants like Apple Siri®, Amazon Echo® and Google Home®.

Output Unit

This “shipping” section takes information the computer has processed and places it on various **output devices** to make it available for use outside the computer. Most information that's output from computers today is displayed on screens (including touch screens), printed on paper (“going green” discourages this), played as audio or video on smart-

phones, tablets, PCs and giant screens in sports stadiums, transmitted over the Internet or used to control other devices, such as self-driving cars, robots and “intelligent” appliances. Information is also commonly output to secondary storage devices, such as solid-state drives (SSDs), hard drives, DVD drives and USB flash drives. Popular recent forms of output are smartphone and game-controller vibration, virtual reality devices like Oculus Rift®, Sony® PlayStation® VR and Google Daydream View™ and Samsung Gear VR®, and mixed reality devices like Magic Leap® One and Microsoft HoloLens™.

Memory Unit

This rapid-access, relatively low-capacity “warehouse” section retains information that has been entered through the input unit, making it immediately available for processing when needed. The memory unit also retains processed information until it can be placed on output devices by the output unit. Information in the memory unit is *volatile*—it’s typically lost when the computer’s power is turned off. The memory unit is often called either **memory**, **primary memory** or **RAM** (Random Access Memory). Main memories on desktop and notebook computers contain as much as 128 GB of RAM, though 8 to 16 GB is most common. GB stands for gigabytes; a gigabyte is approximately one billion bytes. A **byte** is eight bits. A bit is either a 0 or a 1.

Arithmetic and Logic Unit (ALU)

This “manufacturing” section performs *calculations*, such as addition, subtraction, multiplication and division. It also contains the *decision* mechanisms that allow the computer, for example, to compare two items from the memory unit to determine whether they’re equal. In today’s systems, the ALU is part of the next logical unit, the CPU.

Central Processing Unit (CPU)

This “administrative” section coordinates and supervises the operation of the other sections. The CPU tells the input unit when information should be read into the memory unit, tells the ALU when information from the memory unit should be used in calculations and tells the output unit when to send information from the memory unit to specific output devices. Most computers have **multicore processors** that implement multiple processors on a single integrated-circuit chip. Such processors can perform many operations simultaneously. A *dual-core processor* has two CPUs, a *quad-core processor* has four and an *octa-core processor* has eight. Intel has some processors with up to 72 cores. Today’s desktop computers have processors that can execute billions of instructions per second.

Secondary Storage Unit

This is the long-term, high-capacity “warehousing” section. Programs or data not actively being used by the other units normally are placed on secondary storage devices (e.g., your *hard drive*) until they’re again needed, possibly hours, days, months or even years later. Information on secondary storage devices is *persistent*—it’s preserved even when the computer’s power is turned off. Secondary storage information takes much longer to access than information in primary memory, but its cost per unit is much less. Examples of secondary storage devices include solid-state drives (SSDs), hard drives, read/write Blu-ray drives and USB flash drives. Many current drives hold terabytes (TB) of data—a **terabyte** is approximately one trillion bytes). Typical hard drives on desktop and notebook computers hold up to 4 TB, and some recent desktop-computer hard drives hold up to 15 TB.⁴

✓ Self Check for Section 1.2

1 (Fill-In) Every year or two, the capacities of computers have approximately doubled inexpensively. This remarkable trend often is called _____.

Answer: Moore's Law.

2 (True/False) Information in the memory unit is *persistent*—it's preserved even when the computer's power is turned off

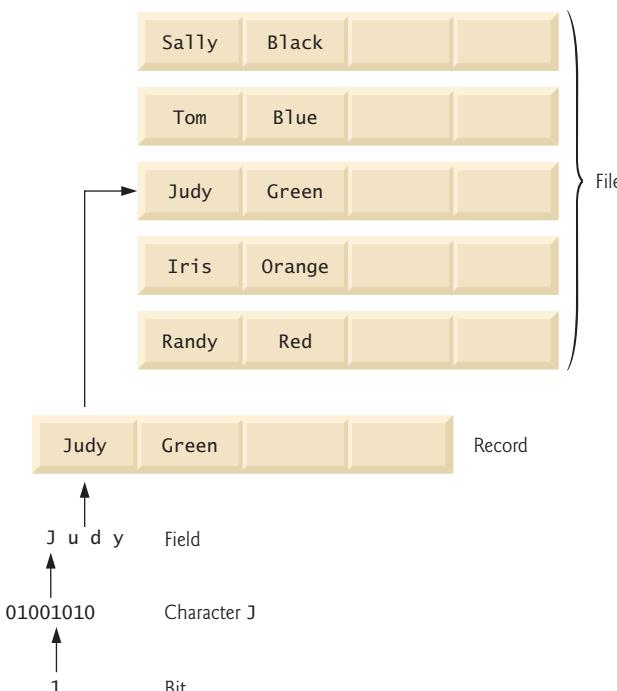
Answer: False. Information in the memory unit is *volatile*—it's typically lost when the computer's power is turned off.

3 (Fill-In) Most computers have _____ processors that implement multiple processors on a single integrated-circuit chip. Such processors can perform many operations simultaneously.

Answer: multicore.

1.3 Data Hierarchy

Data items processed by computers form a **data hierarchy** that becomes larger and more complex in structure as we progress from the simplest data items (called “bits”) to richer ones, such as characters and fields. The following diagram illustrates a portion of the data hierarchy:



4. <https://www.zdnet.com/article/worlds-biggest-hard-drive-meet-western-digital-s-15tb-monster/>.

Bits

A **bit** (short for “*binary digit*”—a digit that can assume one of *two* values) is the smallest data item in a computer. It can have the value 0 or 1. Remarkably, the impressive functions performed by computers involve only the simplest manipulations of 0s and 1s—*examining a bit’s value, setting a bit’s value and reversing a bit’s value* (from 1 to 0 or from 0 to 1). Bits form the basis of the binary number system, which you can study in-depth in our online “Number Systems” appendix.

Characters

Work with data in the low-level form of bits is tedious. Instead, people prefer to work with *decimal digits* (0–9), *letters* (A–Z and a–z) and *special symbols* such as

\$ @ % & * () - + " : ; , ? /

Digits, letters and special symbols are known as **characters**. The computer’s **character set** contains the characters used to write programs and represent data items. Computers process only 1s and 0s, so a computer’s character set represents every character as a pattern of 1s and 0s. Python uses **Unicode®** characters that are composed of one, two, three or four bytes (8, 16, 24 or 32 bits, respectively)—known as **UTF-8 encoding**.⁵

Unicode contains characters for many of the world’s languages. The **ASCII (American Standard Code for Information Interchange)** character set is a subset of Unicode that represents letters (a–z and A–Z), digits and some common special characters. You can view the ASCII subset of Unicode at

<https://www.unicode.org/charts/PDF/U0000.pdf>

The Unicode charts for all languages, symbols, emojis and more are viewable at

<http://www.unicode.org/charts/>

Fields

Just as characters are composed of bits, **fields** are composed of characters or bytes. A field is a group of characters or bytes that conveys meaning. For example, a field consisting of uppercase and lowercase letters can be used to represent a person’s name, and a field consisting of decimal digits could represent a person’s age.

Records

Several related fields can be used to compose a **record**. In a payroll system, for example, the record for an employee might consist of the following fields (possible types for these fields are shown in parentheses):

- Employee identification number (a whole number).
- Name (a string of characters).
- Address (a string of characters).
- Hourly pay rate (a number with a decimal point).
- Year-to-date earnings (a number with a decimal point).
- Amount of taxes withheld (a number with a decimal point).

5. <https://docs.python.org/3/howto/unicode.html>.

8 Introduction to Computers and Python

Thus, a record is a group of related fields. All the fields listed above belong to the same employee. A company might have many employees and a payroll record for each.

Files

A **file** is a group of related records. More generally, a file contains arbitrary data in arbitrary formats. In some operating systems, a file is viewed simply as a *sequence of bytes*—any organization of the bytes in a file, such as organizing the data into records, is a view created by the application programmer. You'll see how to do that in Chapter 9, "Files and Exceptions." It's not unusual for an organization to have many files, some containing billions, or even trillions, of characters of information.

Databases

A **database** is a collection of data organized for easy access and manipulation. The most popular model is the *relational database*, in which data is stored in simple *tables*. A table includes *records* and *fields*. For example, a table of students might include first name, last name, major, year, student ID number and grade-point-average fields. The data for each student is a record, and the individual pieces of information in each record are the fields. You can *search*, *sort* and otherwise manipulate the data, based on its relationship to multiple tables or databases. For example, a university might use data from the student database in combination with data from databases of courses, on-campus housing, meal plans, etc. We discuss databases in Chapter 17, "Big Data: Hadoop, Spark, NoSQL and IoT."

Big Data

The table below shows some common byte measurements:

Unit	Bytes	Which is approximately
1 kilobyte (KB)	1024 bytes	10^3 (1024) bytes exactly
1 megabyte (MB)	1024 kilobytes	10^6 (1,000,000) bytes
1 gigabyte (GB)	1024 megabytes	10^9 (1,000,000,000) bytes
1 terabyte (TB)	1024 gigabytes	10^{12} (1,000,000,000,000) bytes
1 petabyte (PB)	1024 terabytes	10^{15} (1,000,000,000,000,000) bytes
1 exabyte (EB)	1024 petabytes	10^{18} (1,000,000,000,000,000,000) bytes
1 zettabyte (ZB)	1024 exabytes	10^{21} (1,000,000,000,000,000,000,000) bytes

The amount of data being produced worldwide is enormous and its growth is accelerating. **Big data** applications deal with massive amounts of data. This field is growing quickly, creating lots of opportunity for software developers. Millions of IT jobs globally already are supporting big data applications. Section 1.13 discusses big data in more depth. You'll study big data and associated technologies in Chapter 17.



Self Check

- I (Fill-In) A(n) _____ (short for "binary digit"—a digit that can assume one of two values) is the smallest data item in a computer.

Answer: bit.

2 (True/False) In some operating systems, a file is viewed simply as a sequence of bytes—any organization of the bytes in a file, such as organizing the data into records, is a view created by the application programmer.

Answer: True.

3 (Fill-In) A database is a collection of data organized for easy access and manipulation. The most popular model is the _____ database, in which data is stored in simple tables.

Answer: relational

1.4 Machine Languages, Assembly Languages and High-Level Languages

Programmers write instructions in various programming languages, some directly understandable by computers and others requiring intermediate *translation* steps. Hundreds of such languages are in use today. These may be divided into three general types:

1. Machine languages.
2. Assembly languages.
3. High-level languages.

Machine Languages

Any computer can directly understand only its own **machine language**, defined by its hardware design. Machine languages generally consist of strings of numbers (ultimately reduced to 1s and 0s) that instruct computers to perform their most elementary operations one at a time. Machine languages are *machine dependent* (a particular machine language can be used on only one type of computer). Such languages are cumbersome for humans. For example, here's a section of an early machine-language payroll program that adds overtime pay to base pay and stores the result in gross pay:

```
+1300042774
+1400593419
+1200274027
```

Assembly Languages and Assemblers

Programming in machine language was simply too slow and tedious for most programmers. Instead of using the strings of numbers that computers could directly understand, programmers began using English-like abbreviations to represent elementary operations. These abbreviations formed the basis of **assembly languages**. *Translator programs* called **assemblers** were developed to convert assembly-language programs to machine language at computer speeds. The following section of an assembly-language payroll program also adds overtime pay to base pay and stores the result in gross pay:

```
load    basepay
add    overpay
store   grosspay
```

Although such code is clearer to humans, it's incomprehensible to computers until translated to machine language.

High-Level Languages and Compilers

With the advent of assembly languages, computer usage increased rapidly, but programmers still had to use numerous instructions to accomplish even the simplest tasks. To speed the programming process, **high-level languages** were developed in which single statements could be written to accomplish substantial tasks. A typical high-level-language program contains many statements, known as the program's **source code**.

Translator programs called **compilers** convert high-level-language source code into machine language. High-level languages allow you to write instructions that look almost like everyday English and contain commonly used mathematical notations. A payroll program written in a high-level language might contain a *single* statement such as

```
grossPay = basePay + overtimePay
```

From the programmer's standpoint, high-level languages are preferable to machine and assembly languages. Python is among the world's most widely used high-level programming languages.

Interpreters

Compiling a large high-level language program into machine language can take considerable computer time. *Interpreter* programs, developed to execute high-level language programs directly, avoid the delay of compilation, although they run slower than compiled programs. The most widely used Python implementation—CPython (which is written in the C programming language)—uses a clever mixture of compilation and interpretation to run programs.⁶



Self Check

- 1 (Fill-In) Translator programs called _____ convert assembly-language programs to machine language at computer speeds.

Answer: assemblers.

- 2 (Fill-In) _____ programs, developed to execute high-level-language programs directly, avoid the delay of compilation, although they run slower than compiled programs

Answer: Interpreter.

- 3 (True/False) High-level languages allow you to write instructions that look almost like everyday English and contain commonly used mathematical notations.

Answer: True.

1.5 Introduction to Object Technology

As demands for new and more powerful software are soaring, building software quickly, correctly and economically is important. *Objects*, or more precisely, the *classes* objects come from, are essentially *reusable* software components. There are date objects, time objects, audio objects, video objects, automobile objects, people objects, etc. Almost any *noun* can be reasonably represented as a software object in terms of *attributes* (e.g., name, color and size) and *behaviors* (e.g., calculating, moving and communicating). Software-development groups can use a modular, object-oriented design-and-implementation approach to be

6. <https://opensource.com/article/18/4/introduction-python-bytecode>.

much more productive than with earlier popular techniques like “structured programming.” Object-oriented programs are often easier to understand, correct and modify.

Automobile as an Object

To help you understand objects and their contents, let’s begin with a simple analogy. Suppose you want to *drive a car and make it go faster by pressing its accelerator pedal*. What must happen before you can do this? Well, before you can drive a car, someone has to *design* it. A car typically begins as engineering drawings, similar to the *blueprints* that describe the design of a house. These drawings include the design for an accelerator pedal. The pedal *hides* from the driver the complex mechanisms that make the car go faster, just as the brake pedal “hides” the mechanisms that slow the car, and the steering wheel “hides” the mechanisms that turn the car. This enables people with little or no knowledge of how engines, braking and steering mechanisms work to drive a car easily.

Just as you cannot cook meals in the blueprint of a kitchen, you cannot drive a car’s engineering drawings. Before you can drive a car, it must be *built* from the engineering drawings that describe it. A completed car has an *actual* accelerator pedal to make it go faster, but even that’s not enough—the car won’t accelerate on its own (hopefully!), so the driver must *press* the pedal to accelerate the car.

Methods and Classes

Let’s use our car example to introduce some key object-oriented programming concepts. Performing a task in a program requires a **method**. The method houses the program statements that perform its tasks. The method hides these statements from its user, just as the accelerator pedal of a car hides from the driver the mechanisms of making the car go faster. In Python, a program unit called a **class** houses the set of methods that perform the class’s tasks. For example, a class that represents a bank account might contain one method to *deposit* money to an account, another to *withdraw* money from an account and a third to *inquire* what the account’s balance is. A class is similar in concept to a car’s engineering drawings, which house the design of an accelerator pedal, steering wheel, and so on.

Instantiation

Just as someone has to *build a car* from its engineering drawings before you can drive a car, you must *build an object* of a class before a program can perform the tasks that the class’s methods define. The process of doing this is called *instantiation*. An object is then referred to as an **instance** of its class.

Reuse

Just as a car’s engineering drawings can be *reused* many times to build many cars, you can *reuse* a class many times to build many objects. Reuse of existing classes when building new classes and programs saves time and effort. Reuse also helps you build more reliable and effective systems because existing classes and components often have undergone extensive *testing, debugging* (that is, finding and removing errors) and *performance tuning*. Just as the notion of *interchangeable parts* was crucial to the Industrial Revolution, reusable classes are crucial to the software revolution that has been spurred by object technology.

In Python, you’ll typically use a *building-block approach* to create your programs. To avoid reinventing the wheel, you’ll use existing high-quality pieces wherever possible. This software reuse is a key benefit of object-oriented programming.

Messages and Method Calls

When you drive a car, pressing its gas pedal sends a *message* to the car to perform a task—that is, to go faster. Similarly, you *send messages to an object*. Each message is implemented as a **method call** that tells a method of the object to perform its task. For example, a program might call a bank-account object’s *deposit* method to increase the account’s balance.

Attributes and Instance Variables

A car, besides having capabilities to accomplish tasks, also has *attributes*, such as its color, its number of doors, the amount of gas in its tank, its current speed and its record of total miles driven (i.e., its odometer reading). Like its capabilities, the car’s attributes are represented as part of its design in its engineering diagrams (which, for example, include an odometer and a fuel gauge). As you drive an actual car, these attributes are carried along with the car. Every car maintains its *own* attributes. For example, each car knows how much gas is in its own gas tank, but *not* how much is in the tanks of *other* cars.

An object, similarly, has attributes that it carries along as it’s used in a program. These attributes are specified as part of the object’s class. For example, a bank-account object has a *balance attribute* that represents the amount of money in the account. Each bank-account object knows the balance in the account it represents, but *not* the balances of the *other* accounts in the bank. Attributes are specified by the class’s **instance variables**. A class’s (and its object’s) attributes and methods are intimately related, so classes wrap together their attributes and methods.

Inheritance

A new class of objects can be created conveniently by **inheritance**—the new class (called the **subclass**) starts with the characteristics of an existing class (called the **superclass**), possibly customizing them and adding unique characteristics of its own. In our car analogy, an object of class “convertible” certainly *is an* object of the more *general* class “automobile,” but more *specifically*, the roof can be raised or lowered.

Object-Oriented Analysis and Design (OOAD)

Soon you’ll be writing programs in Python. How will you create the **code** (i.e., the program instructions) for your programs? Perhaps, like many programmers, you’ll simply turn on your computer and start typing. This approach may work for small programs (like the ones we present in the early chapters of the book), but what if you were asked to create a software system to control thousands of automated teller machines for a major bank? Or suppose you were asked to work on a team of 1,000 software developers building the next generation of the U.S. air traffic control system? For projects so large and complex, you should not simply sit down and start writing programs.

To create the best solutions, you should follow a detailed **analysis** process for determining your project’s **requirements** (i.e., defining *what* the system is supposed to do), then develop a **design** that satisfies them (i.e., specifying *how* the system should do it). Ideally, you’d go through this process and carefully review the design (and have your design reviewed by other software professionals) before writing any code. If this process involves analyzing and designing your system from an object-oriented point of view, it’s called an **object-oriented analysis-and-design (OOAD) process**. Languages like Python are object-oriented. Programming in such a language, called **object-oriented programming (OOP)**, allows you to implement an object-oriented design as a working system.

Self Check for Section 1.5

1 (*Fill-In*) To create the best solutions, you should follow a detailed analysis process for determining your project's _____ (i.e., defining *what* the system is supposed to do) and developing a design that satisfies them (i.e., specifying *how* the system should do it).

Answer: requirements.

2 (*Fill-In*) The size, shape, color and weight of an object are _____ of the object's class.

Answer: attributes.

3 (*True/False*) Objects, or more precisely, the classes objects come from, are essentially reusable software components.

Answer: True.

1.6 Operating Systems

Operating systems are software systems that make using computers more convenient for users, application developers and system administrators. They provide services that allow each application to execute safely, efficiently and *concurrently* with other applications. The software that contains the core components of the operating system is called the **kernel**. Linux, Windows and macOS are popular desktop computer operating systems—you can use any of these with this book. The most popular mobile operating systems used in smartphones and tablets are Google's Android and Apple's iOS.

Windows—A Proprietary Operating System

In the mid-1980s, Microsoft developed the **Windows operating system**, consisting of a graphical user interface built on top of DOS (Disk Operating System)—an enormously popular personal-computer operating system that users interacted with by typing commands. Windows 10 is Microsoft's latest operating system—it includes the Cortana personal assistant for voice interactions. Windows is a *proprietary* operating system—it's controlled by Microsoft exclusively. Windows is by far the world's most widely used desktop operating system.

Linux—An Open-Source Operating System

The **Linux operating system** is among the greatest successes of the *open-source* movement. **Open-source software** departs from the *proprietary* software development style that dominated software's early years. With open-source development, individuals and companies *contribute* their efforts in developing, maintaining and evolving software in exchange for the right to use that software for their own purposes, typically at *no charge*. Open-source code is often scrutinized by a much larger audience than proprietary software, so errors often get removed faster. Open source also encourages innovation.

There are many organizations in the open-source community. Some key ones are:

- **Python Software Foundation** (responsible for Python).
- **GitHub** (provides tools for managing open-source projects—it has millions of them under development).
- The **Apache Software Foundation** (originally the creators of the Apache web server, they now oversee 350 open-source projects, including several big data infrastructure technologies we present in Chapter 17).

- The **Eclipse Foundation** (the Eclipse Integrated Development Environment helps programmers conveniently develop software)
- The **Mozilla Foundation** (creators of the Firefox web browser)
- **OpenML** (which focuses on open-source tools and data for machine learning—you’ll explore machine learning in Chapter 15).
- **OpenAI** (which does research on artificial intelligence and publishes open-source tools used in AI reinforcement-learning research).
- **OpenCV** (which focuses on open-source computer-vision tools that can be used across a range of operating systems and programming languages—you’ll study computer-vision applications in Chapter 16).

Rapid improvements to computing and communications, decreasing costs and open-source software have made it much easier and more economical to create software-based businesses now than just a decade ago. A great example is Facebook, which was launched from a college dorm room and built with open-source software.

The **Linux kernel** is the core of the most popular open-source, freely distributed, full-featured operating system. It’s developed by a loosely organized team of volunteers and is popular in servers, personal computers and embedded systems (such as the computer systems at the heart of smartphones, smart TVs and automobile systems). Unlike that of proprietary operating systems like Microsoft’s Windows and Apple’s macOS, Linux source code (the program code) is available to the public for examination and modification and is free to download and install. As a result, Linux users benefit from a huge community of developers actively debugging and improving the kernel, and the ability to customize the operating system to meet specific needs.

Apple’s macOS and Apple’s iOS for iPhone® and iPad® Devices

Apple, founded in 1976 by Steve Jobs and Steve Wozniak, quickly became a leader in personal computing. In 1979, Jobs and several Apple employees visited Xerox PARC (Palo Alto Research Center) to learn about Xerox’s desktop computer that featured a graphical user interface (GUI). That GUI served as the inspiration for the Apple Macintosh, launched in 1984.

The Objective-C programming language, created by Stepstone in the early 1980s, added capabilities for object-oriented programming (OOP) to the C programming language. Steve Jobs left Apple in 1985 and founded NeXT Inc. In 1988, NeXT licensed Objective-C from Stepstone and developed an Objective-C compiler and libraries which were used as the platform for the NeXTSTEP operating system’s user interface, and Interface Builder—used to construct graphical user interfaces.

Jobs returned to Apple in 1996 when they bought NeXT. Apple’s **macOS operating system** is a descendant of NeXTSTEP. Apple’s proprietary operating system, **iOS**, is derived from macOS and is used in the iPhone, iPad, Apple Watch and Apple TV devices. In 2014, Apple introduced its new Swift programming language, which became open source in 2015. The iOS app-development community has largely shifted from Objective-C to Swift.

Google’s Android

Android—the fastest growing mobile and smartphone operating system—is based on the Linux kernel and the Java programming language. Android is open source and free.

According to idc.com, as of 2018, Android had 86.8% of the global smartphone market share, compared to 13.2% for Apple.⁷ The Android operating system is used in numerous smartphones, e-reader devices, tablets, in-store touch-screen kiosks, cars, robots, multimedia players and more.

Billions of Devices

In use today are Billions of personal computers and an even larger number of mobile devices. The following table lists many computerized devices. The explosive growth of mobile phones, tablets and other devices is creating significant opportunities for programming mobile apps. There are now various tools that enable you to use Python for Android and iOS app development, including BeeWare, Kivy, PyMob, Pythonista and others. Many are **cross-platform**, meaning that you can use them to develop apps that will run portably on Android, iOS and other platforms (like the web).

Computerized devices		
Access control systems	Airplane systems	ATMs
Automobiles	Blu-ray Disc™ players	Building controls
Cable boxes	Copiers	Credit cards
CT scanners	Desktop computers	e-Readers
Game consoles	GPS navigation systems	Home appliances
Home security systems	Internet-of-Things gateways	Light switches
Logic controllers	Lottery systems	Medical devices
Mobile phones	MRIs	Network switches
Optical sensors	Parking meters	Personal computers
Point-of-sale terminals	Printers	Robots
Routers	Servers	Smartcards
Smart meters	Smartpens	Smartphones
Tablets	Televisions	Thermostats
Transportation passes	TV set-top boxes	Vehicle diagnostic systems



Self Check for Section 1.6

- 1 (Fill-In)** Windows is a(n) _____ operating system—it's controlled by Microsoft exclusively.

Answer: proprietary.

- 2 (True/False)** Proprietary code is often scrutinized by a much larger audience than open-source software, so errors often get removed faster.

Answer: False. Open-source code is often scrutinized by a much larger audience than proprietary software, so errors often get removed faster.

- 3 (True/False)** iOS dominates the global smartphone market over Android.

Answer: False. Android currently controls 88% of the smartphone market.

7. <https://www.idc.com/promo/smartphone-market-share/os>.

1.7 Python

Python is an object-oriented scripting language that was released publicly in 1991. It was developed by Guido van Rossum of the National Research Institute for Mathematics and Computer Science in Amsterdam.

Python has rapidly become one of the world's most popular programming languages. It's now particularly popular for educational and scientific computing,⁸ and it recently surpassed the programming language R as the most popular data-science programming language.^{9,10,11} Here are some reasons why Python is popular and everyone should consider learning it:^{12,13,14}

- It's open source, free and widely available with a massive open-source community.
- It's easier to learn than languages like C, C++, C# and Java, enabling novices and professional developers to get up to speed quickly.
- It's easier to read than many other popular programming languages.
- It's widely used in education.¹⁵
- It enhances developer productivity with extensive standard libraries and *thousands* of third-party open-source libraries, so programmers can write code faster and perform complex tasks with minimal code. We'll say more about this in Section 1.8.
- There are massive numbers of free open-source Python applications.
- It's popular in web development (e.g., Django, Flask).
- It supports popular programming paradigms—procedural, functional, object-oriented and reflective.¹⁶ We'll begin introducing functional-style programming features in Chapter 4 and use them in subsequent chapters.
- It simplifies concurrent programming—with asyncio and async/await, you're able to write single-threaded concurrent code¹⁷, greatly simplifying the inherently complex processes of writing, debugging and maintaining that code.¹⁸
- There are lots of capabilities for enhancing Python performance.
- It's used to build anything from simple scripts to complex apps with massive numbers of users, such as Dropbox, YouTube, Reddit, Instagram and Quora.¹⁹

8. <https://www.oreilly.com/ideas/5-things-to-watch-in-python-in-2017>.

9. <https://www.kdnuggets.com/2017/08/python-overtakes-r-leader-analytics-data-science.html>.

10. <https://www.r-bloggers.com/data-science-job-report-2017-r-passes-sas-but-python-leaves-them-both-behind/>.

11. <https://www.oreilly.com/ideas/5-things-to-watch-in-python-in-2017>.

12. <https://dbader.org/blog/why-learn-python>.

13. <https://simpleprogrammer.com/2017/01/18/7-reasons-why-you-should-learn-python/>.

14. <https://www.oreilly.com/ideas/5-things-to-watch-in-python-in-2017>.

15. Tollervey, N., *Python in Education: Teach, Learn, Program* (O'Reilly Media, Inc., 2015).

16. [https://en.wikipedia.org/wiki/Python_\(programming_language\)](https://en.wikipedia.org/wiki/Python_(programming_language)).

17. <https://docs.python.org/3/library/asyncio.html>.

18. <https://www.oreilly.com/ideas/5-things-to-watch-in-python-in-2017>.

19. https://www.hartmannsoftware.com/Blog/Articles_from_Software_Fans/Most-Famous-Software-Programs-Written-in-Python.

- It's popular in artificial intelligence, which is enjoying explosive growth, in part because of its special relationship with data science.
- It's widely used in the financial community.²⁰
- There's an extensive job market for Python programmers across many disciplines, especially in data-science-oriented positions, and Python jobs are among the highest paid of all programming jobs.^{21,22}

Anaconda Python Distribution

We use the Anaconda Python distribution because it's easy to install on Windows, macOS and Linux and supports the latest versions of Python (3.7 at the time of this writing), the IPython interpreter (introduced in Section 1.10.1) and Jupyter Notebooks (introduced in Section 1.10.3). Anaconda also includes other software packages and libraries commonly used in Python programming and data science, allowing students to focus on learning Python, computer science and data science, rather than software installation issues. The IPython interpreter²³ has features that help students and professionals explore, discover and experiment with Python, the Python Standard Library and the extensive set of third-party libraries.

Zen of Python

We adhere to Tim Peters' *The Zen of Python*, which summarizes Python creator Guido van Rossum's design principles for the language. This list can be viewed in IPython with the command `import this`. The Zen of Python is defined in Python Enhancement Proposal (PEP) 20. "A PEP is a design document providing information to the Python community, or describing a new feature for Python or its processes or environment."²⁴



Self Check

1 (Fill-In) The _____ summarizes Python creator Guido van Rossum's design principles for the Python language.

Answer: Zen of Python.

2 (True/False) The Python language supports popular programming paradigms—procedural, functional, object-oriented and reflective.

Answer: True.

3 (True/False) R is most the popular data-science programming language.

Answer: False. Python recently surpassed R as the most popular data-science programming language.

-
20. Kolanovic, M. and R. Krishnamachari, *Big Data and AI Strategies: Machine Learning and Alternative Data Approach to Investing* (J.P. Morgan, 2017).
 21. <https://www.infoworld.com/article/3170838/developer/get-paid-10-programming-languages-to-learn-in-2017.html>.
 22. <https://medium.com/@ChallengeRocket/top-10-of-programming-languages-with-the-highest-salaries-in-2017-4390f468256e>.
 23. <https://ipython.org/>.
 24. <https://www.python.org/dev/peps/pep-0001/>.

1.8 It's the Libraries!

Throughout the book, we focus on using existing libraries to help you avoid “reinventing the wheel,” thus leveraging your program-development efforts. Often, rather than developing lots of original code—a costly and time-consuming process—you can simply create an object of a pre-existing library class, which takes only a single Python statement. So, libraries will help you perform significant tasks with modest amounts of code. You’ll use a broad range of Python standard libraries, data-science libraries and other third-party libraries.

1.8.1 Python Standard Library

The **Python Standard Library** provides rich capabilities for text/binary data processing, mathematics, functional-style programming, file/directory access, data persistence, data compression/archiving, cryptography, operating-system services, concurrent programming, interprocess communication, networking protocols, JSON/XML/other Internet data formats, multimedia, internationalization, GUI, debugging, profiling and more. The following table lists some of the Python Standard Library modules that we use in examples or that you’ll explore in the exercises.

Some of the Python Standard Library modules we use in the book

collections	—Additional data structures beyond lists, tuples, dictionaries and sets.	os	—Interacting with the operating system.
csv	—Processing comma-separated value files.	timeit	—Performance analysis.
datetime, time	—Date and time manipulations.	queue	—First-in, first-out data structure.
decimal	—Fixed-point and floating-point arithmetic, including monetary calculations.	random	—Pseudorandom numbers.
doctest	—Simple unit testing via validation tests and expected results embedded in docstrings.	re	—Regular expressions for pattern matching.
json	—JavaScript Object Notation (JSON) processing for use with web services and NoSQL document databases.	sqlite3	—SQLite relational database access.
math	—Common math constants and operations.	statistics	—Mathematical statistics functions like <code>mean</code> , <code>median</code> , <code>mode</code> and <code>variance</code> .
		string	—String processing.
		sys	—Command-line argument processing; standard input, standard output and standard error streams.

1.8.2 Data-Science Libraries

Python has an enormous and rapidly growing community of open-source developers in many fields. One of the biggest reasons for Python’s popularity is the extraordinary range of open-source libraries developed by the open-source community. One of our goals is to create examples, exercises, projects (EEPs) and implementation case studies that give you an engaging, challenging and entertaining introduction to Python programming, while also involving you in hands-on data science, key data-science libraries and more. You’ll be amazed at the substantial tasks you can accomplish in just a few lines of code. The following table lists various popular data-science libraries. You’ll use many of these as you work through our data-science examples, exercises and projects. For visualization, we’ll focus primarily on Matplotlib and Seaborn, but there are many more. For a nice summary of Python visualization libraries see <http://pyviz.org/>.

Popular Python libraries used in data science

Scientific Computing and Statistics

NumPy (Numerical Python)—Python does not have a built-in array data structure. It uses lists, which are convenient but relatively slow. NumPy provides the more efficient `ndarray` data structure to represent lists and matrices, and it also provides routines for processing such data structures.

SciPy (Scientific Python)—Built on NumPy, SciPy adds routines for scientific processing, such as integrals, differential equations, additional matrix processing and more. scipy.org controls SciPy and NumPy.

StatsModels—Provides support for estimations of statistical models, statistical tests and statistical data exploration.

Data Manipulation and Analysis

Pandas—An extremely popular library for data manipulations. Pandas makes abundant use of NumPy's `ndarray`. Its two key data structures are `Series` (one dimensional) and `DataFrames` (two dimensional).

Visualization

Matplotlib—A highly customizable visualization and plotting library. Supported plots include regular, scatter, bar, contour, pie, quiver, grid, polar axis, 3D and text.

Seaborn—A higher-level visualization library built on Matplotlib. Seaborn adds a nicer look-and-feel, additional visualizations and enables you to create visualizations with less code.

Machine Learning, Deep Learning and Reinforcement Learning

scikit-learn—Top machine-learning library. Machine learning is a subset of AI. Deep learning is a subset of machine learning that focuses on neural networks.

Keras—One of the easiest to use deep-learning libraries. Keras runs on top of TensorFlow (Google), CNTK (Microsoft's cognitive toolkit for deep learning) or Theano (Université de Montréal).

TensorFlow—From Google, this is the most widely used deep learning library. TensorFlow works with GPUs (graphics processing units) or Google's custom TPUs (Tensor processing units) for performance. TensorFlow is important in AI and big data analytics—where processing demands are enormous. You'll use the version of Keras that's built into TensorFlow.

OpenAI Gym—A library and environment for developing, testing and comparing reinforcement-learning algorithms. You'll explore this in the Chapter 16 exercises.

Natural Language Processing (NLP)

NLTK (Natural Language Toolkit)—Used for natural language processing (NLP) tasks.

TextBlob—An object-oriented NLP text-processing library built on the NLTK and pattern NLP libraries. TextBlob simplifies many NLP tasks.

Gensim—Similar to NLTK. Commonly used to build an index for a collection of documents, then determine how similar another document is to each of those in the index. You'll explore this in the Chapter 12 exercises.



Self Check for Section 1.8

- I (Fill-In) _____ help you avoid “reinventing the wheel,” thus leveraging your program-development efforts.

Answer: Libraries.

2 (Fill-In) The _____ provides rich capabilities for many common Python programming tasks.

Answer: Python Standard Library.

1.9 Other Popular Programming Languages

The following is a brief introduction to several other popular programming languages—in the next section, we take a deeper look at Python:

- *Basic* was developed in the 1960s at Dartmouth College to familiarize novices with programming techniques. Many of its latest versions are object-oriented.
- *C* was developed in the early 1970s by Dennis Ritchie at Bell Laboratories. It initially became widely known as the UNIX operating system's development language. Today, most code for general-purpose operating systems and other performance-critical systems is written in C or C++.
- *C++*, which is based on C, was developed by Bjarne Stroustrup in the early 1980s at Bell Laboratories. C++ provides features that enhance the C language and adds capabilities for object-oriented programming.
- *Java*—Sun Microsystems in 1991 funded an internal corporate research project led by James Gosling, which resulted in the C++-based object-oriented programming language called Java. A key goal of Java is to enable developers to write programs that will run on a great variety of computer systems. This is called “write once, run anywhere.” Java is used to develop enterprise applications, to enhance the functionality of web servers (the computers that provide the content to our web browsers), to provide applications for consumer devices (e.g., smartphones, tablets, television set-top boxes, appliances, automobiles and more) and for many other purposes. Java was originally the key language for developing Android smartphone and tablet apps, though several other languages are now supported.
- *C#* (based on C++ and Java) is one of Microsoft's three primary object-oriented programming languages—the other two are Visual C++ and Visual Basic. C# was developed to integrate the web into computer applications and is now widely used to develop many types of applications. As part of Microsoft's many open-source initiatives implemented over the last few years, they now offer open-source versions of C# and Visual Basic.
- *JavaScript* is the most widely used scripting language. It's primarily used to add programmability to web pages—for example, animations and interactivity with the user. All major web browsers support it. Many Python visualization libraries output JavaScript as part of visualizations that you can interact with in your web browser. Tools like NodeJS also enable JavaScript to run outside of web browsers.
- *Swift*, which was introduced in 2014, is Apple's programming language for developing iOS and macOS apps. Swift is a contemporary language that includes popular features from languages such as Objective-C, Java, C#, Ruby, Python and others. Swift is open source, so it can be used on non-Apple platforms as well.
- *R* is a popular open-source programming language for statistical applications and visualization. Python and R are the two most widely data-science languages.

Self Check

1 (*Fill-In*) Today, most code for general-purpose operating systems and other performance-critical systems is written in _____.

Answer: C or C++.

2 (*Fill-In*) A key goal of _____ is to enable developers to write programs that will run on a great variety of computer systems and computer-controlled devices. This is sometimes called “write once, run anywhere.”

Answer: Java.

1.10 Test-Drive: Using IPython and Jupyter Notebooks

In this section, you’ll test-drive the IPython interpreter²⁵ in two modes:

- In **interactive mode**, you’ll enter small bits of Python code called **snippets** and immediately see their results.
- In **script mode**, you’ll execute code loaded from a file that has the .py extension (short for Python). Such files are called **scripts** or **programs**, and they’re generally longer than the code snippets you’ll do in interactive mode.

Then, you’ll learn how to use the browser-based environment known as the Jupyter Notebook for writing and executing Python code.²⁶

1.10.1 Using IPython Interactive Mode as a Calculator

Let’s use IPython interactive mode to evaluate simple arithmetic expressions.

Entering IPython in Interactive Mode

First, open a command-line window on your system:

- On macOS, open a **Terminal** from the **Applications** folder’s **Utilities** subfolder.
- On Windows, open the **Anaconda Command Prompt** from the start menu.
- On Linux, open your system’s **Terminal** or shell (this varies by Linux distribution).

In the command-line window, type ipython, then press *Enter* (or *Return*). You’ll see text like the following, this varies by platform and by IPython version:

```
Python 3.7.0 | packaged by conda-forge | (default, Jan 20 2019, 17:24:52)
Type 'copyright', 'credits' or 'license' for more information
IPython 6.5.0 -- An enhanced Interactive Python. Type '?' for help.
```

In [1]:

The text "In [1]:" is a *prompt*, indicating that IPython is waiting for your input. You can type ? for help or begin entering snippets, as you’ll do momentarily.

25. Before reading this section, follow the instructions in the Before You Begin section to install the Anaconda Python distribution, which contains the IPython interpreter.

26. Jupyter supports many programming languages by installing their "kernels." For more information see <https://github.com/jupyter/jupyter/wiki/Jupyter-kernels>.

Evaluating Expressions

In interactive mode, you can evaluate expressions:

```
In [1]: 45 + 72
Out[1]: 117
```

```
In [2]:
```

After you type `45 + 72` and press *Enter*, IPython *reads* the snippet, *evaluates* it and *prints* its result in `Out[1]`.²⁷ Then IPython displays the `In [2]` prompt to show that it's waiting for you to enter your second snippet. For each new snippet, IPython adds 1 to the number in the square brackets. Each `In [1]` prompt in the book indicates that we've started a new interactive session. We generally do that for each new section of a chapter.

Let's evaluate a more complex expression:

```
In [2]: 5 * (12.7 - 4) / 2
Out[2]: 21.75
```

Python uses the asterisk (*) for multiplication and the forward slash (/) for division. As in mathematics, parentheses force the evaluation order, so the parenthesized expression `(12.7 - 4)` evaluates first, giving 8.7. Next, `5 * 8.7` evaluates giving 43.5. Then, `43.5 / 2` evaluates, giving the result 21.75, which IPython displays in `Out[2]`. Whole numbers, like 5, 4 and 2, are called **integers**. Numbers with decimal points, like 12.7, 43.5 and 21.75, are called **floating-point numbers**.

Exiting Interactive Mode

To leave interactive mode, you can:

- Type the `exit` command at the current `In []` prompt and press *Enter* to exit immediately.
- Type the key sequence `<Ctrl> + d` (or `<control> + d`). This displays the prompt "Do you really want to exit ([y]/n)". The square brackets around `y` indicate that it's the default response—pressing *Enter* submits the default response and exits.
- Type `<Ctrl> + d` (or `<control> + d`) twice (macOS and Linux only).



Self Check

1 (*Fill-In*) In IPython interactive mode, you'll enter small bits of Python code called _____ and immediately see their results.

Answer: snippets.

2 In IPython _____ mode, you'll execute Python code loaded from a file that has the `.py` extension (short for Python).

Answer: script.

3 (*IPython Session*) Evaluate the expression `5 * (3 + 4)` both with and without the parentheses. Do you get the same result? Why or why not?

27. In the next chapter, you'll see that there are some cases in which `Out[]` is not displayed.

Answer: You get different results because snippet [1] first calculates $3 + 4$, which is 7, then multiplies that by 5. Snippet [2] first multiplies $5 * 3$, which is 15, then adds that to 4.

```
In [1]: 5 * (3 + 4)
Out[1]: 35
```

```
In [2]: 5 * 3 + 4
Out[2]: 19
```

1.10.2 Executing a Python Program Using the IPython Interpreter

In this section, you’ll execute a script named `RollDieDynamic.py` that you’ll write in Chapter 6. The `.py` extension indicates that the file contains Python source code. The script `RollDieDynamic.py` simulates rolling a six-sided die. It presents a colorful animated visualization that dynamically graphs the frequencies of each die face.

Changing to This Chapter’s Examples Folder

You’ll find the script in the book’s `ch01` source-code folder. In the Before You Begin section you extracted the `examples` folder to your user account’s `Documents` folder. Each chapter has a folder containing that chapter’s source code. The folder is named `ch##`, where `##` is a two-digit chapter number from 01 to 17. First, open your system’s command-line window. Next, use the `cd` (“change directory”) command to change to the `ch01` folder:

- On macOS/Linux, type `cd ~/Documents/examples/ch01`, then press *Enter*.
- On Windows, type `cd C:\Users\YourAccount\Documents\examples\ch01`, then press *Enter*.

Executing the Script

To execute the script, type the following command at the command line, then press *Enter*:

```
ipython RollDieDynamic.py 6000 1
```

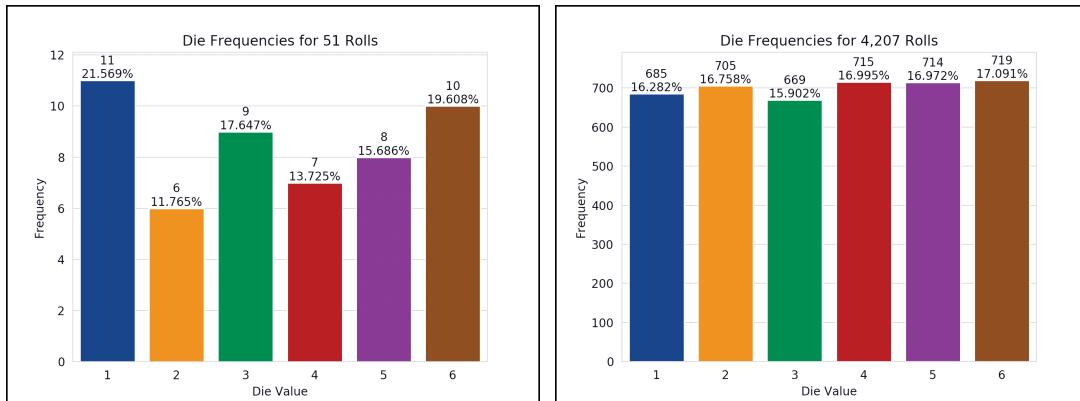
The script displays a window, showing the visualization. The numbers 6000 and 1 tell this script the number of times to roll dice and how many dice to roll each time. In this case, we’ll update the chart 6000 times for 1 die at a time.

For a six-sided die, the values 1 through 6 should each occur with “equal likelihood”—the probability of each is $1/6^{\text{th}}$ or about 16.667%. If we roll a die 6000 times, we’d expect about 1000 of each face. Like coin tossing, die rolling is *random*, so there could be some faces with fewer than 1000, some with 1000 and some with more than 1000. We took the screen captures on the next page during the script’s execution. This script uses randomly generated die values, so your results will differ. Experiment with the script by changing the value 1 to 100, 1000 and 10000. Notice that as the number of die rolls gets larger, the frequencies zero in on 16.667%. This is a phenomenon of the “Law of Large Numbers.”

Creating Scripts

Typically, you create your Python source code in an editor that enables you to type text. Using the editor, you type a program, make any necessary corrections and save it to your computer. **Integrated development environments (IDEs)** provide tools that support the entire software-development process, such as editors, debuggers for locating **logic errors** that cause programs to execute incorrectly and more. Some popular Python IDEs include Spyder (which comes with Anaconda), PyCharm and Visual Studio Code.

Roll the dice 6000 times and roll 1 die each time:
`ipython RollDieDynamic.py 6000 1`



Problems That May Occur at Execution Time

Programs often do not work on the first try. For example, an executing program might try to divide by zero (an illegal operation in Python). This would cause the program to display an error message. If this occurred in a script, you'd return to the editor, make the necessary corrections and re-execute the script to determine whether the corrections fixed the problem(s).

Errors such as division by zero occur as a program runs, so they're called **runtime errors** or **execution-time errors**. **Fatal runtime errors** cause programs to terminate immediately without having successfully performed their jobs. **Non-fatal runtime errors** allow programs to run to completion, often producing incorrect results.



Self Check

1 *(Discussion)* When the example in this section finishes all 6000 rolls, does the chart show that the die faces appeared *about* 1000 times each?

Answer: Most likely, yes. This example is based on random-number generation, so the results may vary. Because of this randomness, most of the counts will be a little more than 1000 or a little less.

2 *(Discussion)* Run the example in this section again. Do the faces appear the same number of times as they did in the previous execution?

Answer: Probably not. This example uses random-number generation, so successive executions likely will produce different results. In Chapter 4, we'll show how to force Python to produce the same sequence of random numbers. This is important for *reproducibility*—a crucial data-science topic you'll investigate in the chapter exercises and throughout the book. You'll want other data scientists to be able to reproduce your results. Also, you'll want to be able to reproduce your own experimental results. This is helpful when you find and fix an error in your program and want to make sure that you've corrected it properly.

1.10.3 Writing and Executing Code in a Jupyter Notebook

The Anaconda Python Distribution that you installed in the Before You Begin section comes with the **Jupyter Notebook**—an interactive, browser-based environment in which

you can write and execute code and intermix the code with text, images and video. Jupyter Notebooks are broadly used in the data-science community in particular and the broader scientific community in general. They're the preferred means of doing Python-based data analytics studies and *reproducibly* communicating their results. The Jupyter Notebook environment actually supports many programming languages.

For your convenience, all of the book's source code also is provided in Jupyter Notebooks that you can simply load and execute. In this section, you'll use the **JupyterLab** interface, which enables you to manage your notebook files and other files that your notebooks use (like images and videos). As you'll see, JupyterLab also makes it convenient to write code, execute it, see the results, modify the code and execute it again.

You'll see that coding in a Jupyter Notebook is similar to working with IPython—in fact, Jupyter Notebooks use IPython by default. In this section, you'll create a notebook, add the code from Section 1.10.1 to it and execute that code.

Opening JupyterLab in Your Browser

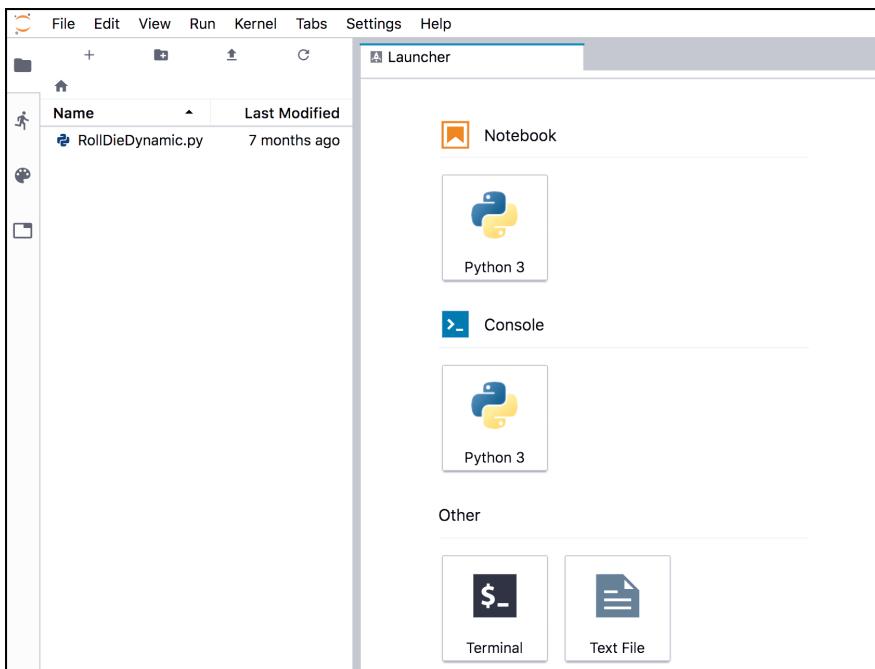
To open JupyterLab, change to the ch01 examples folder in your Terminal, shell or Anaconda Command Prompt (as in Section 1.10.2), type the following command, then press *Enter* (or *Return*):

```
jupyter lab
```

This executes the Jupyter Notebook server on your computer and opens JupyterLab in your default web browser, showing the ch01 folder's contents in the **File Browser** tab



at the left side of the JupyterLab interface:



The Jupyter Notebooks server enables you to load and run Jupyter Notebooks in your web browser. From the JupyterLab **Files** tab, you can double-click files to open them in the right side of the window where the **Launcher** tab is currently displayed. Each file you open appears as a separate tab in this part of the window. If you accidentally close your browser, you can reopen JupyterLab by entering the following address in your web browser

`http://localhost:8888/lab`

Creating a New Jupyter Notebook

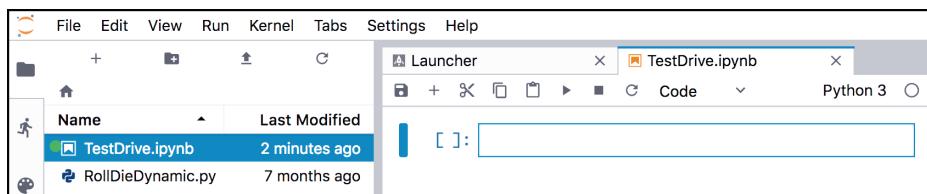
In the **Launcher** tab under **Notebook**, click the **Python 3** button to create a new Jupyter Notebook named `Untitled.ipynb` in which you can enter and execute Python 3 code. The file extension `.ipynb` is short for IPython Notebook—the original name of the Jupyter Notebook.

Renaming the Notebook

Rename `Untitled.ipynb` as `TestDrive.ipynb`:

1. Right-click the `Untitled.ipynb` tab and select **Rename Notebook....**
2. Change the name to `TestDrive.ipynb` and click **RENAME**.

The top of JupyterLab should now appear as follows:

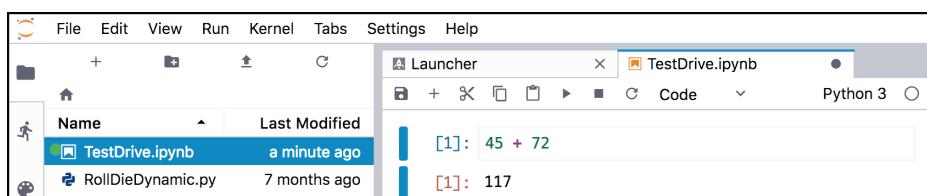


Evaluating an Expression

The unit of work in a notebook is a **cell** in which you can enter code snippets. By default, a new notebook contains one cell—the rectangle in the `TestDrive.ipynb` notebook—but you can add more. To the cell's left, the notation `[]:` is where the Jupyter Notebook will display the cell's snippet number *after* you execute the cell. Click in the cell, then type the expression

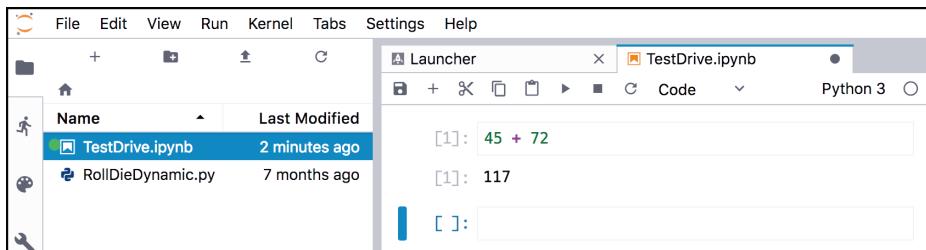
`45 + 72`

To execute the current cell's code, type *Ctrl + Enter* (or *control + Enter*). JupyterLab executes the code in IPython, then displays the results below the cell:



Adding and Executing Another Cell

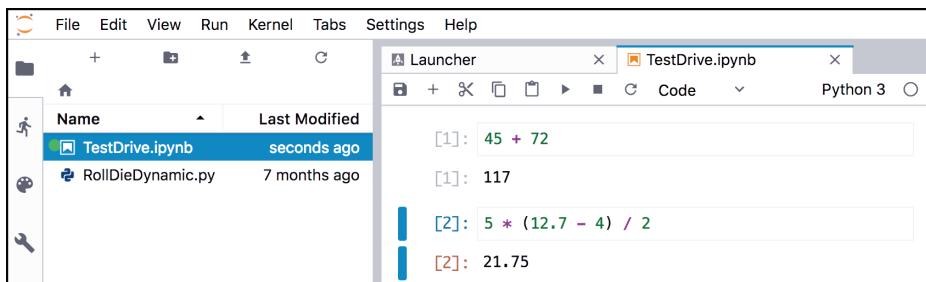
Let's evaluate a more complex expression. First, click the **+** button in the toolbar above the notebook's first cell—this adds a new cell below the current one:



Click in the new cell, then type the expression

$5 * (12.7 - 4) / 2$

and execute the cell by typing *Ctrl + Enter* (or *control + Enter*):



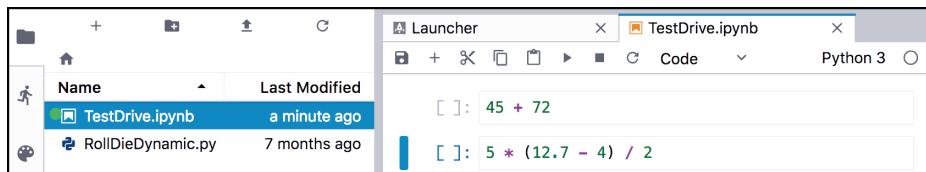
Saving the Notebook

If your notebook has unsaved changes, the X in the notebook's tab will change to ●. To save the notebook, select the **File** menu in JupyterLab (not at the top of your browser's window), then select **Save Notebook**.

Notebooks Provided with Each Chapter's Examples

For your convenience, each chapter's examples also are provided as ready-to-execute notebooks without their outputs. This enables you to work through them snippet-by-snippet and see the outputs appear as you execute each snippet.

So that we can show you how to load an existing notebook and execute its cells, let's reset the `TestDrive.ipynb` notebook to remove its output and snippet numbers. This will return it to a state like the notebooks we provide for the subsequent chapters' examples. From the **Kernel** menu select **Restart Kernel** and **Clear All Outputs...**, then click the **RESTART** button. The preceding command also is helpful whenever you wish to re-execute a notebook's snippets. The notebook should now appear as follows:



From the **File** menu, select **Save Notebook**, then click the `TestDrive.ipynb` tab's **X** button to close the notebook.

Opening and Executing an Existing Notebook

When you launch JupyterLab from a given chapter's examples folder, you'll be able to open notebooks from that folder or any of its subfolders. Once you locate a specific notebook, double-click it to open it. Open the `TestDrive.ipynb` notebook again now. Once a notebook is open, you can execute each cell individually, as you did earlier in this section, or you can execute the entire notebook at once. To do so, from the **Run** menu select **Run All Cells**. The notebook will execute the cells in order, displaying each cell's output below that cell.

Closing JupyterLab

When you're done with JupyterLab, you can close its browser tab, then in the Terminal, shell or Anaconda Command Prompt from which you launched JupyterLab, type *Ctrl + c* (or *control + c*) twice.

JupyterLab Tips

While working in JupyterLab, you might find these tips helpful:

- If you need to enter and execute many snippets, you can execute the current cell and add a new one below it by typing *Shift + Enter*, rather than *Ctrl + Enter* (or *control + Enter*).
- As you get into the later chapters, some of the snippets you'll enter in Jupyter Notebooks will contain many lines of code. To display line numbers within each cell, select **Show line numbers** from JupyterLab's **View** menu.

More Information on Working with JupyterLab

JupyterLab has many more features that you'll find helpful. We recommend that you read the Jupyter team's introduction to JupyterLab at:

<https://jupyterlab.readthedocs.io/en/stable/index.html>

For a quick overview, click **Overview** under **GETTING STARTED**. Also, under **USER GUIDE** read the introductions to **The JupyterLab Interface**, **Working with Files**, **Text Editor** and **Notebooks** for many additional features.



Self Check

1 (*True/False*) Jupyter Notebooks are the preferred means of doing Python-based data analytics studies and reproducibly communicating their results.

Answer: True.

2 (*Jupyter Notebook Session*) Ensure that JupyterLab is running, then open your `TestDrive.ipynb` notebook. Add and execute two more snippets that evaluate the expression $5 * (3 + 4)$ both with and without the parentheses. You should see the same results as in Section 1.10.1's Self Check Exercise 3.

Answer:

The screenshot shows a Jupyter Notebook interface. On the left is a file browser with two files: 'TestDrive.ipynb' (modified 'a minute ago') and 'RollDieDynamic.py' (modified '5 months ago'). The main area displays a series of code cells and their outputs:

- [1]: $45 + 72$
[1]: 117
- [2]: $5 * (12.7 - 4) / 2$
[2]: 21.75
- [3]: $5 * (3 + 4)$
[3]: 35
- [4]: $5 * 3 + 4$
[4]: 19

1.11 Internet and World Wide Web

In the late 1960s, ARPA—the Advanced Research Projects Agency of the United States Department of Defense—rolled out plans for networking the main computer systems of approximately a dozen ARPA-funded universities and research institutions. The computers were to be connected with communications lines operating at speeds on the order of 50,000 bits per second, a stunning rate at a time when most people (of the few who even had networking access) were connecting over telephone lines to computers at a rate of 110 bits per second. Academic research was about to take a giant leap forward. ARPA proceeded to implement what quickly became known as the ARPANET, the precursor to today's **Internet**. Today's fastest Internet speeds are on the order of billions of bits per second with trillion-bits-per-second (terabit) speeds already being tested!²⁸

Things worked out differently from the original plan. Although the ARPANET enabled researchers to network their computers, its main benefit proved to be the capability for quick and easy communication via what came to be known as electronic mail (e-mail). This is true even on today's Internet, with e-mail, instant messaging, file transfer and social media such as Snapchat, Instagram, Facebook and Twitter enabling billions of people worldwide to communicate quickly and easily.

The protocol (set of rules) for communicating over the ARPANET became known as the **Transmission Control Protocol (TCP)**. TCP ensured that messages, consisting of sequentially numbered pieces called *packets*, were properly delivered from sender to receiver, arrived intact and were assembled in the correct order.

1.11.1 Internet: A Network of Networks

In parallel with the early evolution of the Internet, organizations worldwide were implementing their own networks for both intra-organization (that is, within an organization) and inter-organization (that is, between organizations) communication. A huge variety of networking hardware and software appeared. One challenge was to enable these different

28. <https://testinternetspeed.org/blog/bt-testing-1-4-terabit-internet-connections/>.

networks to communicate with each other. ARPA accomplished this by developing the **Internet Protocol (IP)**, which created a true “network of networks,” the current architecture of the Internet. The combined set of protocols is now called **TCP/IP**. Each Internet-connected device has an **IP address**—a unique numerical identifier used by devices communicating via TCP/IP to locate one another on the Internet.

Businesses rapidly realized that by using the Internet, they could improve their operations and offer new and better services to their clients. Companies started spending large amounts of money to develop and enhance their Internet presence. This generated fierce competition among communications carriers and hardware and software suppliers to meet the increased infrastructure demand. As a result, **bandwidth**—the information-carrying capacity of communications lines—on the Internet has increased tremendously, while hardware costs have plummeted.

1.11.2 World Wide Web: Making the Internet User-Friendly

The **World Wide Web** (simply called “the web”) is a collection of hardware and software associated with the Internet that allows computer users to locate and view documents (with various combinations of text, graphics, animations, audios and videos) on almost any subject. In 1989, Tim Berners-Lee of CERN (the European Organization for Nuclear Research) began developing **HyperText Markup Language (HTML)**—the technology for sharing information via “hyperlinked” text documents. He also wrote communication protocols such as **HyperText Transfer Protocol (HTTP)** to form the backbone of his new hypertext information system, which he referred to as the World Wide Web.

In 1994, Berners-Lee founded the **World Wide Web Consortium (W3C)**, <https://www.w3.org>), devoted to developing web technologies. One of the W3C’s primary goals is to make the web universally accessible to everyone regardless of disabilities, language or culture.

1.11.3 The Cloud

More and more computing today is done “in the cloud”—that is, distributed across the Internet worldwide. The apps you use daily are heavily dependent on various **cloud-based services** that use massive clusters of computing resources (computers, processors, memory, disk drives, etc.) and databases that communicate over the Internet with each other and the apps you use. A service that provides access to itself over the Internet is known as a **web service**. As you’ll see, using cloud-based services in Python often is as simple as creating a software object and interacting with it. That object then uses web services that connect to the cloud on your behalf.

Throughout the Chapters 12–17 examples and exercises, you’ll work with many cloud-based services:

- In Chapters 13 and 17, you’ll use Twitter’s web services (via the Python library Tweepy) to get information about specific Twitter users, search for tweets from the last seven days and to receive streams of tweets as they occur—that is, in **real time**.
- In Chapters 12 and 13, you’ll use the Python library TextBlob to translate text between languages. Behind the scenes, TextBlob uses the Google Translate web service to perform those translations.

- In Chapter 14, you’ll use the IBM Watson’s Text to Speech, Speech to Text and Translate services. You’ll implement a traveler’s assistant translation app that enables you to speak a question in English, transcribes the speech to text, translates the text to Spanish and speaks the Spanish text. The app then allows you to speak a Spanish response (in case you don’t speak Spanish, we provide an audio file you can use), transcribes the speech to text, translates the text to English and speaks the English response. Via IBM Watson demos, you’ll also experiment with many other Watson cloud-based services in Chapter 14.
- In Chapter 17, you’ll work with Microsoft Azure’s HDInsight service and other Azure web services as you learn to implement big-data applications using Apache Hadoop and Spark. Azure is Microsoft’s set of cloud-based services.
- In Chapter 17, you’ll use the Dweet.io web service to simulate an Internet-connected thermostat that publishes temperature readings online. You’ll also use a web-based service to create a “dashboard” that visualizes the temperature readings over time and warns you if the temperature gets too low or too high.
- In Chapter 17, you’ll use a web-based dashboard to visualize a simulated stream of live sensor data from the PubNub web service. You’ll also create a Python app that visualizes a PubNub simulated stream of live stock-price changes.
- In multiple exercises, you’ll research, explore and use Wikipedia web services.

In most cases, you’ll create Python objects that interact with web services on your behalf, hiding the details of how to access these services over the Internet.

Mashups

The applications-development methodology of *mashups* enables you to rapidly develop powerful software applications by combining (often free) complementary web services and other forms of information feeds—as you’ll do in our IBM Watson traveler’s assistant translation app. One of the first mashups combined the real-estate listings provided by <http://www.craigslist.org> with the mapping capabilities of Google Maps to offer maps that showed the locations of homes for sale or rent in a given area.

ProgrammableWeb (<http://www.programmableweb.com/>) provides a directory of over 20,750 web services and almost 8,000 mashups. They also provide how-to guides and sample code for working with web services and creating your own mashups. According to their website, some of the most widely used web services are Facebook, Google Maps, Twitter and YouTube.

1.11.4 Internet of Things

The Internet is no longer just a network of *computers*—it’s an **Internet of Things (IoT)**. A *thing* is any object with an IP address and the ability to send, and in some cases receive, data automatically over the Internet. Such *things* include:

- a car with a transponder for paying tolls,
- monitors for parking-space availability in a garage,
- a heart monitor implanted in a human,
- water quality monitors,

- a smart meter that reports energy usage,
- radiation detectors,
- item trackers in a warehouse,
- mobile apps that can track your movement and location,
- smart thermostats that adjust room temperatures based on weather forecasts and activity in the home, and
- intelligent home appliances.

According to statista.com, there are already over 23 billion IoT devices in use today, and there could be over 75 billion IoT devices in 2025.²⁹



Self Check for Section 1.11

1 (Fill-In) The _____ was the precursor to today's Internet.

Answer: ARPANET.

2 (Fill-In) The _____ (simply called “the web”) is a collection of hardware and software associated with the Internet that allows computer users to locate and view documents (with various combinations of text, graphics, animations, audios and videos).

Answer: World Wide Web.

3 (Fill-In) In the Internet of Things (IoT), a *thing* is any object with a(n) _____ and the ability to send, and in some cases receive, data automatically over the Internet.

Answer: IP address.

1.12 Software Technologies

As you learn about and work in software development, you'll frequently encounter the following buzzwords:

- **Refactoring:** Reworking programs to make them clearer and easier to maintain while preserving their correctness and functionality. Many IDEs contain built-in *refactoring tools* to do major portions of the reworking automatically.
- **Design patterns:** Proven architectures for constructing flexible and maintainable object-oriented software. The field of design patterns tries to enumerate those recurring patterns, encouraging software designers to *reuse* them to develop better-quality software using less time, money and effort.
- **Cloud computing:** You can use software and data stored in the “cloud”—i.e., accessed on remote computers (or servers) via the Internet and available on demand—rather than having it stored locally on your desktop, notebook computer or mobile device. This allows you to increase or decrease computing resources to meet your needs at any given time, which is more cost effective than purchasing hardware to provide enough storage and processing power to meet occasional peak demands. Cloud computing also saves money by shifting to the

29. <https://www.statista.com/statistics/471264/iot-number-of-connected-devices-world-wide/>.

service provider the burden of managing these apps (such as installing and upgrading the software, security, backups and disaster recovery).

- **Software Development Kits (SDKs)**—The tools and documentation that developers use to program applications. For example, in Chapter 14, you’ll use the Watson Developer Cloud Python SDK to interact with IBM Watson services from a Python application.



Self Check

I (Fill-In) _____ is the process of reworking programs to make them clearer and easier to maintain while preserving their correctness and functionality.

Answer: refactoring.

1.13 How Big Is Big Data?

For computer scientists and data scientists, data is now as important as writing programs. According to IBM, approximately 2.5 quintillion bytes (*2.5 exabytes*) of data are created daily,³⁰ and 90% of the world’s data was created in the last two years.³¹ According to IDC, the global data supply will reach 175 *zettabytes* (equal to 175 trillion gigabytes or 175 billion terabytes) annually by 2025.³² Consider the following examples of various popular data measures.

Megabytes (MB)

One megabyte is about one million (actually 2^{20}) bytes. Many of the files we use on a daily basis require one or more MBs of storage. Some examples include:

- MP3 audio files—High-quality MP3s range from 1 to 2.4 MB per minute.³³
- Photos—JPEG format photos taken on a digital camera can require about 8 to 10 MB per photo.
- Video—Smartphone cameras can record video at various resolutions. Each minute of video can require many megabytes of storage. For example, on one of our iPhones, the **Camera** settings app reports that 1080p video at 30 frames-per-second (FPS) requires 130 MB/minute and 4K video at 30 FPS requires 350 MB/minute.

Gigabytes (GB)

One gigabyte is about 1000 megabytes (actually 2^{30} bytes). A dual-layer DVD can store up to 8.5 GB³⁴, which translates to:

- as much as 141 hours of MP3 audio,
- approximately 1000 photos from a 16-megapixel camera,

30. <https://www.ibm.com/blogs/watson/2016/06/welcome-to-the-world-of-a-i/>.

31. <https://public.dhe.ibm.com/common/ssi/ecm/wr/en/wrl12345usen/watson-customer-engagement-watson-marketing-wr-other-papers-and-reports-wrl12345usen-20170719.pdf>.

32. <https://www.networkworld.com/article/3325397/storage/idc-expect-175-zettabytes-of-data-worldwide-by-2025.html>.

33. <https://www.audiomountain.com/tech/audio-file-size.html>.

34. <https://en.wikipedia.org/wiki/DVD>.

- approximately 7.7 minutes of 1080p video at 30 FPS, or
- approximately 2.85 minutes of 4K video at 30 FPS.

The current highest-capacity Ultra HD Blu-ray discs can store up to 100 GB of video.³⁵ Streaming a 4K movie can use between 7 and 10 GB per hour (highly compressed).

Terabytes (TB)

One terabyte is about 1000 gigabytes (actually 2^{40} bytes). Recent disk drives for desktop computers come in sizes up to 15 TB,³⁶ which is equivalent to:

- approximately 28 years of MP3 audio,
- approximately 1.68 million photos from a 16-megapixel camera,
- approximately 226 hours of 1080p video at 30 FPS and
- approximately 84 hours of 4K video at 30 FPS.

Nimbus Data now has the largest solid-state drive (SSD) at 100 TB, which can store 6.67 times the 15-TB examples of audio, photos and video listed above.³⁷

Petabytes, Exabytes and Zettabytes

There are nearly four billion people online creating about 2.5 quintillion bytes of data each day³⁸—that's 2500 petabytes (each petabyte is about 1000 terabytes) or 2.5 exabytes (each exabyte is about 1000 petabytes). According to a March 2016 *Analytics Week* article, within five years there will be over 50 billion devices connected to the Internet (most of them through the Internet of Things, which we discuss in Sections 1.11.4 and 17.8) and by 2020 we'll be producing 1.7 megabytes of new data every second *for every person on the planet*.³⁹ At today's numbers (approximately 7.7 billion people⁴⁰), that's about

- 13 petabytes of new data per second,
- 780 petabytes per minute,
- 46,800 petabytes (46.8 exabytes) per hour and
- 1,123 exabytes per day—that's 1.123 zettabytes (ZB) per day (each zettabyte is about 1000 exabytes).

That's the equivalent of over 5.5 million hours (over 600 years) of 4K video every day or approximately 116 billion photos every day!

Additional Big-Data Stats

For a real-time sense of big data, check out <https://www.internetlivestats.com>, with various statistics, including the numbers so far today of

- Google searches.

35. https://en.wikipedia.org/wiki/Ultra_HD_Blu-ray.

36. <https://www.zdnet.com/article/worlds-biggest-hard-drive-meet-western-digital-s-15tb-monster/>.

37. <https://www.cinema5d.com/nimbus-data-100tb-ssd-worlds-largest-ssd/>.

38. <https://public.dhe.ibm.com/common/ssi/ecm/wr/en/wrl12345usen/watson-customer-engagement-watson-marketing-wr-other-papers-and-reports-wrl12345usen-20170719.pdf>.

39. <https://analyticsweek.com/content/big-data-facts/>.

40. https://en.wikipedia.org/wiki/World_population.

- Tweets.
- Videos viewed on YouTube.
- Photos uploaded on Instagram.

You can click each statistic to drill down for more information. For instance, they say over 250 *billion* tweets have been sent in 2018.

Some other interesting big-data facts:

- Every hour, YouTube users upload 24,000 hours of video, and almost 1 billion hours of video are watched on YouTube every day.⁴¹
- Every second, there are 51,773 GBs (or 51.773 TBs) of Internet traffic, 7894 tweets sent, 64,332 Google searches and 72,029 YouTube videos viewed.⁴²
- On Facebook each day there are 800 million “likes,”⁴³ 60 million emojis are sent,⁴⁴ and there are over two billion searches of the more than 2.5 trillion Facebook posts since the site’s inception.⁴⁵
- In June 2017, Will Marshall, CEO of Planet, said the company has 142 satellites that image the whole planet’s land mass once per day. They add one million images and seven TBs of new data each day. Together with their partners, they’re using machine learning on that data to improve crop yields, see how many ships are in a given port and track deforestation. With respect to Amazon deforestation, he said: “Used to be we’d wake up after a few years and there’s a big hole in the Amazon. Now we can literally count every tree on the planet every day.”⁴⁶

Domo, Inc. has a nice infographic called “Data Never Sleeps 6.0” showing how much data is generated *every minute*, including:⁴⁷

- 473,400 tweets sent.
- 2,083,333 Snapchat photos shared.
- 97,222 hours of Netflix video viewed.
- 12,986,111 million text messages sent.
- 49,380 Instagram posts.
- 176,220 Skype calls.
- 750,000 Spotify songs streamed.
- 3,877,140 Google searches.
- 4,333,560 YouTube videos watched.

41. <https://www.brandwatch.com/blog/youtube-stats/>.

42. <http://www.internetlivestats.com/one-second>.

43. <https://newsroom.fb.com/news/2017/06/two-billion-people-coming-together-on-facebook>.

44. <https://mashable.com/2017/07/17/facebook-world-emoji-day/>.

45. <https://techcrunch.com/2016/07/27/facebook-will-make-you-talk/>.

46. <https://www.bloomberg.com/news/videos/2017-06-30/learning-from-planet-s-shoe-boxed-sized-satellites-video>, June 30, 2017.

47. <https://www.domo.com/learn/data-never-sleeps-6>.

Computing Power Over the Years

Data is getting more massive and so is the computing power for processing it. The performance of today's processors is often measured in terms of **FLOPS (floating-point operations per second)**. In the early to mid-1990s, the fastest supercomputer speeds were measured in gigaflops (10^9 FLOPS). By the late 1990s, Intel produced the first teraflop (10^{12} FLOPS) supercomputers. In the early-to-mid 2000s, speeds reached hundreds of teraflops, then in 2008, IBM released the first petaflop (10^{15} FLOPS) supercomputer. Currently, the fastest supercomputer—the IBM Summit, located at the Department of Energy's (DOE) Oak Ridge National Laboratory (ORNL)—is capable of 122.3 petaflops.⁴⁸

Distributed computing can link thousands of personal computers via the Internet to produce even more FLOPS. In late 2016, the Folding@home network—a distributed network in which people volunteer their personal computers' resources for use in disease research and drug design⁴⁹—was capable of over 100 petaflops.⁵⁰ Companies like IBM are now working toward supercomputers capable of exaflops (10^{18} FLOPS).⁵¹

The **quantum computers** now under development theoretically could operate at 18,000,000,000,000,000,000 times the speed of today's "conventional computers"⁵². This number is so extraordinary that in one second, a quantum computer theoretically could do staggeringly more calculations than the total that have been done by all computers since the world's first computer appeared. This almost unimaginable computing power could wreak havoc with blockchain-based cryptocurrencies like Bitcoin. Engineers are already rethinking blockchain to prepare for such massive increases in computing power.⁵³

The history of supercomputing power is that it eventually works its way down from research labs, where extraordinary amounts of money have been spent to achieve those performance numbers, into "reasonably priced" commercial computer systems and even desktop computers, laptops, tablets and smartphones.

Computing power's cost continues to decline, especially with cloud computing. People used to ask the question, "How much computing power do I need on my system to deal with my *peak* processing needs?" Today, that thinking has shifted to "Can I quickly carve out on the cloud what I need *temporarily* for my most demanding computing chores?" You pay for only what you use to accomplish a given task.

Processing the World's Data Requires Lots of Electricity

Data from the world's Internet-connected devices is exploding, and processing that data requires tremendous amounts of energy. According to a recent article, energy use for processing data in 2015 was growing at 20% per year and consuming approximately three to five percent of the world's power. The article says that total data-processing power consumption could reach 20% by 2025.⁵⁴

48. <https://en.wikipedia.org/wiki/FLOPS>.

49. <https://en.wikipedia.org/wiki/Folding@home>.

50. <https://en.wikipedia.org/wiki/FLOPS>.

51. <https://www.ibm.com/blogs/research/2017/06/supercomputing-weather-model-exascale/>.

52. <https://medium.com/@n.biedrzycki/only-god-can-count-that-fast-the-world-of-quantum-computing-406a0a91fcf4>.

53. <https://singularityhub.com/2017/11/05/is-quantum-computing-an-existential-threat-to-blockchain-technology/>.

54. <https://www.theguardian.com/environment/2017/dec/11/tsunami-of-data-could-consume-fifth-global-electricity-by-2025>.

Another enormous electricity consumer is the blockchain-based cryptocurrency Bitcoin. Processing just one Bitcoin transaction uses approximately the same amount of energy as powering the average American home for a week. The energy use comes from the process Bitcoin “miners” use to prove that transaction data is valid.⁵⁵

According to some estimates, a year of Bitcoin transactions consumes more energy than many countries.⁵⁶ Together, Bitcoin and Ethereum (another popular blockchain-based platform and cryptocurrency) consume more energy per year than Israel and almost as much as Greece.⁵⁷

Morgan Stanley predicted in 2018 that “the electricity consumption required to create cryptocurrencies this year could actually outpace the firm’s projected global electric vehicle demand—in 2025.”⁵⁸ This situation is unsustainable, especially given the huge interest in blockchain-based applications, even beyond the cryptocurrency explosion. The blockchain community is working on fixes.^{59,60}

Big-Data Opportunities

The big-data explosion is likely to continue exponentially for years to come. With 50 billion computing devices on the horizon, we can only imagine how many more there will be over the next few decades. It’s crucial for businesses, governments, the military, and even individuals to get a handle on all this data.

It’s interesting that some of the best writings about big data, data science, artificial intelligence and more are coming out of distinguished business organizations, such as J.P. Morgan, McKinsey and more. Big data’s appeal to big business is undeniable given the rapidly accelerating accomplishments. Many companies are making significant investments and getting valuable results through technologies in this book, such as big data, machine learning, deep learning, and natural-language processing. This is forcing competitors to invest as well, rapidly increasing the need for computing professionals with data-science and computer science experience. This growth is likely to continue for many years.



Self Check

1 (Fill-In) Today’s processor performance is often measured in terms of _____.

Answer: FLOPS (floating-point operations per second).

2 (Fill-In) The technology that could wreak havoc with blockchain-based cryptocurrencies, like Bitcoin, and other blockchain-based technologies is _____.

Answer: quantum computers.

3 (True/False) With cloud computing you pay a fixed price for cloud services regardless of how much you use those services?

Answer: False. A key cloud-computing benefit is that you pay for only what you use to accomplish a given task.

55. https://motherboard.vice.com/en_us/article/ywbbpm/bitcoin-mining-electricity-consumption-ethereum-energy-climate-change.

56. <https://digiconomist.net/bitcoin-energy-consumption>.

57. <https://digiconomist.net/ethereum-energy-consumption>.

58. <https://www.morganstanley.com/ideas/cryptocurrencies-global-utilities>.

59. <https://www.technologyreview.com/s/609480/bitcoin-uses-massive-amounts-of-energy-but-theres-a-plan-to-fix-it/>.

60. <http://mashable.com/2017/12/01/bitcoin-energy/>.

1.13.1 Big Data Analytics

Data analytics is a mature and well-developed academic and professional discipline. The term “data analysis” was coined in 1962,⁶¹ though people have been analyzing data using statistics for thousands of years going back to the ancient Egyptians.⁶² Big data analytics is a more recent phenomenon—the term “big data” was coined around 2000.⁶³

Consider four of the V’s of big data^{64,65}:

1. Volume—the amount of data the world is producing is growing exponentially.
2. Velocity—the speed at which that data is being produced, the speed at which it moves through organizations and the speed at which data changes are growing quickly.^{66,67,68}
3. Variety—data used to be alphanumeric (that is, consisting of alphabetic characters, digits, punctuation and some special characters)—today it also includes images, audios, videos and data from an exploding number of Internet of Things sensors in our homes, businesses, vehicles, cities and more.
4. Veracity—the validity of the data—is it complete and accurate? Can we trust that data when making crucial decisions? Is it real?

Most data is now being created digitally in a *variety* of types, in extraordinary *volumes* and moving at astonishing *velocities*. Moore’s Law and related observations have enabled us to store data economically and to process and move it faster—and all at rates growing exponentially over time. Digital data storage has become so vast in capacity, cheap and small that we can now conveniently and economically retain *all* the digital data we’re creating.⁶⁹ That’s big data.

The following Richard W. Hamming quote—although from 1962—sets the tone for the rest of this book:

“The purpose of computing is insight, not numbers.”⁷⁰

Data science is producing new, deeper, subtler and more valuable insights at a remarkable pace. It’s truly making a difference. Big data analytics is an integral part of the answer. We address big data infrastructure in Chapter 17 with hands-on case studies on NoSQL data-

61. <https://www.forbes.com/sites/gilpress/2013/05/28/a-very-short-history-of-data-science/>.
62. <https://www.flydata.com/blog/a-brief-history-of-data-analysis/>.
63. <https://bits.blogs.nytimes.com/2013/02/01/the-origins-of-big-data-an-etymological-detective-story/>.
64. <https://www.ibmbigdatahub.com/infographic/four-vs-big-data>.
65. There are lots of articles and papers that add many other “V-words” to this list.
66. <https://www.zdnet.com/article/volume-velocity-and-variety-understanding-the-three-vs-of-big-data/>.
67. <https://whatis.techtarget.com/definition/3Vs>.
68. <https://www.forbes.com/sites/brentdykes/2017/06/28/big-data-forget-volume-and-variety-focus-on-velocity>.
69. <http://www.lesk.com/mlesk/ksg97/ksg.html>. [The following article pointed us to this Michael Lesk article: <https://www.forbes.com/sites/gilpress/2013/05/28/a-very-short-history-of-data-science/>.]
70. Hamming, R. W., *Numerical Methods for Scientists and Engineers* (New York, NY., McGraw Hill, 1962). [The following article pointed us to Hamming’s book and his quote that we cited: <https://www.forbes.com/sites/gilpress/2013/05/28/a-very-short-history-of-data-science/>.]

bases, Hadoop MapReduce programming, Spark, real-time Internet of Things (IoT) stream programming and more.

To get a sense of big data's scope in industry, government and academia, check out the high-resolution graphic.⁷¹ You can click to zoom for easier readability:

http://mattturck.com/wp-content/uploads/2018/07/Matt_Turck_FirstMark_Big_Data_Landscape_2018_Final.png

I.13.2 Data Science and Big Data Are Making a Difference: Use Cases

The data-science field is growing rapidly because it's producing significant results that are making a difference. We enumerate data-science and big data use cases in the following table. We expect that the use cases and our examples, exercises and projects will inspire interesting term projects, directed-study projects, capstone-course projects and thesis research. Big-data analytics has resulted in improved profits, better customer relations, and even sports teams winning more games and championships while spending less on players.^{72,73,74}

Data-science use cases		
anomaly detection	customer churn	facial recognition
assisting people with disabilities	customer experience	fitness tracking
auto-insurance risk prediction	customer retention	fraud detection
automated closed captioning	customer satisfaction	game playing
automated image captions	customer service	genomics and healthcare
automated investing	customer service agents	Geographic Information Systems (GIS)
autonomous ships	customized diets	GPS Systems
brain mapping	cybersecurity	health outcome improvement
caller identification	data mining	hospital readmission reduction
cancer diagnosis/treatment	data visualization	human genome sequencing
carbon emissions reduction	detecting new viruses	identity-theft prevention
classifying handwriting	diagnosing breast cancer	immunotherapy
computer vision	diagnosing heart disease	insurance pricing
credit scoring	diagnostic medicine	intelligent assistants
crime: predicting locations	disaster-victim identification	Internet of Things (IoT) and medical device monitoring
crime: predicting recidivism	drones	Internet of Things and weather forecasting
crime: predictive policing	dynamic driving routes	inventory control
crime: prevention	dynamic pricing	language translation
CRISPR gene editing	electronic health records	
crop-yield improvement	emotion detection	
	energy-consumption reduction	

71. Turck, M., and J. Hao, "Great Power, Great Responsibility: The 2018 Big Data & AI Landscape," <http://mattturck.com/bigdata2018/>.

72. Sawchik, T., *Big Data Baseball: Math, Miracles, and the End of a 20-Year Losing Streak* (New York, Flat Iron Books, 2015).

73. Ayres, I., *Super Crunchers* (Bantam Books, 2007), pp. 7–10.

74. Lewis, M., *Moneyball: The Art of Winning an Unfair Game* (W. W. Norton & Company, 2004).

Data-science use cases

location-based services	predicting weather-sensitive product sales	smart thermostats
loyalty programs	predictive analytics	smart traffic control
malware detection	preventative medicine	social analytics
mapping	preventing disease outbreaks	social graph analysis
marketing	reading sign language	spam detection
marketing analytics	real-estate valuation	spatial data analysis
music generation	recommendation systems	sports recruiting and coaching
natural-language translation	reducing overbooking	stock market forecasting
new pharmaceuticals	ride sharing	student performance assessment
opioid abuse prevention	risk minimization	summarizing text
personal assistants	robo financial advisors	telemedicine
personalized medicine	security enhancements	terrorist attack prevention
personalized shopping	self-driving cars	theft prevention
phishing elimination	sentiment analysis	travel recommendations
pollution reduction	sharing economy	trend spotting
precision medicine	similarity detection	visual product search
predicting cancer survival	smart cities	voice recognition
predicting disease outbreaks	smart homes	voice search
predicting health outcomes	smart meters	weather forecasting
predicting student enrollments		

1.14 Case Study—A Big-Data Mobile Application

Google's Waze GPS navigation app, with its 90 million monthly active users,⁷⁵ is one of the most widely used big-data apps. Early GPS navigation devices and apps relied on static maps and GPS coordinates to determine the best route to your destination. They could not adjust dynamically to changing traffic situations.

Waze processes massive amounts of **crowdsourced data**—that is, the data that's continuously supplied by their users and their users' devices worldwide. They analyze this data as it arrives to determine the best route to get you to your destination in the least amount of time. To accomplish this, Waze relies on your smartphone's Internet connection. The app automatically sends location updates to their servers (assuming you allow it to). They use that data to dynamically re-route you based on current traffic conditions and to tune their maps. Users report other information, such as roadblocks, construction, obstacles, vehicles in breakdown lanes, police locations, gas prices and more. Waze then alerts other drivers in those locations.

Waze uses many technologies to provide its services. We're not privy to how Waze is implemented, but we infer below a list of technologies they probably use. You'll see many of these in Chapters 12–17. For example,

- Most apps created today use at least some open-source software. You'll take advantage of many open-source libraries and tools throughout this book.

75. <https://www.waze.com/brands/drivers/>.

- Waze communicates information over the Internet between their servers and their users' mobile devices. Today, such data typically is transmitted in JSON (JavaScript Object Notation) format, which we'll introduce in Chapter 9 and use in subsequent chapters. Often the JSON data will be hidden from you by the libraries you use.
- Waze uses speech synthesis to speak driving directions and alerts to you, and speech recognition to understand your spoken commands. We use IBM Watson's speech-synthesis and speech-recognition capabilities in Chapter 14.
- Once Waze converts a spoken natural-language command to text, it must determine the correct action to perform, which requires natural language processing (NLP). We present NLP in Chapter 12 and use it in several subsequent chapters.
- Waze displays dynamically updated visualizations such as alerts and maps. Waze also enables you to interact with the maps by moving them or zooming in and out. We create dynamic visualizations with Matplotlib and Seaborn throughout the book, and we display interactive maps with Folium in Chapters 13 and 17.
- Waze uses your phone as a streaming Internet of Things (IoT) device. Each phone is a GPS sensor that continuously streams data over the Internet to Waze. In Chapter 17, we introduce IoT and work with simulated IoT streaming sensors.
- Waze receives IoT streams from millions of phones at once. It must process, store and analyze that data immediately to update your device's maps, to display and speak relevant alerts and possibly to update your driving directions. This requires massively parallel processing capabilities implemented with clusters of computers in the cloud. In Chapter 17, we'll introduce various big-data infrastructure technologies for receiving streaming data, storing that big data in appropriate databases and processing the data with software and hardware that provide massively parallel processing capabilities.
- Waze uses artificial-intelligence capabilities to perform the data-analysis tasks that enable it to predict the best routes based on the information it receives. In Chapters 15 and 16 we use machine learning and deep learning, respectively, to analyze massive amounts of data and make predictions based on that data.
- Waze probably stores its routing information in a graph database. Such databases can efficiently calculate shortest routes. We introduce graph databases, such as Neo4j, in Chapter 17. Exercise 17.7 asks you to solve the popular “six degrees of separation” problem with Neo4j.
- Many cars are now equipped with devices that enable them to “see” cars and obstacles around them. These are used, for example, to help implement automated braking systems and are a key part of self-driving car technology. Rather than relying on users to report obstacles and stopped cars on the side of the road, navigation apps could take advantage of cameras and other sensors by using deep-learning computer-vision techniques to analyze images “on the fly” and automatically report those items. We introduce deep learning for computer vision in Chapter 16.

1.15 Intro to Data Science: Artificial Intelligence—at the Intersection of CS and Data Science

When a baby first opens its eyes, does it “see” its parent’s faces? Does it understand any notion of what a face is—or even what a simple shape is? Babies must “learn” the world around them. That’s what artificial intelligence (AI) is doing today. It’s looking at massive amounts of data and learning from it. AI is being used to play games, implement a wide range of computer-vision applications, enable self-driving cars, enable robots to learn to perform new tasks, diagnose medical conditions, translate speech to other languages in near real time, create chatbots that can respond to arbitrary questions using massive databases of knowledge, and much more. Who’d have guessed just a few years ago that artificially intelligent self-driving cars would be allowed on our roads—or even become common? Yet, this is now a highly competitive area. The ultimate goal of all this learning is **artificial general intelligence**—an AI that can perform intelligence tasks as well as humans.

Artificial-Intelligence Milestones

Several artificial-intelligence milestones, in particular, captured people’s attention and imagination, made the general public start thinking that AI is real and made businesses think about commercializing AI:

- In a 1997 match between **IBM’s DeepBlue** computer system and chess Grandmaster Gary Kasparov, DeepBlue became the first computer to beat a reigning world chess champion under tournament conditions.⁷⁶ IBM loaded DeepBlue with hundreds of thousands of grandmaster chess games.⁷⁷ DeepBlue was capable of using *brute force* to evaluate up to 200 million moves per second!⁷⁸ This is big data at work. IBM received the Carnegie Mellon University Fredkin Prize, which in 1980 offered \$100,000 to the creators of the first computer to beat a world chess champion.⁷⁹
- In 2011, **IBM’s Watson** beat the two best human Jeopardy! players in a \$1 million match. Watson simultaneously used hundreds of language-analysis techniques to locate correct answers in 200 million pages of content (including all of Wikipedia) requiring four terabytes of storage.^{80,81} Watson was trained with machine learning and **reinforcement-learning techniques**.⁸² Chapter 16 discusses **machine-learning** and Chapter 17’s exercises introduce **reinforcement learning**.
- Go—a board game created in China thousands of years ago⁸³—is widely considered to be one of the most complex games ever invented with 10^{170} possible board configurations.⁸⁴ To give you a sense of how large a number that is, it’s

76. https://en.wikipedia.org/wiki/Deep_Blue_versus_Garry_Kasparov.

77. [https://en.wikipedia.org/wiki/Deep_Blue_\(chess_computer\)](https://en.wikipedia.org/wiki/Deep_Blue_(chess_computer)).

78. [https://en.wikipedia.org/wiki/Deep_Blue_\(chess_computer\)](https://en.wikipedia.org/wiki/Deep_Blue_(chess_computer)).

79. <https://articles.latimes.com/1997/jul/30/news/mn-17696>.

80. <https://www.techrepublic.com/article/ibm-watson-the-inside-story-of-how-the-jeopardy-winning-supercomputer-was-born-and-what-it-wants-to-do-next/>.

81. [https://en.wikipedia.org/wiki/Watson_\(computer\)](https://en.wikipedia.org/wiki/Watson_(computer)).

82. <https://www.aaai.org/Magazine/Watson/watson.php>, *AI Magazine*, Fall 2010.

83. <http://www.usgo.org/brief-history-go>.

84. <https://www.pbs.org/newshour/science/google-artificial-intelligence-beats-champion-at-worlds-most-complicated-board-game>.

believed that there are (only) between 10^{78} and 10^{87} atoms in the known universe!^{85,86} In 2015, **AlphaGo**—created by Google’s DeepMind group—used *deep learning with two neural networks to beat the European Go champion Fan Hui*. Go is considered to be a far more complex game than chess. Chapter 17 discusses neural networks and deep learning.

- More recently, Google generalized its AlphaGo AI to create **AlphaZero**—a game-playing AI that *teaches itself to play other games*. In December 2017, AlphaZero learned the rules of and taught itself to play chess in less than four hours using reinforcement learning. It then beat the world champion chess program, Stockfish 8, in a 100-game match—winning or drawing every game. After *training itself in Go for just eight hours*, AlphaZero was able to play Go vs. its AlphaGo predecessor, winning 60 of 100 games.⁸⁷ Chapter 17 discusses reinforcement learning.

A Personal Anecdote

When one of the authors, Harvey Deitel, was an undergraduate student at MIT in the mid-1960s, he took a graduate-level artificial-intelligence course with Marvin Minsky (to whom this book is dedicated), one of the founders of artificial intelligence (AI). Harvey:

Professor Minsky required a major term project. He told us to think about what intelligence is and to make a computer do something intelligent. Our grade in the course would be almost solely dependent on the project. No pressure!

I researched the standardized IQ tests that schools administer to help evaluate their students’ intelligence capabilities. Being a mathematician at heart, I decided to tackle the popular IQ-test problem of predicting the next number in a sequence of numbers of arbitrary length and complexity. I used interactive Lisp running on an early Digital Equipment Corporation PDP-1 and was able to get my sequence predictor running on some pretty complex stuff, handling challenges well beyond what I recalled seeing on IQ tests. Lisp’s ability to manipulate arbitrarily long lists recursively was exactly what I needed to meet the project’s requirements. Python offers recursion (Chapter 11) and generalized list processing (Chapter 5).

I tried the sequence predictor on many of my MIT classmates. They would make up number sequences and type them into my predictor. The PDP-1 would “think” for a while—often a long while—and almost always came up with the right answer.

Then I hit a snag. One of my classmates typed in the sequence 14, 23, 34 and 42. My predictor went to work on it, and the PDP-1 chugged away for a long time, failing to predict the next number. I couldn’t get it either. My classmate told me to think about it overnight, and he’d reveal the answer the next day, claiming that it was a simple sequence. My efforts were to no avail.

The following day he told me the next number was 57, but I didn’t understand why. So he told me to think about it overnight again, and the following day he

85. <https://www.universetoday.com/36302/atoms-in-the-universe/>.

86. https://en.wikipedia.org/wiki/Observable_universe#Matter_content.

87. <https://www.theguardian.com/technology/2017/dec/07/alphazero-google-deepmind-ai-beats-champion-program-teaching-itself-to-play-four-hours>.

said the next number was 125. That didn't help a bit—I was stumped. He said that the sequence was the numbers of the two-way crosstown streets of Manhattan. I cried, "foul," but he said it met my criterion of predicting the next number in a numerical sequence. My world view was mathematics—his was broader.

Over the years, I've tried that sequence on friends, relatives and professional colleagues. A few who either lived in Manhattan or spent time there got it right. My sequence predictor needed a lot more than just mathematical knowledge to handle problems like this, requiring (a possibly vast) world knowledge.

Watson and Big Data Open New Possibilities

When Paul and I started working on this Python book, we were immediately drawn to IBM's Watson using big data and artificial-intelligence techniques like natural language processing (NLP) and machine learning to beat two of the world's best human Jeopardy! players. We realized that Watson could probably handle problems like the sequence predictor because it was loaded with the world's street maps and a whole lot more. That whet our appetite for digging in deep on big data and today's artificial-intelligence technologies.

It's notable that all of the data-science implementation case studies in Chapters 12 to 17 either are rooted in artificial intelligence technologies or discuss the big data hardware and software infrastructure that enables data scientists to implement leading-edge AI-based solutions effectively.

AI: A Field with Problems But No Solutions

For many decades, AI has been a field with problems and *no* solutions. That's because once a particular problem is solved people say, "Well, that's not intelligence, it's just a computer program that tells the computer exactly what to do." However, with machine learning (Chapter 15), deep learning (Chapter 16) and reinforcement learning (Chapter 16 exercises), we're not pre-programming solutions to *specific* problems. Instead, we're letting our computers solve problems by learning from data—and, typically, lots of it.

Many of the most interesting and challenging problems are being pursued with deep learning. Google alone has thousands of deep-learning projects underway and that number is growing quickly.^{88, 89} As you work through this book, we'll introduce you to many edge-of-the-practice artificial intelligence, big data and cloud technologies and you'll work through hundreds of (often intriguing) examples, exercises and projects.



Self Check

1 (Fill-In) The ultimate goal of AI is to produce a(n) _____.

Answer: artificial general intelligence.

2 (Fill-In) IBM's Watson beat the two best human Jeopardy! players. Watson was trained using a combination of _____ learning and _____ learning techniques.

Answer: machine, reinforcement.

88. <http://theweek.com/speedreads/654463/google-more-than-1000-artificial-intelligence-projects-works>.

89. <https://www.zdnet.com/article/google-says-exponential-growth-of-ai-is-changing-nature-of-compute/>.

3 (Fill-In) Google’s _____ taught itself to play chess in less than four hours using reinforcement learning, then beat the world champion chess program, Stockfish 8, in a 100-game match—winning or drawing every game.

Answer: AlphaZero.

Exercises

1.1 (IPython Session) Using the techniques you learned in Section 1.10.1, execute the following expressions. Which, if any, produce a runtime error?

- a) $10 / 3$
- b) $10 // 3$
- c) $10 / 0$
- d) $10 // 0$
- e) $0 / 10$
- f) $0 // 10$

1.2 (IPython Session) Using the techniques you learned in Section 1.10.1, execute the following expressions. Which, if any, produce a runtime error?

- a) $10 / 3 + 7$
- b) $10 // 3 + 7$
- c) $10 / (3 + 7)$
- d) $10 / 3 - 3$
- e) $10 / (3 - 3)$
- f) $10 // (3 - 3)$

1.3 (Creating a Jupyter Notebook) Using the techniques you learned in Section 1.10.3, create a Jupyter Notebook containing cells for the previous exercise’s expressions and execute those expressions.

1.4 (Computer Organization) Fill in the blanks in each of the following statements:

- a) The logical unit that receives information from outside the computer for use by the computer is the _____.
- b) _____ is a logical unit that sends information which has already been processed by the computer to various devices so that it may be used outside the computer.
- c) _____ and _____ are logical units of the computer that retain information.
- d) _____ is a logical unit of the computer that performs calculations.
- e) _____ is a logical unit of the computer that makes logical decisions.
- f) _____ is a logical unit of the computer that coordinates the activities of all the other logical units.

1.5 (Clock as an Object) Clocks are among the world’s most common objects. Discuss how each of the following terms and concepts applies to the notion of a clock: class, object, instantiation, instance variable, reuse, method, inheritance (consider, for example, an alarm clock), superclass, subclass.

1.6 (Gender Neutrality) Write down the steps of a manual procedure for processing a paragraph of text and replacing gender-specific words with gender-neutral ones. Assuming that you’ve been given a list of gender-specific words and their gender-neutral replacements (for example, replace “wife” or “husband” with “spouse,” replace “man” or “woman” with “person,” replace “daughter” or “son” with “child,” and so on), explain the

procedure you'd use to read through a paragraph of text and manually perform these replacements. How might your procedure generate a strange term like "woperchild" and how might you modify your procedure to avoid this possibility? In Chapter 3, you'll learn that a more formal computing term for "procedure" is "algorithm," and that an algorithm specifies the *steps* to be performed and the *order* in which to perform them.

1.7 (Self-Driving Cars) Just a few years back the notion of driverless cars on our streets would have seemed impossible (in fact, our spell-checking software doesn't recognize the word "driverless"). Many of the technologies you'll study in this book are making self-driving cars possible. They're already common in some areas.

- a) If you hailed a taxi and a driverless taxi stopped for you, would you get into the back seat? Would you feel comfortable telling it where you want to go and trusting that it would get you there? What kinds of safety measures would you want in place? What would you do if the car headed off in the wrong direction?
- b) What if two self-driving cars approached a one-lane bridge from opposite directions? What protocol should they go through to determine which car should proceed?
- c) If a police officer pulls over a speeding self-driving car in which you're the only passenger, who—or what entity—should pay the ticket?
- d) What if you're behind a car stopped at a red light, the light turns green and the car doesn't move? You honk and nothing happens. You get out of your car and notice that there's no driver. What would you do?
- e) One serious concern with self-driving vehicles is that they could potentially be hacked. Someone could set the speed high (or low), which could be dangerous. What if they redirect you to a destination other than what you want?
- f) Imagine other scenarios that self-driving cars will encounter.

1.8 (Research: Reproducibility) A crucial concept in data-science studies is reproducibility, which helps others (and you) reproduce your results. Research reproducibility and list the concepts used to create reproducible results in data-science studies. Research and discuss the part that Jupyter Notebooks play in reproducibility.

1.9 (Research: Artificial General Intelligence) One of the most ambitious goals in the field of AI is to achieve *artificial general intelligence*—the point at which machine intelligence would equal human intelligence. Research this intriguing topic. When is this forecast to happen? What are some key ethical issues this raises? Human intelligence seems to be stable over long periods. Powerful computers with artificial general intelligence could conceivably (and quickly) evolve intelligence far beyond that of humans. Research and discuss the issues this raises.

1.10 (Research: Intelligent Assistants) Many companies now offer computerized intelligent assistants, such as IBM Watson, Amazon Alexa, Apple Siri, Google Assistant and Microsoft Cortana. Research these and others and list uses that can improve people's lives. Research privacy and ethics issues for intelligent assistants. Locate amusing intelligent-assistant anecdotes.

1.11 (Research: AI in Health Care) Research the rapidly growing field of AI big-data applications in health care. For example, suppose a diagnostic medical application had access to every x-ray that's ever been taken and the associated diagnoses—that's surely big data. As you'll see in the "Deep Learning" chapter, computer-vision applications can work with

this "labeled" data to learn to diagnose medical problems. Research deep learning in diagnostic medicine and describe some of its most significant accomplishments. What are some ethical issues of having machines instead of human doctors performing medical diagnoses? Would you trust a machine-generated diagnosis? Would you ask for a second opinion?

1.12 (Research: Big Data, AI and the Cloud—How Companies Use These Technologies) For a major organization of your choice, research how they may be using each of the following technologies that you'll use in this book: Python, AI, big data, the cloud, mobile, natural language processing, speech recognition, speech synthesis, database, machine learning, deep learning, reinforcement learning, Hadoop, Spark, Internet of Things (IoT) and web services.

1.13 (Research: Raspberry Pi and the Internet of Things) It's now possible to have a computer at the heart of just about any type of device and to connect those devices to the Internet. This has led to the Internet of Things (IoT), which already interconnects tens of billions of devices. The Raspberry Pi is an economical computer which is often at the heart of IoT devices. Research the Raspberry Pi and some of the many IoT applications in which it's used.

1.14 (Research: The Ethics of Deep Fakes) Artificial-intelligence technologies are making it possible to create *deep fakes*—realistic fake videos of people that capture their appearance, voice, body motions and facial expressions. You can have them say and do whatever you specify. Research the ethics of deep fakes. What would happen if you turned on your TV and saw a deep-fake video of a prominent government official or newscaster reporting that a nuclear attack was about to happen? Research Orson Welles and his "War of the Worlds" radio broadcast of 1938, which created mass panic.

1.15 (Public-Key Cryptography) Cryptography is a crucial technology for privacy and security. Research Python's cryptography capabilities. Research online for a simple explanation of how public-key cryptography is used to implement the BitCoin cryptocurrency.

1.16 (Blockchain: A World of Opportunity) Cryptocurrencies like Bitcoin and Ethereum are based on a technology called blockchain that has seen explosive growth over the last few years. Research blockchain's origin, applications and how it came to be used as the basis for cryptocurrencies. Research other major applications of blockchain. Over the next many years there will be extraordinary opportunities for software developers who thoroughly understand blockchain applications development.

1.17 (OWASP Python Security Project) Building secure computer applications is a tremendous challenge. Many of the world's largest companies, government agencies, and military organizations have had their systems compromised. The OWASP project is concerned with "hardening" computer systems and applications to resist attacks. Research OWASP and discuss their accomplishments and current challenges.

1.18 (IBM Watson) We discuss IBM's Watson in Chapter 14. You'll use its cognitive computing capabilities to quickly build some intriguing applications. IBM is partnering with tens of thousands of companies—including our publisher, Pearson Education—across a wide range of industries. Research some of Watson's key accomplishments and the kinds of challenges IBM and its partners are addressing.

1.19 (Research: Mobile App Development with Python) Research the tools that are available for Python-based iOS and Android app development, such as BeeWare, Kivy, Py-

Mob, Pythonista and others. Which of these are cross-platform? Mobile applications development is one of the fastest growing areas of software development, and it's a great source of class projects, directed study projects, capstone exercise projects and even thesis projects. With cross-platform app-development tools, you'll be able to write your own apps and deploy them on many app stores quickly.

Introduction to Python Programming

2



Objectives

In this chapter, you'll:

- Continue using IPython interactive mode to enter code snippets and see their results immediately.
- Write simple Python statements and scripts.
- Create variables to store data for later use.
- Become familiar with built-in data types.
- Use arithmetic operators and comparison operators, and understand their precedence.
- Use single-, double- and triple-quoted strings.
- Use built-in function `print` to display text.
- Use built-in function `input` to prompt the user to enter data at the keyboard and get that data for use in the program.
- Convert text to integer values with built-in function `int`.
- Use comparison operators and the `if` statement to decide whether to execute a statement or group of statements.
- Learn about objects and Python's dynamic typing.
- Use built-in function `type` to get an object's type.

Outline

-
- | | |
|--|--|
| 2.1 Introduction
2.2 Variables and Assignment Statements
2.3 Arithmetic
2.4 Function <code>print</code> and an Intro to Single- and Double-Quoted Strings
2.5 Triple-Quoted Strings
2.6 Getting Input from the User | 2.7 Decision Making: The <code>if</code> Statement and Comparison Operators
2.8 Objects and Dynamic Typing
2.9 Intro to Data Science: Basic Descriptive Statistics
2.10 Wrap-Up Exercises |
|--|--|
-

2.1 Introduction

In this chapter, we introduce Python programming and present examples illustrating key language features. We assume you've read the IPython Test-Drive in Chapter 1, which introduced the IPython interpreter and used it to evaluate simple arithmetic expressions.

2.2 Variables and Assignment Statements

You've used IPython's interactive mode as a calculator with expressions such as

```
In [1]: 45 + 72
Out[1]: 117
```

As in algebra, Python expressions also may contain **variables**, which store values for later use in your code. Let's create a variable named `x` that stores the integer 7, which is the variable's **value**:

```
In [2]: x = 7
```

Snippet [2] is a **statement**. Each statement specifies a task to perform. The preceding statement creates `x` and uses the **assignment symbol** (`=`) to give `x` a value. The entire statement is an **assignment statement** that we read as "`x` is assigned the value 7." Most statements stop at the end of the line, though it's possible for statements to span more than one line. The following statement creates the variable `y` and assigns to it the value 3:

```
In [3]: y = 3
```

Adding Variable Values and Viewing the Result

You can now use the values of `x` and `y` in expressions:

```
In [4]: x + y
Out[4]: 10
```

The `+` symbol is the **addition operator**. It's a **binary operator** because it has *two operands* (in this case, the variables `x` and `y`) on which it performs its operation.

Calculations in Assignment Statements

You'll often save calculation results for later use. The following assignment statement adds the values of variables `x` and `y` and assigns the result to the variable `total`, which we then display:

```
In [5]: total = x + y
```

```
In [6]: total
Out[6]: 10
```

Snippet [5] is read, “`total` is assigned the value of $x + y$.” The `=` symbol is not an operator. The right side of the `=` symbol always executes first, then the result is assigned to the variable on the symbol’s left side.

Python Style

The *Style Guide for Python Code*¹ helps you write code that conforms to Python’s coding conventions. The style guide recommends inserting one space on each side of the assignment symbol `=` and binary operators like `+` to make programs more readable.

Variable Names

A variable name, such as `x`, is an **identifier**. Each identifier may consist of letters, digits and underscores (`_`) but may not begin with a digit. Python is **case sensitive**, so `number` and `Number` are *different* identifiers because one begins with a lowercase letter and the other begins with an uppercase letter.

Types

Each value in Python has a **type** that indicates the kind of data the value represents. You can view a value’s type, as in:

```
In [7]: type(x)
Out[7]: int

In [8]: type(10.5)
Out[8]: float
```

The variable `x` contains the integer value `7` (from snippet [2]), so Python displays `int` (short for integer). The value `10.5` is a **floating-point number** (that is, a number with a decimal point), so Python displays `float`.

Python’s **type built-in function** determines a value’s type. A **function** performs a task when you **call** it by writing its name, followed by parentheses, `()`. The parentheses contain the function’s **argument**—the data that the `type` function needs to perform its task. You’ll create *custom* functions in later chapters.



Self Check

- 1 *(True/False)* The following are valid variable names: `3g`, `87` and `score_4`.
Answer: False. Because they begin with a digit, `3g` and `87` are invalid names.

- 2 *(True/False)* Python treats `y` and `Y` as the same identifier.
Answer: False. Python is case sensitive, so `y` and `Y` are different identifiers.

- 3 *(IPython Session)* Calculate the sum of `10.8`, `12.2` and `0.2`, store it in the variable `total`, then display `total`’s value.
Answer:

```
In [1]: total = 10.8 + 12.2 + 0.2

In [2]: total
Out[2]: 23.1
```

1. <https://www.python.org/dev/peps/pep-0008/>.

2.3 Arithmetic

Many programs perform arithmetic calculations. The following table summarizes the **arithmetic operators**, which include some symbols not used in algebra.

Python operation	Arithmetic operator	Algebraic expression	Python expression
Addition	+	$f + 7$	<code>f + 7</code>
Subtraction	-	$p - c$	<code>p - c</code>
Multiplication	*	$b \cdot m$	<code>b * m</code>
Exponentiation	**	x^y	<code>x ** y</code>
True division	/	x/y or $\frac{x}{y}$ or $x \div y$	<code>x / y</code>
Floor division	//	$\lfloor x/y \rfloor$ or $\left\lfloor \frac{x}{y} \right\rfloor$ or $\lfloor x \div y \rfloor$	<code>x // y</code>
Remainder (modulo)	%	$r \bmod s$	<code>r % s</code>

Multiplication (*)

Rather than algebra's center dot (\cdot), Python uses the **asterisk (*) multiplication operator**:

```
In [1]: 7 * 4
Out[1]: 28
```

Exponentiation (***)

The **exponentiation (***) operator** raises one value to the power of another:

```
In [2]: 2 ** 10
Out[2]: 1024
```

To calculate the square root, you can use the exponent $1/2$ (that is, 0.5):

```
In [3]: 9 ** (1 / 2)
Out[3]: 3.0
```

True Division (/) vs. Floor Division (//)

True division (/) divides a numerator by a denominator and yields a floating-point number with a decimal point, as in:

```
In [4]: 7 / 4
Out[4]: 1.75
```

Floor division (//) divides a numerator by a denominator, yielding the highest *integer* that's not greater than the result. Python **truncates** (discards) the fractional part:

```
In [5]: 7 // 4
Out[5]: 1
```

```
In [6]: 3 // 5
Out[6]: 0
```

```
In [7]: 14 // 7
Out[7]: 2
```

In true division, -13 divided by 4 gives -3.25:

```
In [8]: -13 / 4
Out[8]: -3.25
```

Floor division gives the closest integer that's *not greater than* -3.25—which is -4:

```
In [9]: -13 // 4
Out[9]: -4
```

Exceptions and Tracebacks

Dividing by zero with / or // is not allowed and results in an **exception**—a sign that a problem occurred:

```
In [10]: 123 / 0
-----
ZeroDivisionError                                 Traceback (most recent call last)
<ipython-input-10-cd759d3fcf39> in <module>()
----> 1 123 / 0

ZeroDivisionError: division by zero
```

Python reports an exception with a **traceback**. This traceback indicates that an exception of type `ZeroDivisionError` occurred—most exception names end with `Error`. In interactive mode, the snippet number that caused the exception is specified by the 10 in the line

```
<ipython-input-10-cd759d3fcf39> in <module>()
```

The line that begins with ----> 1 shows the code that caused the exception. Sometimes snippets have more than one line of code—the 1 to the right of ----> indicates that line 1 within the snippet caused the exception. The last line shows the exception that occurred, followed by a colon (:) and an error message with more information about the exception:

```
ZeroDivisionError: division by zero
```

The “Files and Exceptions” chapter discusses exceptions in detail.

An exception also occurs if you try to use a variable that you have not yet created. The following snippet tries to add 7 to the undefined variable `z`, resulting in a `NameError`:

```
In [11]: z + 7
-----
NameError                                 Traceback (most recent call last)
<ipython-input-11-f2cbf4fe75d> in <module>()
----> 1 z + 7

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

Remainder Operator

Python’s **remainder operator (%)** yields the remainder after the left operand is divided by the right operand:

```
In [12]: 17 % 5
Out[12]: 2
```

In this case, 17 divided by 5 yields a quotient of 3 and a remainder of 2. This operator is most commonly used with integers, but also can be used with other numeric types:

```
In [13]: 7.5 % 3.5
Out[13]: 0.5
```

In the exercises, we use the remainder operator for applications such as determining whether one number is a multiple of another—a special case of this is determining whether a number is odd or even.

Straight-Line Form

Algebraic notations such as

$$\frac{a}{b}$$

generally are not acceptable to compilers or interpreters. For this reason, algebraic expressions must be typed in **straight-line form** using Python's operators. The expression above must be written as `a / b` (or `a // b` for floor division) so that all operators and operands appear in a horizontal straight line.

Grouping Expressions with Parentheses

Parentheses group Python expressions, as they do in algebraic expressions. For example, the following code multiplies 10 times the quantity $5 + 3$:

```
In [14]: 10 * (5 + 3)
Out[14]: 80
```

Without these parentheses, the result is *different*:

```
In [15]: 10 * 5 + 3
Out[15]: 53
```

The parentheses are **redundant** (unnecessary) if removing them yields the *same* result.

Operator Precedence Rules

Python applies the operators in arithmetic expressions according to the following **rules of operator precedence**. These are generally the same as those in algebra:

1. Expressions in parentheses evaluate first, so parentheses may force the order of evaluation to occur in any sequence you desire. Parentheses have the highest level of precedence. In expressions with **nested parentheses**, such as `(a / (b - c))`, the expression in the *innermost* parentheses (that is, $b - c$) evaluates first.
2. Exponentiation operations evaluate next. If an expression contains several exponentiation operations, Python applies them from right to left.
3. Multiplication, division and modulus operations evaluate next. If an expression contains several multiplication, true-division, floor-division and modulus operations, Python applies them from left to right. Multiplication, division and modulus are “on the same level of precedence.”
4. Addition and subtraction operations evaluate last. If an expression contains several addition and subtraction operations, Python applies them from left to right. Addition and subtraction also have the same level of precedence.

We'll expand these rules as we introduce other operators. For the complete list of operators and their precedence (in lowest-to-highest order), see

<https://docs.python.org/3/reference/expressions.html#operator-precedence>

Operator Grouping

When we say that Python applies certain operators from left to right, we are referring to the operators' **grouping**. For example, in the expression

$$a + b + c$$

the addition operators (+) group from left to right as if we parenthesized the expression as $(a + b) + c$. All Python operators of the same precedence group left-to-right except for the exponentiation operator (**), which groups right-to-left.

Redundant Parentheses

You can use redundant parentheses to group subexpressions to make the expression clearer. For example, the second-degree polynomial

$$y = a * x ** 2 + b * x + c$$

can be parenthesized, for clarity, as

$$y = (a * (x ** 2)) + (b * x) + c$$

Breaking a complex expression into a sequence of statements with shorter, simpler expressions also can promote clarity.

Operand Types

Each arithmetic operator may be used with integers and floating-point numbers. If both operands are integers, the result is an integer—except for the true-division (/) operator, which always yields a floating-point number. If both operands are floating-point numbers, the result is a floating-point number. Expressions containing an integer and a floating-point number are **mixed-type expressions**—these always produce floating-point numbers.



Self Check

1 (Multiple Choice) Given that $y = ax^3 + 7$, which of the following is not a correct statement for this equation?

- a) $y = a * x * x * x + 7$
- b) $y = a * x ** 3 + 7$
- c) $y = a * (x * x * x) + 7$
- d) $y = a * x * (x * x + 7)$

Answer: d is incorrect.

2 (True/False) In nested parentheses, the expression in the innermost pair evaluates last.
Answer: False. The expression in the innermost parentheses evaluates first.

3 (IPython Session) Evaluate the expression $3 * (4 - 5)$ with and without parentheses.
Are the parentheses redundant?

Answer:

```
In [1]: 3 * (4 - 5)
Out[1]: -3
```

```
In [2]: 3 * 4 - 5
Out[2]: 7
```

The parentheses are not redundant—if you remove them the resulting value is different.

4 (IPython Session) Evaluate the expressions $4 ** 3 ** 2$, $(4 ** 3) ** 2$ and $4 ** (3 ** 2)$. Are any of the parentheses redundant?

Answer:

```
In [3]: 4 ** 3 ** 2
Out[3]: 262144
```

```
In [4]: (4 ** 3) ** 2
Out[4]: 4096
```

```
In [5]: 4 ** (3 ** 2)
Out[5]: 262144
```

Only the parentheses in the last expression are redundant.

2.4 Function print and an Intro to Single- and Double-Quoted Strings

The built-in **print** function displays its argument(s) as a line of text:

```
In [1]: print('Welcome to Python!')
Welcome to Python!
```

In this case, the argument 'Welcome to Python!' is a **string**—a sequence of characters enclosed in single quotes (''). Unlike when you evaluate expressions in interactive mode, the text that **print** displays here is not preceded by **Out[1]**. Also, **print** does not display a string's quotes, though we'll soon show how to display quotes in strings.

You also may enclose a string in double quotes (""), as in:

```
In [2]: print("Welcome to Python!")
Welcome to Python!
```

Python programmers generally prefer single quotes.

When **print** completes its task, it positions the screen cursor at the beginning of the next line. This is similar to what happens when you press the *Enter* (or *Return*) key while typing in a text editor.

Printing a Comma-Separated List of Items

The **print** function can receive a comma-separated list of arguments, as in:

```
In [3]: print('Welcome', 'to', 'Python!')
Welcome to Python!
```

The **print** function displays each argument separated from the next by a space, producing the same output as in the two preceding snippets. Here we showed a comma-separated list of strings, but the values can be of any type. We'll show in the next chapter how to prevent automatic spacing between values or use a different separator than space.

Printing Many Lines of Text with One Statement

When a backslash (\) appears in a string, it's known as the **escape character**. The backslash and the character immediately following it form an **escape sequence**. For example, \n represents the **newline character** escape sequence, which tells **print** to move the output cursor to the next line. Placing two newline characters back-to-back displays a blank line. The following snippet uses three newline characters to create many lines of output:

```
In [4]: print('Welcome\n to\n Python!')
Welcome
to
Python!
```

Other Escape Sequences

The following table shows some common escape sequences.

Escape sequence	Description
\n	Insert a newline character in a string. When the string is displayed, for each newline, move the screen cursor to the beginning of the next line.
\t	Insert a horizontal tab. When the string is displayed, for each tab, move the screen cursor to the next tab stop.
\\"	Insert a backslash character in a string.
\"	Insert a double quote character in a string.
\'	Insert a single quote character in a string.

Ignoring a Line Break in a Long String

You may also split a long string (or a long statement) over several lines by using the **\ continuation character** as the last character on a line to ignore the line break:

```
In [5]: print('this is a longer string, so we \
...: split it over two lines')
this is a longer string, so we split it over two lines
```

The interpreter reassembles the string's parts into a single string with no line break. Though the backslash character in the preceding snippet is inside a string, it's not the escape character because another character does not follow it.

Printing the Value of an Expression

Calculations can be performed in `print` statements:

```
In [6]: print('Sum is', 7 + 3)
Sum is 10
```



Self Check

1 *(Fill-In)* The _____ function instructs the computer to display information on the screen.

Answer: `print`.

2 *(Fill-In)* Values of the _____ data type contain a sequence of characters.

Answer: string (type `str`).

3 *(IPython Session)* Write an expression that displays the type of 'word'.

Answer:

```
In [1]: type('word')
Out[1]: str
```

4 (*IPython Session*) What does the following `print` statement display?

```
print('int(5.2)', 'truncates 5.2 to', int(5.2))
```

Answer:

```
In [2]: print('int(5.2)', 'truncates 5.2 to', int(5.2))
int(5.2) truncates 5.2 to 5
```

2.5 Triple-Quoted Strings

Earlier, we introduced strings delimited by a pair of single quotes ('') or a pair of double quotes (""). **Triple-quoted strings** begin and end with three double quotes (""""") or three single quotes (''''). The *Style Guide for Python Code* recommends three double quotes ("""""). Use these to create:

- multiline strings,
- strings containing single or double quotes and
- **docstrings**, which are the recommended way to document the purposes of certain program components.

Including Quotes in Strings

In a string delimited by single quotes, you may include double-quote characters:

```
In [1]: print('Display "hi" in quotes')
Display "hi" in quotes
```

but not single quotes:

```
In [2]: print('Display 'hi' in quotes')
  File "<ipython-input-2-19bf596ccf72>", line 1
    print('Display 'hi' in quotes')
               ^
SyntaxError: invalid syntax
```

unless you use the \' escape sequence:

```
In [3]: print('Display \'hi\' in quotes')
Display 'hi' in quotes
```

Snippet [2] displayed a **syntax error**, which is a violation of Python's language rules—in this case, a single quote inside a single-quoted string. IPython displays information about the line of code that caused the syntax error and points to the error with a ^ symbol. It also displays the message `SyntaxError: invalid syntax`.

A string delimited by double quotes may include single quote characters:

```
In [4]: print("Display the name O'Brien")
Display the name O'Brien
```

but not double quotes, unless you use the \" escape sequence:

```
In [5]: print("Display \"hi\" in quotes")
Display "hi" in quotes
```

To avoid using \' and \" inside strings, you can enclose such strings in triple quotes:

```
In [6]: print("""Display "hi" and 'bye' in quotes""")
Display "hi" and 'bye' in quotes
```

Multiline Strings

The following snippet assigns a multiline triple-quoted string to `triple_quoted_string`:

```
In [7]: triple_quoted_string = """This is a triple-quoted
...: string that spans two lines"""


```

IPython knows that the string is incomplete because we did not type the closing """" before we pressed *Enter*. So, IPython displays a **continuation prompt** ...: at which you can input the multiline string's next line. This continues until you enter the ending """" and press *Enter*. The following displays `triple_quoted_string`:

```
In [8]: print(triple_quoted_string)
This is a triple-quoted
string that spans two lines
```

Python stores multiline strings with embedded newline escape sequences. When we evaluate `triple_quoted_string` rather than printing it, IPython displays the string in single quotes with a \n character where you pressed *Enter* in snippet [7]. The quotes IPython displays indicate that `triple_quoted_string` is a string—they're not part of the string's contents:

```
In [9]: triple_quoted_string
Out[9]: 'This is a triple-quoted\nstring that spans two lines'
```



Self Check

1 (*Fill-In*) Multiline strings are enclosed either in _____ or in _____.

Answer: """ (triple double quotes) or ''' (triple single quotes).

2 (*IPython Session*) What displays when you execute the following statement?

```
print("""This is a lengthy
       multiline string containing
       a few lines \
       of text""")
```

Answer:

```
In [1]: print("""This is a lengthy
...:     multiline string containing
...:     a few lines \
...:     of text""")
This is a lengthy
       multiline string containing
       a few lines of text
```

2.6 Getting Input from the User

The built-in `input` function requests and obtains user input:

```
In [1]: name = input("What's your name? ")
What's your name? Paul
```

```
In [2]: name
Out[2]: 'Paul'
```

```
In [3]: print(name)
Paul
```

The snippet executes as follows:

- First, `input` displays its string argument—called a **prompt**—to tell the user what to type and waits for the user to respond. We typed Paul (without quotes) and pressed *Enter*. We use **bold** text to distinguish the user's input from the prompt text that `input` displays.
- Function `input` then **returns** (that is, gives back) those characters as a string that the program can use. Here we assigned that string to the variable `name`.

Snippet [2] shows `name`'s value. Evaluating `name` displays its value in single quotes as '**Paul**' because it's a string. Printing `name` (in snippet [3]) displays the string without the quotes. If you enter quotes, they're part of the string, as in:

```
In [4]: name = input("What's your name? ")
What's your name? 'Paul'

In [5]: name
Out[5]: "'Paul'"

In [6]: print(name)
'Paul'
```

Function `input` Always Returns a String

Consider the following snippets that attempt to read two numbers and add them:

```
In [7]: value1 = input('Enter first number: ')
Enter first number: 7

In [8]: value2 = input('Enter second number: ')
Enter second number: 3

In [9]: value1 + value2
Out[9]: '73'
```

Rather than adding the integers 7 and 3 to produce 10, Python “adds” the *string* values '7' and '3', producing the *string* '73'. This is known as **string concatenation**. It creates a new string containing the left operand's value followed by the right operand's value.

Getting an Integer from the User

If you need an integer, convert the string to an integer using the built-in **`int`** function:

```
In [10]: value = input('Enter an integer: ')
Enter an integer: 7

In [11]: value = int(value)

In [12]: value
Out[12]: 7
```

We could have combined the code in snippets [10] and [11]:

```
In [13]: another_value = int(input('Enter another integer: '))
Enter another integer: 13

In [14]: another_value
Out[14]: 13
```

Variables `value` and `another_value` now contain integers. Adding them produces an integer result (rather than concatenating them):

```
In [15]: value + another_value
Out[15]: 20
```

If the string passed to `int` cannot be converted to an integer, a `ValueError` occurs:

```
In [16]: bad_value = int(input('Enter another integer: '))
Enter another integer: hello
-----
ValueError                                Traceback (most recent call last)
<ipython-input-16-cd36e6cf8911> in <module>()
      1 bad_value = int(input('Enter another integer: '))
-----
```

```
ValueError: invalid literal for int() with base 10: 'hello'
```

Function `int` also can convert a floating-point value to an integer:

```
In [17]: int(10.5)
Out[17]: 10
```

To convert strings to floating-point numbers, use the built-in **float function**.



Self Check

- 1 (*Fill-In*) The built-in _____ function converts a floating-point value to an integer value or converts a string representation of an integer to an integer value.

Answer: `int`.

- 2 (*True/False*) Built-in function `get_input` requests and obtains input from the user.

Answer: False. The built-in function's name is `input`.

- 3 (*IPython Session*) Use `float` to convert '`6.2`' (a string) to a floating-point value. Multiply that value by `3.3` and show the result.

Answer:

```
In [1]: float('6.2') * 3.3
Out[1]: 20.46
```

2.7 Decision Making: The if Statement and Comparison Operators

A **condition** is a Boolean expression with the value **True** or **False**. The following determines whether 7 is greater than 4 and whether 7 is less than 4:

```
In [1]: 7 > 4
Out[1]: True

In [2]: 7 < 4
Out[2]: False
```

`True` and `False` are **keywords**—words that Python reserves for its language features. Using a keyword as an identifier causes a `SyntaxError`. `True` and `False` are each capitalized.

You'll often create conditions using the **comparison operators** in the table at the top of the next page:

Algebraic operator	Python operator	Sample condition	Meaning
>	>	$x > y$	x is greater than y
<	<	$x < y$	x is less than y
\geq	\geq	$x \geq y$	x is greater than or equal to y
\leq	\leq	$x \leq y$	x is less than or equal to y
=	==	$x == y$	x is equal to y
\neq	!=	$x != y$	x is not equal to y

Operators $>$, $<$, \geq and \leq all have the same precedence. Operators $==$ and $!=$ both have the same precedence, which is lower than that of $>$, $<$, \geq and \leq . A syntax error occurs when any of the operators $==$, $!=$, \geq and \leq contains spaces between its pair of symbols:

```
In [3]: 7 > = 4
File "<ipython-input-3-5c6e2897f3b3>", line 1
    7 > = 4
          ^
SyntaxError: invalid syntax
```

Another syntax error occurs if you reverse the symbols in the operators $!=$, \geq and \leq (by writing them as $=!$, \geq and \leq).

Making Decisions with the `if` Statement: Introducing Scripts

We now present a simple version of the **if statement**, which uses a condition to decide whether to execute a statement (or a group of statements). Here we'll read two integers from the user and compare them using six consecutive `if` statements, one for each comparison operator. If the condition in a given `if` statement is `True`, the corresponding `print` statement executes; otherwise, it's skipped.

IPython interactive mode is helpful for executing brief code snippets and seeing immediate results. When you have many statements to execute as a group, you typically write them as a **script** stored in a file with the `.py` (short for Python) extension—such as `fig02_01.py` for this example's script. Scripts are also called **programs**. For instructions on locating and executing the scripts in this book, see Chapter 1's IPython Test-Drive.

Each time you execute this script, three of the six conditions are `True`. To show this, we execute the script three times—once with the first integer *less than* the second, once with the *same* value for both integers and once with the first integer *greater than* the second. The three sample executions appear after the script

Figure 2.1 shows the script. Each time we present a script, we introduce it before the figure, then explain the script's code after the figure. We show line numbers for your convenience—these are not part of Python. Integrated development environments (IDEs) enable you to choose whether to display line numbers. To run this example, change to this chapter's `ch02` examples folder, then enter:

```
ipython fig02_01.py
```

or, if you're in IPython already, use the command:

```
run fig02_01.py
```

```
1 # fig02_01.py
2 """Comparing integers using if statements and comparison operators."""
3
4 print('Enter two integers, and I will tell you',
5       'the relationships they satisfy.')
6
7 # read first integer
8 number1 = int(input('Enter first integer: '))
9
10 # read second integer
11 number2 = int(input('Enter second integer: '))
12
13 if number1 == number2:
14     print(number1, 'is equal to', number2)
15
16 if number1 != number2:
17     print(number1, 'is not equal to', number2)
18
19 if number1 < number2:
20     print(number1, 'is less than', number2)
21
22 if number1 > number2:
23     print(number1, 'is greater than', number2)
24
25 if number1 <= number2:
26     print(number1, 'is less than or equal to', number2)
27
28 if number1 >= number2:
29     print(number1, 'is greater than or equal to', number2)
```

```
Enter two integers and I will tell you the relationships they satisfy.
Enter first integer: 37
Enter second integer: 42
37 is not equal to 42
37 is less than 42
37 is less than or equal to 42
```

```
Enter two integers and I will tell you the relationships they satisfy.
Enter first integer: 7
Enter second integer: 7
7 is equal to 7
7 is less than or equal to 7
7 is greater than or equal to 7
```

```
Enter two integers and I will tell you the relationships they satisfy.
Enter first integer: 54
Enter second integer: 17
54 is not equal to 17
54 is greater than 17
54 is greater than or equal to 17
```

Fig. 2.1 | Comparing integers using if statements and comparison operators.

Comments

Line 1 begins with the hash character (#), which indicates that the rest of the line is a **comment**:

```
# fig02_01.py
```

You insert comments to document your code and to improve readability. Comments also help other programmers read and understand your code. They do not cause the computer to perform any action when the code executes. For easy reference, we begin each script with a comment indicating the script's file name.

A comment also can begin to the right of the code on a given line and continue until the end of that line. Such a comment documents the code to its left.

Docstrings

The *Style Guide for Python Code* states that each script should start with a docstring that explains the script's purpose, such as the one in line 2:

```
"""Comparing integers using if statements and comparison operators."""
```

For more complex scripts, the docstring often spans many lines. In later chapters, you'll use docstrings to describe script components you define, such as new functions and new types called classes. We'll also discuss how to access docstrings with the IPython help mechanism.

Blank Lines

Line 3 is a blank line. You use blank lines and space characters to make code easier to read. Together, blank lines, space characters and tab characters are known as **white space**. Python ignores most white space—you'll see that some indentation is required.

Splitting a Lengthy Statement Across Lines

Lines 4–5

```
print('Enter two integers, and I will tell you',
      'the relationships they satisfy.')
```

display instructions to the user. These are too long to fit on one line, so we broke them into two strings. Recall that you can display several values by passing to `print` a comma-separated list—`print` separates each value from the next with a space character.

Typically, you write statements on one line. You may spread a lengthy statement over several lines with the \ continuation character. Python also allows you to split long code lines in parentheses without using continuation characters (as in lines 4–5). This is the preferred way to break long code lines according to the *Style Guide for Python Code*. Always choose breaking points that make sense, such as after a comma in the preceding call to `print` or before an operator in a lengthy expression.

Reading Integer Values from the User

Next, lines 8 and 11 use the built-in `input` and `int` functions to prompt for and read two integer values from the user.

if Statements

The `if` statement in lines 13–14

```
if number1 == number2:
    print(number1, 'is equal to', number2)
```

uses the `==` comparison operator to determine whether the values of variables `number1` and `number2` are equal. If so, the condition is `True`, and line 14 displays a line of text indicating that the values are equal. If any of the remaining `if` statements' conditions are `True` (lines 16, 19, 22, 25 and 28), the corresponding `print` displays a line of text.

Each `if` statement consists of the keyword `if`, the condition to test, and a colon (`:`) followed by an indented body called a **suite**. Each suite must contain one or more statements. Forgetting the colon (`:`) after the condition is a common syntax error.

Suite Indentation

Python requires you to indent the statements in suites. The *Style Guide for Python Code* recommends four-space indents—we use that convention throughout this book. You'll see in the next chapter that incorrect indentation can cause errors.

Confusing `==` and `=`

Using the assignment symbol (`=`) instead of the equality operator (`==`) in an `if` statement's condition is a common syntax error. To help avoid this, read `==` as “is equal to” and `=` as “is assigned.” You'll see in the next chapter that using `==` in place of `=` in an assignment statement can lead to subtle problems.

Chaining Comparisons

You can chain comparisons to check whether a value is in a range. The following comparison determines whether `x` is in the range 1 through 5, inclusive:

```
In [1]: x = 3
In [2]: 1 <= x <= 5
Out[2]: True
In [3]: x = 10
In [4]: 1 <= x <= 5
Out[4]: False
```

Precedence of the Operators We've Presented So Far

The precedence of the operators introduced in this chapter is shown below:

Operators	Grouping	Type
<code>()</code>	left to right	parentheses
<code>**</code>	right to left	exponentiation
<code>*</code> / <code>//</code> <code>%</code>	left to right	multiplication, true division, floor division, remainder
<code>+</code> <code>-</code>	left to right	addition, subtraction
<code>></code> <code><=</code> <code><</code> <code>>=</code>	left to right	less than, less than or equal, greater than, greater than or equal
<code>==</code> <code>!=</code>	left to right	equal, not equal

The table lists the operators top-to-bottom in decreasing order of precedence. When writing expressions containing multiple operators, confirm that they evaluate in the order you expect by referring to the operator precedence chart at

<https://docs.python.org/3/reference/expressions.html#operator-precedence>



Self Check

- 1 (*Fill-In*) You use _____ to document code and improve its readability.

Answer: comments.

- 2 (*True/False*) The comparison operators evaluate left to right and all have the same level of precedence.

Answer: False. The operators `<`, `<=`, `>` and `>=` all have the same level of precedence and evaluate left to right. The operators `==` and `!=` have the same level of precedence and evaluate left to right. Their precedence is lower than that of `<`, `<=`, `>` and `>=`.

- 3 (*IPython Session*) For any of the operators `!=`, `>=` or `<=`, show that a syntax error occurs if you reverse the symbols in a condition.

Answer:

```
In [1]: 7 <= 10
File "<ipython-input-1-090d4004a38e>", line 1
      7 < 10
          ^
SyntaxError: invalid syntax
```

- 4 (*IPython Session*) Use all six comparison operators to compare the values 5 and 9. Display the values on one line using `print`.

Answer:

```
In [2]: print(5 < 9, 5 <= 9, 5 > 9, 5 >= 9, 5 == 9, 5 != 9)
True True False False True
```

2.8 Objects and Dynamic Typing

The first chapter introduced the terms *classes* and *objects* and in Section 2.2, we discussed variables, values and types. Values such as 7 (an integer), 4.1 (a floating-point number) and 'dog' are all objects. Every object has a type and a value:

```
In [1]: type(7)
Out[1]: int

In [2]: type(4.1)
Out[2]: float

In [3]: type('dog')
Out[3]: str
```

An object's value is the data stored in the object. The snippets above show objects of Python built-in types `int` (for integers), `float` (for floating-point numbers) and `str` (for strings).

Variables Refer to Objects

Assigning an object to a variable **binds** (associates) that variable's name to the object. As you've seen, you can then use the variable in your code to access the object's value:

```
In [4]: x = 7
```

```
In [5]: x + 10
Out[5]: 17
```

```
In [6]: x
Out[6]: 7
```

After snippet [4]’s assignment, the variable `x` **refers** to the integer object containing 7. As shown in snippet [6], snippet [5] does not change `x`’s value. You can change `x` as follows:

```
In [7]: x = x + 10
```

```
In [8]: x
Out[8]: 17
```

Dynamic Typing

Python uses **dynamic typing**—it determines the type of the object a variable refers to while executing your code. We can show this by rebinding the variable `x` to different objects and checking their types:

```
In [9]: type(x)
Out[9]: int
```

```
In [10]: x = 4.1
```

```
In [11]: type(x)
Out[11]: float
```

```
In [12]: x = 'dog'
```

```
In [13]: type(x)
Out[13]: str
```

Garbage Collection

Python creates objects in memory and removes them from memory as necessary. After snippet [10], the variable `x` now refers to a `float` object. The integer object from snippet [7] is no longer bound to a variable. As we’ll discuss in a later chapter, Python automatically removes such objects from memory. This process—called **garbage collection**—helps ensure that memory is available for new objects you create.



Self Check

1 (Fill-In) Assigning an object to a variable _____ the variable’s name to the object.
Answer: binds.

2 (True/False) A variable always references the same object.

Answer: False. You can make an existing variable refer to a different object and even one of a different type.

3 (IPython Session) What is the type of the expression `7.5 * 3`?

Answer:

```
In [1]: type(7.5 * 3)
Out[1]: float
```

2.9 Intro to Data Science: Basic Descriptive Statistics

In data science, you'll often use statistics to describe and summarize your data. Here, we begin by introducing several such **descriptive statistics**, including:

- **minimum**—the smallest value in a collection of values.
- **maximum**—the largest value in a collection of values.
- **range**—the range of values from the minimum to the maximum.
- **count**—the number of values in a collection.
- **sum**—the total of the values in a collection.

We'll look at determining the *count* and *sum* in the next chapter. **Measures of dispersion** (also called **measures of variability**), such as *range*, help determine how spread out values are. Other measures of dispersion that we'll present in later chapters include *variance* and *standard deviation*.

Determining the Minimum of Three Values

First, let's show how to determine the minimum of three values manually. The following script prompts for and inputs three values, uses *if* statements to determine the minimum value, then displays it.

```

1 # fig02_02.py
2 """Find the minimum of three values."""
3
4 number1 = int(input('Enter first integer: '))
5 number2 = int(input('Enter second integer: '))
6 number3 = int(input('Enter third integer: '))
7
8 minimum = number1
9
10 if number2 < minimum:
11     minimum = number2
12
13 if number3 < minimum:
14     minimum = number3
15
16 print('Minimum value is', minimum)

```

```

Enter first integer: 12
Enter second integer: 27
Enter third integer: 36
Minimum value is 12

```

```

Enter first integer: 27
Enter second integer: 12
Enter third integer: 36
Minimum value is 12

```

Fig. 2.2 | Find the minimum of three values. (Part 1 of 2.)

```
Enter first integer: 36
Enter second integer: 27
Enter third integer: 12
Minimum value is 12
```

Fig. 2.2 | Find the minimum of three values. (Part 2 of 2.)

After inputting the three values, we process one value at a time:

- First, we assume that `number1` contains the smallest value, so line 8 assigns it to the variable `minimum`. Of course, it's possible that `number2` or `number3` contains the actual smallest value, so we still must compare each of these with `minimum`.
- The first `if` statement (lines 10–11) then tests `number2 < minimum` and if this condition is True assigns `number2` to `minimum`.
- The second `if` statement (lines 13–14) then tests `number3 < minimum`, and if this condition is True assigns `number3` to `minimum`.

Now, `minimum` contains the smallest value, so we display it. We executed the script three times to show that it always finds the smallest value regardless of whether the user enters it first, second or third.

Determining the Minimum and Maximum with Built-In Functions `min` and `max`

Python has many built-in functions for performing common tasks. Built-in functions `min` and `max` calculate the minimum and maximum, respectively, of a collection of values:

```
In [1]: min(36, 27, 12)
Out[1]: 12
```

```
In [2]: max(36, 27, 12)
Out[2]: 36
```

The functions `min` and `max` can receive any number of arguments.

Determining the Range of a Collection of Values

The *range* of values is simply the minimum through the maximum value. In this case, the range is 12 through 36. Much data science is devoted to getting to know your data. Descriptive statistics is a crucial part of that, but you also have to understand how to interpret the statistics. For example, if you have 100 numbers with a range of 12 through 36, those numbers could be distributed evenly over that range. At the opposite extreme, you could have clumping with 99 values of 12 and one 36, or one 12 and 99 values of 36. In later data science sections, we'll look at common *data distributions*.

Functional-Style Programming: Reduction

Throughout this book, we introduce various *functional-style programming* capabilities. These enable you to write code that can be more concise, clearer and easier to `debug`—that is, find and correct errors. The `min` and `max` functions are examples of a functional-style programming concept called **reduction**. They reduce a collection of values to a *single* value. Other reductions you'll see include the sum, average, variance and standard deviation of a collection of values. You'll also learn how to define custom reductions.

Upcoming Intro to Data Science Sections

In the next two chapters, we'll continue our discussion of basic descriptive statistics with *measures of central tendency*, including *mean*, *median* and *mode*, and *measures of dispersion*, including *variance* and *standard deviation*.



Self Check

- 1 (Fill-In) The range of a collection of values is a measure of _____.

Answer: dispersion.

- 2 (IPython Session) For the values 47, 95, 88, 73, 88 and 84 calculate the minimum, maximum and range.

Answer:

```
In [1]: min(47, 95, 88, 73, 88, 84)
Out[1]: 47

In [2]: max(47, 95, 88, 73, 88, 84)
Out[2]: 95

In [3]: print('Range:', min(47, 95, 88, 73, 88, 84), '-',
...:         max(47, 95, 88, 73, 88, 84))
...:
Range: 47 - 95
```

2.10 Wrap-Up

This chapter continued our discussion of arithmetic. You used variables to store values for later use. We introduced Python's arithmetic operators and showed that you must write all expressions in straight-line form. You used the built-in function `print` to display data. We created single-, double- and triple-quoted strings. You used triple-quoted strings to create multiline strings and to embed single or double quotes in strings.

You used the `input` function to prompt for and get input from the user at the keyboard. We used the functions `int` and `float` to convert strings to numeric values. We presented Python's comparison operators. Then, you used them in a script that read two integers from the user and compared their values using a series of `if` statements.

We discussed Python's dynamic typing and used the built-in function `type` to display an object's type. Finally, we introduced the basic descriptive statistics minimum and maximum and used them to calculate the range of a collection of values. In the next chapter, you'll learn Python's control statements and program development.

Exercises

Unless specified otherwise, use IPython sessions for each exercise.

- 2.1** (What does this code do?) Create the variables `x = 2` and `y = 3`, then determine what each of the following statements displays:

- `print('x =', x)`
- `print('Value of', x, '+', x, 'is', (x + x))`
- `print('x =')`
- `print((x + y), '=', (y + x))`

2.2 (*What's wrong with this code?*) The following code should read an integer into the variable `rating`:

```
rating = input('Enter an integer rating between 1 and 10')
```

2.3 (*Fill in the missing code*) Replace `***` in the following code with a statement that will print a message like 'Congratulations! Your grade of 91 earns you an A in this course'. Your statement should print the value stored in the variable `grade`:

```
if grade >= 90:  
    ***
```

2.4 (*Arithmetic*) For each of the arithmetic operators `+`, `-`, `*`, `/`, `//` and `**`, display the value of an expression with `27.5` as the left operand and `2` as the right operand.

2.5 (*Circle Area, Diameter and Circumference*) For a circle of radius `2`, display the diameter, circumference and area. Use the value `3.14159` for π . Use the following formulas (r is the radius): $diameter = 2r$, $circumference = 2\pi r$ and $area = \pi r^2$. [In a later chapter, we'll introduce Python's `math` module which contains a higher-precision representation of π .]

2.6 (*Odd or Even*) Use `if` statements to determine whether an integer is odd or even. [Hint: Use the remainder operator. An even number is a multiple of 2. Any multiple of 2 leaves a remainder of 0 when divided by 2.]

2.7 (*Multiples*) Use `if` statements to determine whether `1024` is a multiple of `4` and whether `2` is a multiple of `10`. (Hint: Use the remainder operator.)

2.8 (*Table of Squares and Cubes*) Write a script that calculates the squares and cubes of the numbers from `0` to `5`. Print the resulting values in table format, as shown below. Use the tab escape sequence to achieve the three-column output.

number	square	cube
0	0	0
1	1	1
2	4	8
3	9	27
4	16	64
5	25	125

The next chapter shows how to "right align" numbers. You could try that as an extra challenge here. The output would be:

number	square	cube
0	0	0
1	1	1
2	4	8
3	9	27
4	16	64
5	25	125

2.9 (*Integer Value of a Character*) Here's a peek ahead. In this chapter, you learned about strings. Each of a string's characters has an integer representation. The set of characters a computer uses together with the characters' integer representations is called that computer's *character set*. You can indicate a character value in a program by enclosing that character in quotes, as in '`A`'. To determine a character's integer value, call the built-in function `ord`:

```
In [1]: ord('A')
Out[1]: 65
```

Display the integer equivalents of `B` `C` `D` `b` `c` `d` `0` `1` `2` `$` `*` `+` and the space character.

2.10 (*Arithmetic, Smallest and Largest*) Write a script that inputs three integers from the user. Display the sum, average, product, smallest and largest of the numbers. Note that each of these is a reduction in functional-style programming.

2.11 (*Separating the Digits in an Integer*) Write a script that inputs a five-digit integer from the user. Separate the number into its individual digits. Print them separated by three spaces each. For example, if the user types in the number 42339, the script should print

```
4   2   3   3   9
```

Assume that the user enters the correct number of digits. Use both the floor division and remainder operations to “pick off” each digit.

2.12 (*7% Investment Return*) Some investment advisors say that it’s reasonable to expect a 7% return over the long term in the stock market. Assuming that you begin with \$1000 and leave your money invested, calculate and display how much money you’ll have after 10, 20 and 30 years. Use the following formula for determining these amounts:

$$a = p(1 + r)^n$$

where

p is the original amount invested (i.e., the principal of \$1000),

r is the annual rate of return (7%),

n is the number of years (10, 20 or 30) and

a is the amount on deposit at the end of the n th year.

2.13 (*How Big Can Python Integers Be?*) We’ll answer this question later in the book. For now, use the exponentiation operator `**` with large and very large exponents to produce some huge integers and assign those to the variable `number` to see if Python accepts them. Did you find any integer value that Python won’t accept?

2.14 (*Target Heart-Rate Calculator*) While exercising, you can use a heart-rate monitor to see that your heart rate stays within a safe range suggested by your doctors and trainers. According to the American Heart Association (AHA) (<http://bit.ly/AHATargetHeart-Rates>), the formula for calculating your maximum heart rate in beats per minute is 220 minus your age in years. Your target heart rate is 50–85% of your maximum heart rate. Write a script that prompts for and inputs the user’s age and calculates and displays the user’s maximum heart rate and the range of the user’s target heart rate. [These formulas are estimates provided by the AHA; maximum and target heart rates may vary based on the health, fitness and gender of the individual. Always consult a physician or qualified healthcare professional before beginning or modifying an exercise program.]

2.15 (*Sort in Ascending Order*) Write a script that inputs three different floating-point numbers from the user. Display the numbers in increasing order. Recall that an `if` statement’s suite can contain more than one statement. Prove that your script works by running it on all six possible orderings of the numbers. Does your script work with duplicate numbers? [This is challenging. In later chapters you’ll do this more conveniently and with many more numbers.]

Control Statements and Program Development

3



Objectives

In this chapter, you'll:

- Decide whether to execute actions with the statements `if`, `if...else` and `if...elif...else`.
- Execute statements repeatedly with `while` and `for`.
- Shorten assignment expressions with augmented assignments.
- Use the `for` statement and the built-in `range` function to repeat actions for a sequence of values.
- Perform sentinel-controlled repetition with `while`.
- Learn problem-solving skills: understanding problem requirements, dividing problems into smaller pieces, developing algorithms to solve problems and implementing those algorithms in code.
- Develop algorithms through the process of top-down, stepwise refinement.
- Create compound conditions with the Boolean operators `and`, `or` and `not`.
- Stop looping with `break`.
- Force the next iteration of a loop with `continue`.
- Use some functional-style programming features to write scripts that are more concise, clearer, easier to debug and easier to parallelize.

Outline

3.1	Introduction	3.11	Program Development: Sentinel-Controlled Repetition
3.2	Algorithms	3.12	Program Development: Nested Control Statements
3.3	Pseudocode	3.13	Built-In Function <code>range</code> : A Deeper Look
3.4	Control Statements	3.14	Using Type <code>Decimal</code> for Monetary Amounts
3.5	<code>if</code> Statement	3.15	<code>break</code> and <code>continue</code> Statements
3.6	<code>if...else</code> and <code>if...elif...else</code> Statements	3.16	Boolean Operators <code>and</code> , <code>or</code> and <code>not</code>
3.7	<code>while</code> Statement	3.17	Intro to Data Science: Measures of Central Tendency—Mean, Median and Mode
3.8	<code>for</code> Statement	3.18	Wrap-Up Exercises
3.8.1	Iterables, Lists and Iterators		
3.8.2	Built-In <code>range</code> Function		
3.9	Augmented Assignments		
3.10	Program Development: Sequence-Controlled Repetition		
3.10.1	Requirements Statement		
3.10.2	Pseudocode for the Algorithm		
3.10.3	Coding the Algorithm in Python		
3.10.4	Introduction to Formatted Strings		

3.1 Introduction

Before writing a program to solve a particular problem, you must understand the problem and have a carefully planned approach to solving it. You must also understand Python’s building blocks and use proven program-construction principles.

3.2 Algorithms

You can solve any computing problem by executing a series of actions in a specific order. An **algorithm** is a *procedure* for solving a problem in terms of:

1. the **actions** to execute, and
2. the **order** in which these actions execute.

Correctly specifying the order in which the actions execute is essential. Consider the “rise-and-shine algorithm” that an executive follows for getting out of bed and going to work: (1) Get out of bed; (2) take off pajamas; (3) take a shower; (4) get dressed; (5) eat breakfast; (6) carpool to work. This routine gets the executive to work well prepared to make critical decisions. Suppose the executive performs these steps in a different order: (1) Get out of bed; (2) take off pajamas; (3) get dressed; (4) take a shower; (5) eat breakfast; (6) carpool to work. Now, our executive shows up for work soaking wet. **Program control** specifies the order in which statements (actions) execute in a program. This chapter investigates program control using Python’s **control statements**.



Self Check

- I (Fill-In) A(n) _____ is a procedure for solving a problem. It specifies the _____ to execute and the _____ in which they execute.

Answer: algorithm, actions, order.

3.3 Pseudocode

Pseudocode is an informal English-like language for “thinking out” algorithms. You write text that describes what your program should do. You then convert the pseudocode to Python by replacing pseudocode statements with their Python equivalents.

Addition-Program Pseudocode

The following pseudocode algorithm prompts the user to enter two integers, inputs them from the user at the keyboard, adds them, then stores and displays their sum:

Prompt the user to enter the first integer

Input the first integer

Prompt the user to enter the second integer

Input the second integer

Add first integer and second integer, store their sum

Display the numbers and their sum

This is the complete pseudocode algorithm. Later in the chapter, we’ll show a simple process for creating a pseudocode algorithm from a *requirements statement*. The English pseudocode statements specify the actions you wish to perform and the order in which you wish to perform them.



Self Check

- 1 (*True/False*) Pseudocode is a simple programming language.

Answer: False. Pseudocode is not a programming language. It’s an artificial and informal language that helps you develop algorithms.

- 2 (*IPython Session*) Write Python statements that perform the tasks described by this section’s pseudocode. Enter the integers 10 and 5.

Answer:

```
In [1]: number1 = int(input('Enter first integer: '))
Enter first integer: 10

In [2]: number2 = int(input('Enter second integer: '))
Enter second integer: 5

In [3]: total = number1 + number2

In [4]: print('The sum of', number1, 'and', number2, 'is', total)
The sum of 10 and 5 is 15
```

3.4 Control Statements

Usually, statements in a program execute in the order in which they’re written. This is called *sequential execution*. Various Python statements enable you to specify that the next statement to execute may be *other than* the next one *in sequence*. This is called *transfer of control* and is achieved with Python *control statements*.

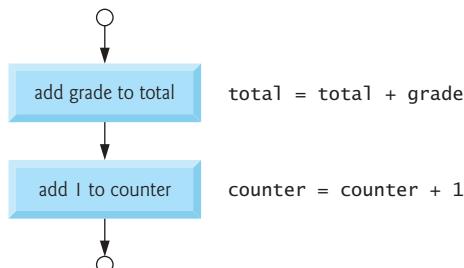
Forms of Control

In the 1960s, extensive use of control transfers was causing difficulty in software development. Blame was pointed at the `goto` statement. This statement allowed you to transfer control to one of many possible destinations in a program. Bohm and Jacopini's research¹ demonstrated that programs could be written without `goto` statements. The notion of *structured programming* became almost synonymous with "goto elimination." Python does not have a `goto` statement. Structured programs are clearer, easier to debug and change, and more likely to be bug-free.

Bohm and Jacopini demonstrated that all programs could be written using three forms of control—namely, **sequential execution**, the **selection statement** and the **repetition statement**. Sequential execution is simple. Python statements execute one after the other "in sequence," unless directed otherwise.

Flowcharts

A **flowchart** is a *graphical* representation of an algorithm or a part of one. You draw flowcharts using *rectangles*, *diamonds*, *rounded rectangles* and *small circles* that you connect by *arrows* called **flowlines**. Like pseudocode, flowcharts are useful for developing and representing algorithms. They clearly show how forms of control operate. Consider the following flowchart segment, which shows *sequential execution*:



We use the **rectangle (or action) symbol** to indicate any *action*, such as a calculation or an input/output operation. The flowlines show the *order* in which the actions execute. First, the grade is added to the total, then 1 is added to the counter. We show the Python code next to each action symbol for comparison purposes. This code is not part of the flowchart.

In a flowchart for a *complete* algorithm, the first symbol is a **rounded rectangle** containing the word "Begin." The last symbol is a rounded rectangle containing the word "End." In a flowchart for only a *part* of an algorithm, we omit the rounded rectangles, instead using small circles called **connector symbols**. The most important symbol is the **decision (or diamond) symbol**, which indicates that a *decision* is to be made, such as in an `if` statement. We begin using decision symbols in the next section.

Selection Statements

Python provides three types of selection statements that execute code based on a *condition*—an expression that evaluates to either `True` or `False`:

1. Bohm, C., and G. Jacopini, "Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules," *Communications of the ACM*, Vol. 9, No. 5, May 1966, pp. 336–371.

- The `if` statement performs an action if a condition is `True` or skips the action if the condition is `False`.
- The `if...else statement` performs an action if a condition is `True` or performs a *different* action if the condition is `False`.
- The `if...elif...else statement` performs one of many different actions, depending on the truth or falsity of *several* conditions.

Anywhere a single action can be placed, a group of actions can be placed.

The `if` statement is called a **single-selection statement** because it selects or ignores a *single* action (or group of actions). The `if...else` statement is called a **double-selection statement** because it selects between *two different* actions (or groups of actions). The `if...elif...else` statement is called a **multiple-selection statement** because it selects one of many different actions (or groups of actions).

Repetition Statements

Python provides two repetition statements—`while` and `for`:

- The `while` statement repeats an action (or a group of actions) as long as a condition remains `True`.
- The `for` statement repeats an action (or a group of actions) for every item in a sequence of items.

Keywords

The words `if`, `elif`, `else`, `while`, `for`, `True` and `False` are keywords that Python reserves to implement its features, such as control statements. Using a keyword as an identifier such as a variable name is a syntax error. The following table lists Python’s keywords.

Python keywords						
<code>and</code>	<code>as</code>	<code>assert</code>	<code>async</code>	<code>await</code>	<code>break</code>	<code>class</code>
<code>continue</code>	<code>def</code>	<code>del</code>	<code>elif</code>	<code>else</code>	<code>except</code>	<code>False</code>
<code>finally</code>	<code>for</code>	<code>from</code>	<code>global</code>	<code>if</code>	<code>import</code>	<code>in</code>
<code>is</code>	<code>lambda</code>	<code>None</code>	<code>nonlocal</code>	<code>not</code>	<code>or</code>	<code>pass</code>
<code>raise</code>	<code>return</code>	<code>True</code>	<code>try</code>	<code>while</code>	<code>with</code>	<code>yield</code>

Control Statements Summary

You form each Python program by combining as many control statements of each type as you need for the algorithm the program implements. With **Single-entry/single-exit** (one way in/one way out) **control statements**, the exit point of one connects to the entry point of the next. This is similar to the way a child stacks building blocks—hence, the term **control-statement stacking**. **Control-statement nesting** also connects control statements—we’ll see how later in the chapter.

You can construct any Python program from only six different forms of control (sequential execution, and the `if`, `if...else`, `if...elif...else`, `while` and `for` statements). You combine these in only two ways (control-statement stacking and control-statement nesting). This is the essence of simplicity.



Self Check

- 1 (*Fill-In*) You can write all programs using three forms of control—_____ , _____ and _____.

Answer: sequential execution, selection statements, repetition statements.

- 2 (*Fill-In*) A(n) _____ is a graphical representation of an algorithm.

Answer: flowchart.

3.5 if Statement

Suppose that a passing grade on an examination is 60. The pseudocode

```
If student's grade is greater than or equal to 60
    Display 'Passed'
```

determines whether the condition “student’s grade is greater than or equal to 60” is true or false. If the condition is true, ‘Passed’ is displayed. Then, the next pseudocode statement in order is “performed.” (Remember that pseudocode is not a real programming language.) If the condition is false, nothing is displayed, and the next pseudocode statement is “performed.” The pseudocode’s second line is indented. Python code requires indentation. Here it emphasizes that ‘Passed’ is displayed *only* if the condition is true.

Let’s assign 85 to the variable `grade`, then show and execute the Python `if` statement for the pseudocode:

```
In [1]: grade = 85

In [2]: if grade >= 60:
...:     print('Passed')
...:
Passed
```

The `if` statement closely resembles the pseudocode. The condition `grade >= 60` is True, so the indented `print` statement displays ‘Passed’.

Suite Indentation

Indenting a suite is required; otherwise, an `IndentationError` syntax error occurs:

```
In [3]: if grade >= 60:
...:     print('Passed') # statement is not indented properly
File "<ipython-input-3-f42783904220>", line 2
    print('Passed') # statement is not indented properly
          ^
IndentationError: expected an indented block
```

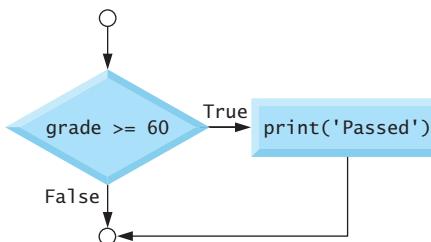
An `IndentationError` also occurs if you have more than one statement in a suite and those statements do not have the *same* indentation:

```
In [4]: if grade >= 60:
...:     print('Passed') # indented 4 spaces
...:     print('Good job!') # incorrectly indented only two spaces
File <ipython-input-4-8c0d75c127bf>, line 3
    print('Good job!') # incorrectly indented only two spaces
          ^
IndentationError: unindent does not match any outer indentation level
```

Sometimes error messages may not be clear. The fact that Python calls attention to the line is usually enough for you to figure out what's wrong. Apply indentation conventions uniformly throughout your code. Programs that are not uniformly indented are hard to read.

if Statement Flowchart

The flowchart for the `if` statement in snippet [2] is:



The decision (diamond) symbol contains a condition that can be either `True` or `False`. The diamond has two flowlines emerging from it:

- One indicates the direction to follow when the condition in the symbol is `True`. This points to the action (or group of actions) that should execute.
- The other indicates the direction to follow when the condition is `False`. This skips the action (or group of actions).

Every Expression Can Be Interpreted as Either True or False

You can base decisions on *any* expression. A nonzero value is `True`. Zero is `False`:

```

In [5]: if 1:
    ...:     print('Nonzero values are true, so this will print')
    ...:
Nonzero values are true, so this will print

In [6]: if 0:
    ...:     print('Zero is false, so this will not print')

In [7]:
  
```

Strings containing characters are `True` and empty strings ('', "" or "") are `False`.

An Additional Note on Confusing == and =

Using the equality operator `==` instead of the assignment symbol `=` in an assignment statement can lead to subtle problems. For example, in this session, snippet [1] defined `grade` with the assignment:

```
grade = 85
```

If instead we accidentally wrote:

```
grade == 85
```

then `grade` would be undefined and we'd get a `NameError`.

If `grade` had been defined before the preceding statement, then `grade == 85` would evaluate to `True` or `False`, depending on `grade`'s value, and not perform the intended assignment. This is a logic error.



Self Check

1 (*True/False*) If you indent a suite's statements, you will not get an `IndentationError`.
Answer: False. All the statements in a suite must have the *same* indentation. Otherwise, an `IndentationError` occurs.

2 (*IPython Session*) Redo this section's snippets [1] and [2], then change `grade` to 55 and repeat the `if` statement to show that its suite does not execute. The next section shows how to recall and re-execute earlier snippets to avoid having to re-enter the code.

Answer:

```
In [1]: grade = 85

In [2]: if grade >= 60:
...:     print('Passed')
...:
Passed

In [3]: grade = 55

In [4]: if grade >= 60:
...:     print('Passed')
...:

In [5]:
```

3.6 `if...else` and `if...elif...else` Statements

The `if...else` statement performs different suites, based on whether a condition is `True` or `False`. The pseudocode below displays 'Passed' if the student's grade is greater than or equal to 60; otherwise, it displays 'Failed':

```
If student's grade is greater than or equal to 60
    Display 'Passed'
Else
    Display 'Failed'
```

In either case, the next pseudocode statement in sequence after the entire *If...Else* is “performed.” We indent both the *If* and *Else* suites, and by the same amount. Let’s create and **initialize** (that is, give a starting value to) the variable `grade`, then show and execute the Python `if...else` statement for the preceding pseudocode:

```
In [1]: grade = 85

In [2]: if grade >= 60:
...:     print('Passed')
...: else:
...:     print('Failed')
...:
Passed
```

The condition above is `True`, so the `if` suite displays 'Passed'. Note that when you press *Enter* after typing `print('Passed')`, IPython indents the next line four spaces. You must delete those four spaces so that the `else:` suite correctly aligns under the `i` in `if`.

The following code assigns 57 to the variable `grade`, then shows the `if...else` statement again to demonstrate that only the `else` suite executes when the condition is `False`:

```
In [3]: grade = 57

In [4]: if grade >= 60:
....:     print('Passed')
....: else:
....:     print('Failed')
....:

Failed
```

The up and down arrow keys navigate backwards and forwards through the current interactive session's snippets. Pressing *Enter* re-executes the snippet that's displayed. Let's set `grade` to 99, press the up arrow key twice to recall the code from snippet [4], then press *Enter* to re-execute that code as snippet [6]. Every recalled snippet that you execute gets a new ID:

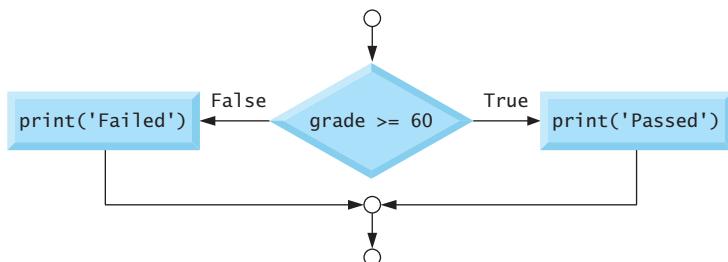
```
In [5]: grade = 99

In [6]: if grade >= 60:
....:     print('Passed')
....: else:
....:     print('Failed')
....:

Passed
```

if...else Statement Flowchart

The flowchart below shows the preceding `if...else` statement's flow of control:



Conditional Expressions

Sometimes the suites in an `if...else` statement assign different values to a variable, based on a condition, as in:

```
In [7]: grade = 87

In [8]: if grade >= 60:
....:     result = 'Passed'
....: else:
....:     result = 'Failed'
....:
```

We can then print or evaluate that variable:

```
In [9]: result
Out[9]: 'Passed'
```

You can write statements like snippet [8] using a concise **conditional expression**:

```
In [10]: result = ('Passed' if grade >= 60 else 'Failed')
```

```
In [11]: result
Out[11]: 'Passed'
```

The parentheses are not required, but they make it clear that the statement assigns the conditional expression's value to `result`. First, Python evaluates the condition `grade >= 60`:

- If it's True, snippet [10] assigns to `result` the value of the expression to the *left* of `if`, namely 'Passed'. The `else` part does not execute.
- If it's False, snippet [10] assigns to `result` the value of the expression to the *right* of `else`, namely 'Failed'.

In interactive mode, you also can evaluate the conditional expression directly, as in:

```
In [12]: 'Passed' if grade >= 60 else 'Failed'
Out[12]: 'Passed'
```

Multiple Statements in a Suite

The following code shows two statements in the `else` suite of an `if...else` statement:

```
In [13]: grade = 49

In [14]: if grade >= 60:
    ...:     print('Passed')
    ...: else:
    ...:     print('Failed')
    ...:     print('You must take this course again')
    ...:

Failed
You must take this course again
```

In this case, `grade` is less than 60, so *both* statements in the `else`'s suite execute. If you do not indent the second `print`, then it's not in the `else`'s suite. So, that statement *always* executes, creating strange incorrect output:

```
In [15]: grade = 100

In [16]: if grade >= 60:
    ...:     print('Passed')
    ...: else:
    ...:     print('Failed')
    ...: print('You must take this course again')
    ...:

Passed
You must take this course again
```

if...elif...else Statement

You can test for many cases using the **if...elif...else statement**. The following pseudocode displays “A” for grades greater than or equal to 90, “B” for grades in the range 80–89, “C” for grades 70–79, “D” for grades 60–69 and “F” for all other grades:

```
If student's grade is greater than or equal to 90
    Display "A"
Else If student's grade is greater than or equal to 80
    Display "B"
Else If student's grade is greater than or equal to 70
    Display "C"
Else If student's grade is greater than or equal to 60
    Display "D"
Else
    Display "F"
```

Only the action for the first True condition executes. Let’s show and execute the Python code for the preceding pseudocode. The pseudocode *Else If* is written with the keyword `elif`. Snippet [18] displays C, because `grade` is 77:

```
In [17]: grade = 77
```

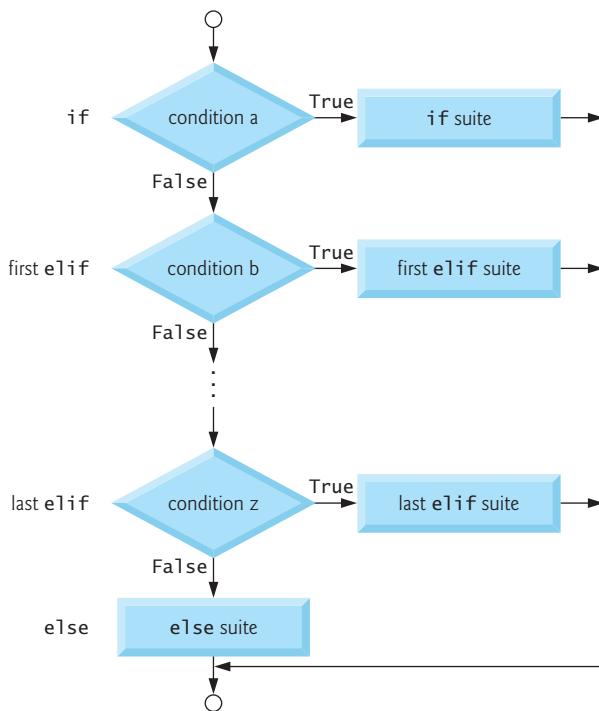
```
In [18]: if grade >= 90:
...:     print('A')
...: elif grade >= 80:
...:     print('B')
...: elif grade >= 70:
...:     print('C')
...: elif grade >= 60:
...:     print('D')
...: else:
...:     print('F')
...:
```

```
C
```

The first condition—`grade >= 90`—is False, so `print('A')` is skipped. The second condition—`grade >= 80`—also is False, so `print('B')` is skipped. The third condition—`grade >= 70`—is True, so `print('C')` executes. Then all the remaining code in the `if...elif...else` statement is skipped. An `if...elif...else` is faster than separate `if` statements, because condition testing stops as soon as a condition is True.

if...elif...else Statement Flowchart

The following flowchart shows the general flow through an `if...elif...else` statement. It shows that, after any suite executes, control immediately exits the statement. The words to the left are not part of the flowchart. We added them to show how the flowchart corresponds to the equivalent Python code.



else Is Optional

The `else` in the `if...elif...else` statement is optional. Including it enables you to handle values that do not satisfy *any* of the conditions. When an `if...elif` statement without an `else` tests a value that does not make any of its conditions True, the program does not execute any of the statement's suites. The next statement in sequence after the `if...elif` statement executes. If you specify the `else`, you must place it after the last `elif`; otherwise, a `SyntaxError` occurs.

Logic Errors

The incorrectly indented code segment in snippet [16] is an example of a **nonfatal logic error**. The code executes, but it produces incorrect results. For a **fatal logic error** in a script, an exception occurs (such as a `ZeroDivisionError` from an attempt to divide by 0), so Python displays a traceback, then terminates the script. A fatal error in interactive mode terminates only the current snippet. Then IPython waits for your next input.



Self Check

- 1 (True/False)** A fatal logic error causes a script to produce incorrect results, then continue executing.

Answer: False. A fatal logic error causes a script to terminate.

- 2 (IPython Session)** Show that a `SyntaxError` occurs if an `if...elif` statement specifies an `else` before the last `elif`.

Answer:

```
In [1]: grade = 80

In [2]: if grade >= 90:
...:     print('A')
...: else:
...:     print('Not A or B')
...: elif grade >= 80:
...:     elif grade >= 80:
...:         ^
SyntaxError: invalid syntax
```

3.7 while Statement

The **while statement** allows you to *repeat* one or more actions while a condition remains *True*. Such a statement often is called a **loop**.

The following pseudocode specifies what happens when you go shopping:

*While there are more items on my shopping list
Buy next item and cross it off my list*

If the condition “there are more items on my shopping list” is *true*, you perform the action “Buy next item and cross it off my list.” You *repeat* this action while the condition remains *true*. You stop repeating this action when the condition becomes *false*—that is, when you’ve crossed all items off your shopping list.

Let’s use a **while** statement to find the first power of 3 larger than 50:

```
In [1]: product = 3

In [2]: while product <= 50:
...:     product = product * 3
...:

In [3]: product
Out[3]: 81
```

First, we create **product** and initialize it to 3. Then the **while** statement executes as follows:

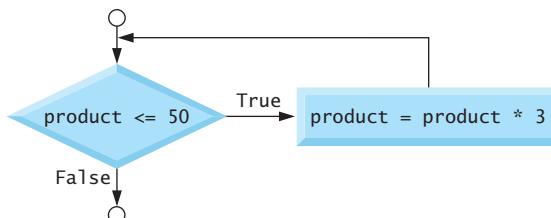
1. Python tests the condition **product <= 50**, which is *True* because **product** is 3. The statement in the suite multiplies **product** by 3 and assigns the result (9) to **product**. One *iteration of the loop* is now complete.
2. Python again tests the condition, which is *True* because **product** is now 9. The suite’s statement sets **product** to 27, completing the second iteration of the loop.
3. Python again tests the condition, which is *True* because **product** is now 27. The suite’s statement sets **product** to 81, completing the third iteration of the loop.
4. Python again tests the condition, which is finally *False* because **product** is now 81. The repetition now terminates.

Snippet [3] evaluates **product** to see its value, 81, which is the first power of 3 larger than 50. If this **while** statement were part of a larger script, execution would continue with the next statement in sequence after the **while**.

Something in the `while` statement's suite must change `product`'s value, so the condition eventually becomes `False`. Otherwise, a logic error called an **infinite loop** occurs. Such an error prevents the `while` statement from ever terminating—the program appears to “hang.” In applications executed from a Terminal, Command Prompt or shell, type `Ctrl + c` or `control + c` (depending on your keyboard) to terminate an infinite loop. IDEs typically have a toolbar button or menu option for stopping a program’s execution.

while Statement Flowchart

The following flowchart shows the preceding `while` statement’s flow of control:



Follow the flowlines to experience the repetition. The flowline from the rectangle “closes the loop” by flowing back into the condition `product <= 50` that’s tested during each iteration. When that condition becomes `False`, the `while` statement exits and control proceeds to the next statement in sequence.



Self Check

1 (*True/False*) A `while` statement performs its suite while some condition remains `True`.
Answer: True.

2 (*IPython Session*) Write statements to determine the first power of 7 greater than 1000.
Answer:

```

In [1]: product = 7

In [2]: while product <= 1000:
...:     product = product * 7
...:

In [3]: product
Out[3]: 2401
  
```

3.8 for Statement

Like the `while` statement, the **for statement** allows you to *repeat* an action or several actions. The `for` statement performs its action(s) for each item in a **sequence** of items. For example, a string is a sequence of individual characters. Let’s display ‘Programming’ with its characters separated by two spaces:

```

In [1]: for character in 'Programming':
...:     print(character, end=' ')
...:

P r o g r a m m i n g
  
```

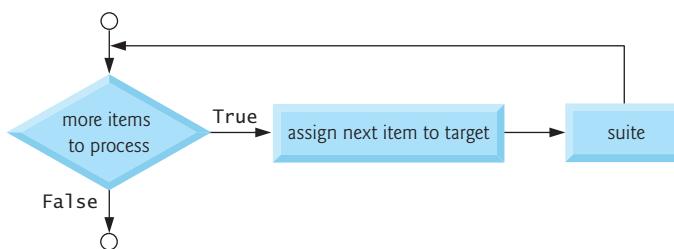
The `for` statement executes as follows:

- Upon entering the statement, it assigns the 'P' in 'Programming' to the **target** variable between keywords `for` and `in`—in this case, character.
- Next, the statement in the suite executes, displaying character's value followed by two spaces—we'll say more about this momentarily.
- After executing the suite, Python assigns to `character` the next item in the sequence (that is, the 'r' in 'Programming'), then executes the suite again.
- This continues while there are more items in the sequence to process. In this case, the statement terminates after displaying the letter 'g', followed by two spaces.

Using the target in the suite, as we did here to display its value, is common but not required.

For Statement Flowchart

The `for` statement's flowchart is similar to that of the `while` statement:



First, Python determines whether there are more items to process. If so, the `for` statement assigns the next item to the target, then performs the suite's action(s).

Function print's end Keyword Argument

The built-in function `print` displays its argument(s), then moves the cursor to the next line. You can change this behavior with the argument **end**, as in

```
print(character, end=' ')
```

We used two spaces (' '), so each call to `print` displays `character`'s value followed by two spaces. So, all the characters display horizontally on the *same* line. Python calls `end` a **keyword argument**, but `end` is not a Python keyword. The `end` keyword argument is *optional*. If you do not include it, `print` uses a newline ('\n') by default. The *Style Guide for Python Code* recommends placing no spaces around a keyword argument's `=`. Keyword arguments are sometimes called named arguments.

Function print's sep Keyword Argument

You can use the keyword argument **sep** (short for separator) to specify the string that appears *between* the items that `print` displays. When you do not specify this argument, `print` uses a space character by default. Let's display three numbers, each separated from the next by a comma and a space, rather than just a space:

```
In [2]: print(10, 20, 30, sep=', ')
10, 20, 30
```

To remove the spaces, use an **empty string** with no characters between its quotes.

3.8.1 Iterables, Lists and Iterators

The sequence to the right of the `for` statement's `in` keyword must be an **iterable**. An iterable is an object from which the `for` statement can take one item at a time until no more items remain. Python has other iterable sequence types besides strings. One of the most common is a **list**, which is a comma-separated collection of items enclosed in square brackets ([and]). The following code totals five integers in a list:

```
In [3]: total = 0
In [4]: for number in [2, -3, 0, 17, 9]:
...:     total = total + number
...:
In [5]: total
Out[5]: 25
```

Each sequence has an **iterator**. The `for` statement uses the iterator “behind the scenes” to get each consecutive item until there are no more to process. The iterator is like a bookmark—it always knows where it is in the sequence, so it can return the next item when it's called upon to do so.

For each `number` in the list, the suite adds the `number` to the `total`. When there are no more items to process, `total` contains the sum (25) of the list's items. We cover lists in detail in the “Sequences: Lists and Tuples” chapter. There, you'll see that the order of the items in a list matters and that a list's items are **mutable** (that is, modifiable).

3.8.2 Built-In range Function

Let's use a `for` statement and the built-in **range function** to iterate precisely 10 times, displaying the values from 0 through 9:

```
In [6]: for counter in range(10):
...:     print(counter, end=' ')
...:
0 1 2 3 4 5 6 7 8 9
```

The function call `range(10)` creates an iterable object that represents a sequence of consecutive integer values starting from 0 and continuing up to, but *not* including, the argument value (10). In this case, the sequence is 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. The `for` statement exits when it finishes processing the last integer that `range` produces. Iterators and iterable objects are two of Python's *functional-style programming* features. We'll introduce more of these throughout the book.

Off-By-One Errors

A logic error known as an **off-by-one error** occurs when you assume that `range`'s argument value is included in the generated sequence. For example, if you provide 9 as `range`'s argument when trying to produce the sequence 0 through 9, `range` generates only 0 through 8.



Self Check

I (Fill-In) Function _____ generates a sequence of integers.

Answer: `range`.

2 (IPython Session) Use the `range` function and a `for` statement to calculate the total of the integers from 0 through 1,000,000.

Answer:

```
In [1]: total = 0

In [2]: for number in range(1000001):
...:     total = total + number
...:

In [3]: total
Out[3]: 500000500000
```

3.9 Augmented Assignments

Augmented assignments abbreviate assignment expressions in which the same variable name appears on the left and right of the assignment's `=`, as `total` does in:

```
for number in [1, 2, 3, 4, 5]:
    total = total + number
```

Snippet [2] reimplements this using an **addition augmented assignment (`+=`) statement**:

```
In [1]: total = 0

In [2]: for number in [1, 2, 3, 4, 5]:
...:     total += number # add number to total and store in number
...:

In [3]: total
Out[3]: 15
```

The `+=` expression in snippet [2] first adds `number`'s value to the current `total`, then stores the new value in `total`. The table below shows sample augmented assignments:

Augmented assignment	Sample expression	Explanation	Assigns
<i>Assume: c = 3, d = 5, e = 4, f = 2, g = 9, h = 12</i>			
<code>+=</code>	<code>c += 7</code>	<code>c = c + 7</code>	10 to c
<code>-=</code>	<code>d -= 4</code>	<code>d = d - 4</code>	1 to d
<code>*=</code>	<code>e *= 5</code>	<code>e = e * 5</code>	20 to e
<code>**=</code>	<code>f **= 3</code>	<code>f = f ** 3</code>	8 to f
<code>/=</code>	<code>g /= 2</code>	<code>g = g / 2</code>	4.5 to g
<code>//=</code>	<code>g //=- 2</code>	<code>g = g // 2</code>	4 to g
<code>%=</code>	<code>h %= 9</code>	<code>h = h % 9</code>	3 to h



Self Check

1 (*Fill-In*) If x is 7, the value of x after evaluating $x *= 5$ is _____.

Answer: 35.

2 (*IPython Session*) Create a variable x with the value 12. Use an exponentiation augmented assignment statement to square x 's value. Show x 's new value.

Answer:

```
In [1]: x = 12
```

```
In [2]: x **= 2
```

```
In [3]: x
```

```
Out[3]: 144
```

3.10 Program Development: Sequence-Controlled Repetition

Experience has shown that the most challenging part of solving a problem on a computer is developing an algorithm for the solution. As you'll see, once a correct algorithm has been specified, creating a working Python program from the algorithm is typically straightforward. This section and the next present problem solving and program development by creating scripts that solve two class-averaging problems.

3.10.1 Requirements Statement

A **requirements statement** describes *what* a program is supposed to do, but not *how* the program should do it. Consider the following simple requirements statement:

A class of ten students took a quiz. Their grades (integers in the range 0 – 100) are 98, 76, 71, 87, 83, 90, 57, 79, 82, 94. Determine the class average on the quiz.

Once you know the problem's requirements, you can begin creating an algorithm to solve it. Then, you can implement that solution as a program.

The algorithm for solving this problem must:

1. Keep a running total of the grades.
2. Calculate the average—the total of the grades divided by the number of grades.
3. Display the result.

For this example, we'll place the 10 grades in a list. You also could input the grades from a user at the keyboard (as we'll do in the next example) or read them from a file (as you'll see how to do in the "Files and Exceptions" chapter). We also show you how to read data from SQL and NoSQL databases in later chapters.

3.10.2 Pseudocode for the Algorithm

The following pseudocode lists the actions to execute and specifies the order in which they should execute:

*Set total to zero
Set grade counter to zero
Set grades to a list of the ten grades*

*For each grade in the grades list:
 Add the grade to the total
 Add one to the grade counter*

*Set the class average to the total divided by the number of grades
Display the class average*

Note the mentions of *total* and *grade counter*. In Fig. 3.1's script, the variable `total` (line 5) stores the grade values' running total, and `grade_counter` (line 6) counts the number of grades we've processed. We'll use these to calculate the average. Variables for totaling and counting normally are initialized to zero before they're used, as we do in lines 5 and 6.

3.10.3 Coding the Algorithm in Python

The following script implements the pseudocode algorithm.

```

1 # fig03_01.py
2 """Class average program with sequence-controlled repetition."""
3
4 # initialization phase
5 total = 0 # sum of grades
6 grade_counter = 0
7 grades = [98, 76, 71, 87, 83, 90, 57, 79, 82, 94] # list of 10 grades
8
9 # processing phase
10 for grade in grades:
11     total += grade # add current grade to the running total
12     grade_counter += 1 # indicate that one more grade was processed
13
14 # termination phase
15 average = total / grade_counter
16 print(f'Class average is {average}')

```

Class average is 81.7

Fig. 3.1 | Class average program with sequence-controlled repetition.

Execution Phases

We used blank lines and comments to break the script into three **execution phases**—initialization, processing and termination:

- The **initialization phase** creates the variables needed to process the grades and set these variables to appropriate initial values.
- The **processing phase** processes the grades, calculating the running total and counting the number of grades processed so far.
- The **termination phase** calculates and displays the class average.

Many scripts can be **decomposed** (that is, broken apart) into these three phases.

Initialization Phase

Lines 5–6 create the variables `total` and `grade_counter` and initialize each to 0. Line 7

```
grades = [98, 76, 71, 87, 83, 90, 57, 79, 82, 94] # list of 10 grades
```

creates the variable `grades` and initializes it with a list of 10 integer grades.

Processing Phase

The `for` statement processes each grade in the list `grades`. Line 11 adds the current grade to the `total`. Then, line 12 adds 1 to the variable `grade_counter` to keep track of the number of grades processed so far. Repetition terminates when all 10 grades in the list have been processed. This is called **definite repetition** because the number of repetitions is known before the loop begins executing. In this case, it's the number of elements in the list `grades`. The *Style Guide for Python Code* recommends placing a blank line above and below each control statement (as in lines 8 and 13).

Termination Phase

When the `for` statement terminates, line 15 calculates the average and assigns it to the variable `average`. Then line 16 displays `average`. Later in this chapter, we use functional-style programming features to calculate the average of a list's items more concisely.

3.10.4 Introduction to Formatted Strings

Line 16 uses the following simple **f-string** (short for **formatted string**) to format this script's result by inserting the value of `average` into a string:

```
f'Class average is {average}'
```

The letter `f` before the string's opening quote indicates it's an f-string. You specify where to insert values by using placeholders delimited by curly braces (`{` and `}`). The placeholder `{average}`

converts the variable `average`'s value to a string representation, then replaces `{average}` with that **replacement text**. Replacement-text expressions may contain values, variables or other expressions, such as calculations or function calls. In line 16, we could have used `total / grade_counter` in place of `average`, eliminating the need for line 15.



Self Check

- 1 (Fill-In)** A(n) _____ describes *what* a program is supposed to do, but not *how* the program should do it.

Answer: requirements statement.

- 2 (Fill-In)** Many of the scripts you'll write can be decomposed into three phases: _____, _____ and _____.

Answer: initialization, processing, termination.

- 3 (IPython Session)** Display an f-string in which you insert the values of the variables `number1` (7) and `number2` (5) and their product. The displayed string should be

7 times 5 is 35

Answer:

```
In [1]: number1 = 7
In [2]: number2 = 5
In [3]: print(f'{number1} times {number2} is {number1 * number2}')
7 times 5 is 35
```

3.11 Program Development: Sentinel-Controlled Repetition

Let's generalize the class-average problem. Consider the following requirements statement:

Develop a class-averaging program that processes an arbitrary number of grades each time the program executes.

In the first class-average example, we knew in advance the 10 grades to process. The requirements statement does not state what the grades are or how many there are, so we're going to have the user enter the grades into the program. The program processes an arbitrary number of grades. How can the program determine when to stop processing grades so that it can move on to calculate and display the class average?

One way to solve this problem is to use a special value called a **sentinel value** (also called a **signal value**, a **dummy value** or a **flag value**) to indicate "end of data entry." This is a bit like the way a caboose "marks" the end of a train. The user enters grades one at a time until all the grades have been entered. The user then enters the sentinel value to indicate that there are no more grades. **Sentinel-controlled repetition** is often called **indefinite repetition** because the number of repetitions is *not* known before the loop begins executing.

A sentinel value must not be confused with any acceptable input value. Grades on a quiz are typically nonnegative integers between 0 and 100, so the value -1 is an acceptable sentinel value for this problem. Thus, a run of the class-average program might process a stream of inputs such as 95, 96, 75, 74, 89 and -1 . The program would then compute and print the class average for the grades 95, 96, 75, 74 and 89. The sentinel value -1 should *not* enter into the averaging calculation.

Developing the Pseudocode Algorithm with Top-Down, Stepwise Refinement
 We approach this class-average problem with a technique called **top-down, stepwise refinement**. We begin with a pseudocode representation of the top:

Determine the class average for the quiz

The top is a single statement that conveys the program's overall function. Although it's a *complete* representation of a program, the top rarely conveys enough detail from which to write a program. The *top* specifies *what* should be done, but not *how* to implement it. So we begin the **refinement process**. We decompose the top into a sequence of smaller tasks—a process sometimes called **divide and conquer**. This results in the following **first refinement**:

*Initialize variables
 Input, sum and count the quiz grades
 Calculate and display the class average*

Each refinement represents the *complete* algorithm—only the level of detail varies. In this refinement, the three pseudocode statements happen to correspond to the three execution phases described in the preceding section. The algorithm does not yet provide enough detail for us to write the Python program. So, we continue with the next refinement.

Second Refinement

To proceed to the **second refinement**, we commit to specific variables. The program needs to maintain

- a grade variable in which each successive user input will be stored,
- a running total of the grades,
- a count of how many grades have been processed and
- a variable that contains the calculated average.

The pseudocode statement

Initialize variables

can be refined as follows:

Initialize total to zero

Initialize grade counter to zero

Only the variables *total* and *grade counter* need to be initialized before they’re used. We do not initialize the variables for the user input and calculated average. Their values will be replaced each time we input a grade from the user and when we calculate the class average, respectively. We’ll create these variables when they’re needed.

The next pseudocode statement requires a loop that successively inputs each grade:

Input, sum and count the quiz grades

We do not know how many grades will be entered, so we use sentinel-controlled repetition. The user enters legitimate grades successively. After the last legitimate grade has been entered, the user enters the sentinel value. The program tests for the sentinel value after each grade is input and terminates the loop when the sentinel has been entered. The second refinement of the preceding pseudocode statement is

Input the first grade (possibly the sentinel)

While the user has not entered the sentinel

Add this grade into the running total

Add one to the grade counter

Input the next grade (possibly the sentinel)

The pseudocode statement

Calculate and display the class average

can be refined as follows:

If the counter is not equal to zero

Set the average to the total divided by the grade counter

Display the average

Else

Display “No grades were entered”

Notice that we're testing for the possibility of division by zero. If undetected, this would cause a fatal logic error. In the “Files and Exceptions” chapter, we discuss how to write programs that recognize such exceptions and take appropriate actions.

The following is the class-average problem's complete second refinement:

```

Initialize total to zero
Initialize grade counter to zero

Input the first grade (possibly the sentinel)
While the user has not entered the sentinel
    Add this grade into the running total
    Add one to the grade counter
    Input the next grade (possibly the sentinel)

If the counter is not equal to zero
    Set the average to the total divided by the counter
    Display the average
Else
    Display "No grades were entered"
```

Sometimes more than two refinements are necessary. You stop refining when there is enough detail for you to convert the pseudocode to Python. We include blank lines for readability. Here, they happen to separate the algorithm into the three popular execution phases.

Implementing Sentinel-Controlled Iteration

The following script implements the pseudocode algorithm and shows a sample execution in which the user enters three grades and the sentinel value.

```

1 # fig03_02.py
2 """Class average program with sentinel-controlled iteration."""
3
4 # initialization phase
5 total = 0 # sum of grades
6 grade_counter = 0 # number of grades entered
7
8 # processing phase
9 grade = int(input('Enter grade, -1 to end: ')) # get one grade
10
11 while grade != -1:
12     total += grade
13     grade_counter += 1
14     grade = int(input('Enter grade, -1 to end: '))
15
16 # termination phase
17 if grade_counter != 0:
18     average = total / grade_counter
19     print(f'Class average is {average:.2f}')
20 else:
21     print('No grades were entered')
```

Fig. 3.2 | Class average program with sentinel-controlled iteration. (Part I of 2.)

```

Enter grade, -1 to end: 97
Enter grade, -1 to end: 88
Enter grade, -1 to end: 72
Enter grade, -1 to end: -1
Class average is 85.67

```

Fig. 3.2 | Class average program with sentinel-controlled iteration. (Part 2 of 2.)

Program Logic for Sentinel-Controlled Repetition

In sentinel-controlled repetition, the program reads the first value (line 9) before reaching the `while` statement. Line 9 demonstrates why we did not create the variable `grade` until we needed it in the program. If we had initialized it, that value would have been replaced immediately by this assignment.

The value input in line 9 determines whether the program's flow of control should enter the `while`'s suite (lines 12–14). If the condition in line 11 is `False`, the user entered the sentinel value (`-1`), so the suite does not execute because the user did not enter any grades. If the condition is `True`, the suite executes, adding the `grade` value to the `total` and incrementing the `grade_counter`. Next, line 14 inputs another grade from the user. Then, the `while`'s condition (line 11) is tested again, using the most recent `grade` entered by the user. The value of `grade` is always input immediately before the program tests the `while` condition, so we can determine whether the value just input is the sentinel before processing that value as a grade. When the sentinel value is input, the loop terminates, and the program does not add `-1` to the total. In a sentinel-controlled loop that performs user input, any prompts (lines 9 and 14) should remind the user of the sentinel value.

After the loop terminates, the `if...else` statement (lines 17–21) executes. Line 17 determines whether the user entered any grades. If not, the `else` part (lines 20–21) executes and displays the message 'No grades were entered' and the program terminates.

Formatting the Class Average with Two Decimal Places

This example formatted the class average with two digits to the right of the decimal point. In an f-string, you can optionally follow a replacement-text expression with a colon (:) and a **format specifier** that describes how to format the replacement text. The format specifier `.2f` (line 19) formats the average as a floating-point number (f) with two digits to the right of the decimal point (.2). In this example, the sum of the grades was 257, which, when divided by 3, yields 85.66666666.... Formatting the average with `.2f` rounds it to the hundredths position, producing the replacement text `85.67`. An average with only one digit to the right of the decimal point would be formatted with a **trailing zero** (e.g., `85.50`). The chapter "Strings: A Deeper Look" discusses many string-formatting features.

Control-Statement Stacking

In this example, notice that control statements are stacked in sequence. The `while` statement (lines 11–14) is followed immediately by an `if...else` statement (lines 17–21).



Self Check

- I (Fill-In) Sentinel-controlled repetition is called _____ because the number of repetitions is not known before the loop begins executing.

Answer: indefinite repetition.

- 2** (*True/False*) Sentinel-controlled repetition uses a counter variable to control the number of times a set of instructions executes.

Answer: False. Sentinel-control repetition terminates repetition when the sentinel value is encountered.

3.12 Program Development: Nested Control Statements

Let's work through another complete problem. Once again, we plan the algorithm using pseudocode and top-down, stepwise refinement and we develop a corresponding Python script. Consider the following requirements statement:

A college offers a course that prepares students for the state licensing exam for real-estate brokers. Last year, several of the students who completed this course took the licensing examination. The college wants to know how well its students did on the exam. You have been asked to write a program to summarize the results. You have been given a list of these 10 students. Next to each name is written a 1 if the student passed the exam and a 2 if the student failed.

Your program should analyze the results of the exam as follows:

1. *Input each test result (i.e., a 1 or a 2). Display the message "Enter result" each time the program requests another test result.*
2. *Count the number of test results of each type.*
3. *Display a summary of the test results indicating the number of students who passed and the number of students who failed.*
4. *If more than eight students passed the exam, display "Bonus to instructor."*

After reading the requirements statement carefully, we make the following observations about the problem:

1. The program must process 10 test results. We'll use a `for` statement and the `range` function to control repetition.
2. Each test result is a number—either a 1 or a 2. Each time the program reads a test result, the program must determine if the number is a 1 or a 2. We test for a 1 in our algorithm. If the number is not a 1, we assume that it's a 2. (An exercise at the end of the chapter considers the consequences of this assumption.)
3. We'll use two counters—one to count the number of students who passed the exam and one to count the number of students who failed.
4. After the script processes all the results, it must decide if more than eight students passed the exam so that it can bonus the instructor.

Top-Down, Stepwise Refinement

We begin with a pseudocode representation of the top:

Analyze exam results and decide whether instructor should receive a bonus

Once again, the top is a *complete* representation of the program, but several refinements are likely to be needed before the pseudocode can evolve naturally into a Python program.

First Refinement

Our first refinement is

Initialize variables

Input the ten exam grades and count passes and failures

Summarize the exam results and decide whether instructor should receive a bonus

Here, too, even though we have a *complete* representation of the entire program, further refinement is necessary. Note again that this first refinement happens to correspond to the three-execution-phases model.

Second Refinement

We now commit to specific variables. We need counters to record the passes and failures, and a variable to store the user input. The pseudocode statement

Initialize variables

can be refined as follows:

Initialize passes to zero

Initialize failures to zero

Only the counters for the number of passes and number of failures need to be initialized.

The pseudocode statement

Input the ten exam grades and count passes and failures

requires a loop that successively inputs the result of each exam. Here it's known in advance that there are ten exam results, so the `for` statement and the `range` function are appropriate. Inside the loop (that is, *nested* within the loop), an `if...else` statement determines whether each exam result is a pass or a failure and increments the appropriate counter. The refinement of the preceding pseudocode statement is

For each of the ten students

Input the next exam result

If the student passed

Add one to passes

Else

Add one to failures

The blank line before the *If...Else* improves readability.

The pseudocode statement

Summarize the exam results and decide whether instructor should receive a bonus

may be refined as follows:

Display the number of passes

Display the number of failures

If more than eight students passed

Display "Bonus to instructor"

Complete Pseudocode Algorithm

The pseudocode is now sufficiently refined for conversion to Python—the complete second refinement is shown below:

```
Initialize passes to zero
Initialize failures to zero

For each of the ten students
    Input the next exam result

    If the student passed
        Add one to passes
    Else
        Add one to failures

Display the number of passes
Display the number of failures

If more than eight students passed
    Display "Bonus to instructor"
```

Implementing the Algorithm

The following script implements the algorithm and is followed by two sample executions. Once again, notice that the Python code closely resembles the pseudocode. Lines 9–16 loop 10 times, inputting and processing one exam result each time. The `if...else` statement (lines 13–16) that processes each result is *nested* in the `for` statement—that is, it's part of the `for` statement's suite. If the `result` is 1, we add 1 to `passes`; otherwise, we assume the `result` is 2 and add 1 to `failures`. After inputting 10 values, the loop terminates and lines 19 and 20 display `passes` and `failures`. Lines 22–23 determine whether more than eight students passed the exam and, if so, display 'Bonus to instructor'.

```
1 # fig03_03.py
2 """Using nested control statements to analyze examination results."""
3
4 # initialize variables
5 passes = 0 # number of passes
6 failures = 0 # number of failures
7
8 # process 10 students
9 for student in range(10):
10     # get one exam result
11     result = int(input('Enter result (1=pass, 2=fail): '))
12
13     if result == 1:
14         passes = passes + 1
15     else:
16         failures = failures + 1
17
```

Fig. 3.3 | Analysis of examination results. (Part I of 2.)

```

18 # termination phase
19 print('Passed:', passes)
20 print('Failed:', failures)
21
22 if passes > 8:
23     print('Bonus to instructor')

```

```

Enter result (1=pass, 2=fail): 1
Enter result (1=pass, 2=fail): 2
Enter result (1=pass, 2=fail): 2
Enter result (1=pass, 2=fail): 1
Enter result (1=pass, 2=fail): 1
Enter result (1=pass, 2=fail): 1
Enter result (1=pass, 2=fail): 2
Enter result (1=pass, 2=fail): 1
Enter result (1=pass, 2=fail): 1
Enter result (1=pass, 2=fail): 2
Passed: 6
Failed: 4

```

```

Enter result (1=pass, 2=fail): 1
Enter result (1=pass, 2=fail): 2
Enter result (1=pass, 2=fail): 1
Passed: 9
Failed: 1
Bonus to instructor

```

Fig. 3.3 | Analysis of examination results. (Part 2 of 2.)

✓ Self Check

I (*IPython Session*) Use a `for` statement to input two integers. Use a nested `if...else` statement to display whether each value is even or odd. Enter 10 and 7 to test your code.

Answer:

```

In [1]: for count in range(2):
...:     value = int(input('Enter an integer: '))
...:     if value % 2 == 0:
...:         print(f'{value} is even')
...:     else:
...:         print(f'{value} is odd')
...
Enter an integer: 10
10 is even
Enter an integer: 7
7 is odd

```

3.13 Built-In Function range: A Deeper Look

Function `range` also has two- and three-argument versions. As you've seen, `range`'s one-argument version produces a sequence of consecutive integers from 0 up to, but not including, the argument's value. Function `range`'s two-argument version produces a sequence of consecutive integers from its first argument's value up to, but not including, the second argument's value, as in:

```
In [1]: for number in range(5, 10):
...:     print(number, end=' ')
...:
5 6 7 8 9
```

Function `range`'s three-argument version produces a sequence of integers from its first argument's value up to, but not including, the second argument's value, *incrementing* by the third argument's value, which is known as the **step**:

```
In [2]: for number in range(0, 10, 2):
...:     print(number, end=' ')
...:
0 2 4 6 8
```

If the third argument is negative, the sequence progresses from the first argument's value *down* to, but not including the second argument's value, *decrementing* by the third argument's value, as in:

```
In [3]: for number in range(10, 0, -2):
...:     print(number, end=' ')
...:
10 8 6 4 2
```



Self Check

1 (*True/False*) Function call `range(1, 10)` generates the sequence 1 through 10.

Answer: False. Function call `range(1, 10)` generates the sequence 1 through 9.

2 (*IPython Session*) What happens if you try to print the items in `range(10, 0, 2)`?

Answer: Nothing displays because the step is not negative (this is not a fatal error):

```
In [1]: for number in range(10, 0, 2):
...:     print(number, end=' ')
...:

In [2]:
```

3 (*IPython Session*) Use a `for` statement, `range` and `print` to display on one line the sequence of values 99 88 77 66 55 44 33 22 11 0, each separated by one space.

Answer:

```
In [3]: for number in range(99, -1, -11):
...:     print(number, end=' ')
...:
99 88 77 66 55 44 33 22 11 0
```

4 (*IPython Session*) Use `for` and `range` to sum the even integers from 2 through 100, then display the sum.

Answer:

```
In [4]: total = 0

In [5]: for number in range(2, 101, 2):
...:     total += number
...:

In [6]: total
Out[6]: 2550
```

3.14 Using Type Decimal for Monetary Amounts

In this section, we introduce `Decimal` capabilities for precise monetary calculations. If you enter banking or other fields that require the accuracy provided by type `Decimal`, you should investigate `Decimal`'s capabilities in depth.

For most scientific and other mathematical applications that use numbers with decimal points, Python's built-in floating-point numbers work well. For example, when we speak of a "normal" body temperature of 98.6, we do not need to be precise to a large number of digits. When we view the temperature on a thermometer and read it as 98.6, the actual value may be 98.5999473210643. The point here is that calling this number 98.6 is adequate for most body-temperature applications.

Floating-point values are stored in binary format (we introduced binary in the first chapter and discuss it in depth in the online "Number Systems" appendix). Some floating-point values are represented only approximately when they're converted to binary. For example, consider the variable `amount` with the dollars-and-cents value 112.31. If you display `amount`, it appears to have the exact value you assigned to it:

```
In [1]: amount = 112.31

In [2]: print(amount)
112.31
```

However, if you print `amount` with 20 digits of precision to the right of the decimal point, you can see that the actual floating-point value in memory is not exactly 112.31—it's only an approximation:

```
In [3]: print(f'{amount:.20f}')
112.31000000000000227374
```

Many applications require *precise* representation of numbers with decimal points. Institutions like banks that deal with millions or even billions of transactions per day have to tie out their transactions "to the penny." Floating-point numbers can represent some but not all monetary amounts with to-the-penny precision.

The [Python Standard Library](#)² provides many predefined capabilities you can use in your Python code to avoid "reinventing the wheel." For monetary calculations and other applications that require precise representation and manipulation of numbers with decimal points, the Python Standard Library provides type `Decimal`, which uses a special coding scheme to solve the problem of to-the-penny precision. That scheme requires additional memory to hold the numbers and additional processing time to perform calcu-

2. <https://docs.python.org/3.7/library/index.html>.

lations but provides the to-the-penny precision required for monetary calculations. Banks also have to deal with other issues such as using a fair rounding algorithm when they're calculating daily interest on accounts. Type `Decimal` offers such capabilities.³

Importing Type `Decimal` from the `decimal` Module

We've used several built-in types—`int` (for integers, like 10), `float` (for floating-point numbers, like 7.5) and `str` (for strings like 'Python'). The `Decimal` type is not built into Python. Rather, it's part of the Python Standard Library, which is divided into **modules**—groups of related capabilities. The `decimal` module defines type `Decimal` and its capabilities.

To use capabilities from a module, you must first `import` the entire module, as in

```
import decimal
```

and refer to the `Decimal` type as `decimal.Decimal`, or you must indicate a specific capability to import using `from...import`, as we do here:

```
In [4]: from decimal import Decimal
```

This imports only the type `Decimal` from the `decimal` module so that you can use it in your code. We'll discuss other `import` forms beginning in the next chapter.

Creating Decimals

You typically create a `Decimal` from a string:

```
In [5]: principal = Decimal('1000.00')
```

```
In [6]: principal
```

```
Out[6]: Decimal('1000.00')
```

```
In [7]: rate = Decimal('0.05')
```

```
In [8]: rate
```

```
Out[8]: Decimal('0.05')
```

We'll soon use the variables `principal` and `rate` in a compound-interest calculation.

Decimal Arithmetic

Decimals support the standard arithmetic operators `+`, `-`, `*`, `/`, `//`, `**` and `%`, as well as the corresponding augmented assignments:

```
In [9]: x = Decimal('10.5')
```

```
In [10]: y = Decimal('2')
```

```
In [11]: x + y
```

```
Out[11]: Decimal('12.5')
```

```
In [12]: x // y
```

```
Out[12]: Decimal('5')
```

```
In [13]: x += y
```

```
In [14]: x
```

```
Out[14]: Decimal('12.5')
```

3. For more `decimal` module features, visit <https://docs.python.org/3.7/library/decimal.html>.

You may perform arithmetic between Decimals and integers, but not between Decimals and floating-point numbers.

Compound-Interest Problem Requirements Statement

Let's compute compound interest using the `Decimal` type for *precise* monetary calculations. Consider the following requirements statement:

A person invests \$1000 in a savings account yielding 5% interest. Assuming that the person leaves all interest on deposit in the account, calculate and display the amount of money in the account at the end of each year for 10 years. Use the following formula for determining these amounts:

$$a = p(1 + r)^n$$

where

p is the original amount invested (i.e., the principal),

r is the annual interest rate,

n is the number of years and

a is the amount on deposit at the end of the n th year.

Calculating Compound Interest

To solve this problem, let's use variables `principal` and `rate` that we defined in snippets [5] and [7], and a `for` statement that performs the interest calculation for each of the 10 years the money remains on deposit. For each year, the loop displays a formatted string containing the year number and the amount on deposit at the end of that year:

```
In [15]: for year in range(1, 11):
    ...:     amount = principal * (1 + rate) ** year
    ...:     print(f'{year:>2}{amount:>10.2f}')
    ...:
1 1050.00
2 1102.50
3 1157.62
4 1215.51
5 1276.28
6 1340.10
7 1407.10
8 1477.46
9 1551.33
10 1628.89
```

The algebraic expression $(1 + r)^n$ from the requirements statement is written as

$$(1 + \text{rate}) ^\star \text{year}$$

where variable `rate` represents r and variable `year` represents n .

Formatting the Year and Amount on Deposit

The statement

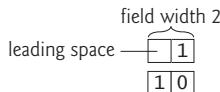
```
print(f'{year:>2}{amount:>10.2f}')
```

uses an f-string with two placeholders to format the loop's output.

The placeholder

```
{year:>2}
```

uses the format specifier `>2` to indicate that year's value should be **right aligned (>)** in a field of width 2—the **field width** specifies the number of character positions to use when displaying the value. For the single-digit year values 1–9, the format specifier `>2` displays a space character followed by the value, thus right aligning the years in the first column. The following diagram shows the numbers 1 and 10 each formatted in a field width of 2:

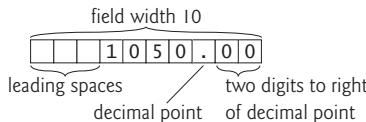


You can **left align** values with `<`.

The format specifier `10.2f` in the placeholder

```
{amount:>10.2f}
```

formats amount as a floating-point number (`f`) right aligned (`>`) in a field width of 10 with a decimal point and two digits to the right of the decimal point (`.2`). Formatting the amounts this way *aligns their decimal points vertically*, as is typical with monetary amounts. In the 10 character positions, the three rightmost characters are the number's decimal point followed by the two digits to its right. The remaining seven character positions are the leading spaces and the digits to the decimal point's left. In this example, all the dollar amounts have four digits to the left of the decimal point, so each number is formatted with three leading spaces. The following diagram shows the formatting for the value 1050.00:



Self Check

- 1 (Fill-In) A field width specifies the _____ to use when displaying a value.

Answer: number of character positions.

- 2 (IPython Session) Assume that the tax on a restaurant bill is 6.25% and that the bill amount is \$37.45. Use type `Decimal` to calculate the bill total, then print the result with two digits to the right of the decimal point.

Answer:

```
In [1]: from decimal import Decimal
```

```
In [2]: print(f"{Decimal('37.45') * Decimal('1.0625'):.2f}")
39.79
```

3.15 break and continue Statements

The `break` and `continue` statements alter a loop's flow of control. Executing a `break` statement in a `while` or `for` immediately exits that statement. In the following code, `range` produces the integer sequence 0–99, but the loop terminates when `number` is 10:

```
In [1]: for number in range(100):
    ...
        if number == 10:
            break
        print(number, end=' ')
    ...
0 1 2 3 4 5 6 7 8 9
```

In a script, execution would continue with the next statement after the `for` loop. The `while` and `for` statements each have an optional `else` clause that executes only if the loop terminates normally—that is, not as a result of a `break`. We explore this in the exercises.

Executing a `continue` statement in a `while` or `for` loop skips the remainder of the loop's suite. In a `while`, the condition is then tested to determine whether the loop should continue executing. In a `for`, the loop processes the next item in the sequence (if any):

```
In [2]: for number in range(10):
    ...
        if number == 5:
            continue
        print(number, end=' ')
    ...
0 1 2 3 4 6 7 8 9
```

3.16 Boolean Operators and, or and not

The conditional operators `>`, `<`, `>=`, `<=`, `==` and `!=` can be used to form simple conditions such as `grade >= 60`. To form more complex conditions that combine simple conditions, use the `and`, `or` and `not` Boolean operators.

Boolean Operator and

To ensure that two conditions are *both* `True` before executing a control statement's suite, use the **Boolean and operator** to combine the conditions. The following code defines two variables, then tests a condition that's `True` if and only if *both* simple conditions are `True`—if either (or both) of the simple conditions is `False`, the entire `and` expression is `False`:

```
In [1]: gender = 'Female'

In [2]: age = 70

In [3]: if gender == 'Female' and age >= 65:
    ...
        print('Senior female')
    ...
Senior female
```

The `if` statement has two simple conditions:

- `gender == 'Female'` determines whether a person is a female and
- `age >= 65` determines whether that person is a senior citizen.

The simple condition to the left of the `and` operator evaluates first because `==` has higher precedence than `and`. If necessary, the simple condition to the right of `and` evaluates next, because `>=` has higher precedence than `and`. (We'll discuss shortly why the right side of an `and` operator evaluates *only* if the left side is `True`.) The entire `if` statement condition is `True` if and only if *both* of the simple conditions are `True`. The combined condition can be made clearer by adding redundant (unnecessary) parentheses

```
(gender == 'Female') and (age >= 65)
```

The table below summarizes the `and` operator by showing all four possible combinations of `False` and `True` values for `expression1` and `expression2`—such tables are called *truth tables*:

<code>expression1</code>	<code>expression2</code>	<code>expression1 and expression2</code>
<code>False</code>	<code>False</code>	<code>False</code>
<code>False</code>	<code>True</code>	<code>False</code>
<code>True</code>	<code>False</code>	<code>False</code>
<code>True</code>	<code>True</code>	<code>True</code>

Boolean Operator `or`

Use the **Boolean `or` operator** to test whether one *or* both of two conditions are `True`. The following code tests a condition that's `True` if either *or* both simple conditions are `True`—the entire condition is `False` only if *both* simple conditions are `False`:

```
In [4]: semester_average = 83
In [5]: final_exam = 95
In [6]: if semester_average >= 90 or final_exam >= 90:
...:     print('Student gets an A')
...:
Student gets an A
```

Snippet [6] also contains two simple conditions:

- `semester_average >= 90` determines whether a student's average was an `A` (90 or above) during the semester, and
- `final_exam >= 90` determines whether a student's final-exam grade was an `A`.

The truth table below summarizes the Boolean `or` operator. Operator `and` has higher precedence than `or`.

<code>expression1</code>	<code>expression2</code>	<code>expression1 or expression2</code>
<code>False</code>	<code>False</code>	<code>False</code>
<code>False</code>	<code>True</code>	<code>True</code>
<code>True</code>	<code>False</code>	<code>True</code>
<code>True</code>	<code>True</code>	<code>True</code>

Improving Performance with Short-Circuit Evaluation

Python stops evaluating an `and` expression as soon as it knows whether the entire condition is `False`. Similarly, Python stops evaluating an `or` expression as soon as it knows whether the entire condition is `True`. This is called *short-circuit evaluation*. So the condition

```
gender == 'Female' and age >= 65
```

stops evaluating immediately if `gender` is not equal to `'Female'` because the entire expression must be `False`. If `gender` is equal to `'Female'`, execution continues, because the entire expression will be `True` if the age is greater than or equal to 65.

Similarly, the condition

```
semester_average >= 90 or final_exam >= 90
```

stops evaluating immediately if `semester_average` is greater than or equal to 90 because the entire expression must be True. If `semester_average` is less than 90, execution continues, because the expression could still be True if the `final_exam` is greater than or equal to 90.

In operator expressions that use `and`, make the condition that's more likely to be `False` the leftmost condition. In `or` operator expressions, make the condition that's more likely to be True the leftmost condition. These can reduce a program's execution time.

Boolean Operator `not`

The **Boolean `not` operator** “reverses” the meaning of a condition—`True` becomes `False` and `False` becomes `True`. This is a **unary operator**—it has only *one* operand. You place the `not` operator before a condition to choose a path of execution if the original condition (without the `not` operator) is `False`, such as in the following code:

```
In [7]: grade = 87
```

```
In [8]: if not grade == -1:  
....:     print('The next grade is', grade)  
....:  
The next grade is 87
```

Often, you can avoid using `not` by expressing the condition in a more “natural” or convenient manner. For example, the preceding `if` statement can also be written as follows:

```
In [9]: if grade != -1:  
....:     print('The next grade is', grade)  
....:  
The next grade is 87
```

The truth table below summarizes the `not` operator.

expression	<code>not</code> expression
<code>False</code>	<code>True</code>
<code>True</code>	<code>False</code>

The following table shows the precedence and grouping of the operators introduced so far, from top to bottom, in decreasing order of precedence.

Operators	Grouping
<code>()</code>	left to right
<code>**</code>	right to left
<code>*</code> <code>/</code> <code>//</code> <code>%</code>	left to right
<code>+</code> <code>-</code>	left to right
<code><</code> <code><=</code> <code>></code> <code>>=</code> <code>==</code> <code>!=</code>	left to right
<code>not</code>	left to right
<code>and</code>	left to right
<code>or</code>	left to right



Self Check

I (*IPython Session*) Assume that $i = 1$, $j = 2$, $k = 3$ and $m = 2$. What does each of the following conditions display?

- a) $(i \geq 1) \text{ and } (j < 4)$
- b) $(m \leq 99) \text{ and } (k < m)$
- c) $(j \geq i) \text{ or } (k == m)$
- d) $(k + m < j) \text{ or } (3 - j \geq k)$
- e) $\text{not } (k > m)$

Answer:

```
In [1]: i = 1
In [2]: j = 2
In [3]: k = 3
In [4]: m = 2
In [5]: (i >= 1) and (j < 4)
Out[5]: True
In [6]: (m <= 99) and (k < m)
Out[6]: False
In [7]: (j >= i) or (k == m)
Out[7]: True
In [8]: (k + m < j) or (3 - j >= k)
Out[8]: False
In [9]: not (k > m)
Out[9]: False
```

3.17 Intro to Data Science: Measures of Central Tendency—Mean, Median and Mode

Here we continue our discussion of using statistics to analyze data with several additional descriptive statistics, including:

- **mean**—the average value in a set of values.
- **median**—the middle value when all the values are arranged in sorted order.
- **mode**—the most frequently occurring value.

These are **measures of central tendency**—each is a way of producing a single value that represents a “central” value in a set of values, i.e., a value which is in some sense typical of the others.

Let’s calculate the mean, median and mode on a list of integers. The following session creates a list called `grades`, then uses the built-in `sum` and `len` functions to calculate the mean “by hand”—`sum` calculates the total of the grades (397) and `len` returns the number of grades (5):

```
In [1]: grades = [85, 93, 45, 89, 85]
```

```
In [2]: sum(grades) / len(grades)
Out[2]: 79.4
```

The previous chapter mentioned the descriptive statistics `count` and `sum`—implemented in Python as the built-in functions `len` and `sum`. Like functions `min` and `max` (introduced in the preceding chapter), `sum` and `len` are both examples of functional-style programming *reductions*—they reduce a collection of values to a single value—the sum of those values and the number of values, respectively. In Fig. 3.1’s class-average example, we could have deleted lines 10–15 and replaced average in line 16 with snippet [2]’s calculation.

The Python Standard Library’s **statistics module** provides functions for calculating the mean, median and mode—these, too, are reductions. To use these capabilities, first import the `statistics` module:

```
In [3]: import statistics
```

Then, you can access the module’s functions with “`statistics.`” followed by the name of the function to call. The following calculates the `grades` list’s mean, median and mode, using the `statistics` module’s `mean`, `median` and `mode` functions:

```
In [4]: statistics.mean(grades)
Out[4]: 79.4
```

```
In [5]: statistics.median(grades)
Out[5]: 85
```

```
In [6]: statistics.mode(grades)
Out[6]: 85
```

Each function’s argument must be an *iterable*—in this case, the list `grades`. To confirm that the median and mode are correct, you can use the built-in **sorted function** to get a copy of `grades` with its values arranged in increasing order:

```
In [7]: sorted(grades)
Out[7]: [45, 85, 85, 89, 93]
```

The `grades` list has an odd number of values (5), so `median` returns the middle value (85). If the list’s number of values is even, `median` returns the *average* of the *two* middle values. Studying the sorted values, you can see that 85 is the mode because it occurs most frequently (twice). The `mode` function causes a `StatisticsError` for lists like

```
[85, 93, 45, 89, 85, 93]
```

in which there are two or more “most frequent” values. Such a set of values is said to be **bimodal**. Here, both 85 and 93 occur twice. We’ll say more about mean, median and mode in the Intro to Data Science exercises at the end of the chapter.



Self Check

1 (Fill-In) The _____ statistic indicates the average value in a set of values.

Answer: mean.

2 (Fill-In) The _____ statistic indicates the most frequently occurring value in a set of values.

Answer: mode.

3 (Fill-In) The _____ statistic indicates the middle value in a set of values.

Answer: median.

4 (IPython Session) For the values 47, 95, 88, 73, 88 and 84, use the `statistics` module to calculate the mean, median and mode.

Answer:

```
In [1]: import statistics
In [2]: values = [47, 95, 88, 73, 88, 84]
In [3]: statistics.mean(values)
Out[3]: 79.16666666666667
In [4]: statistics.median(values)
Out[4]: 86.0
In [5]: statistics.mode(values)
Out[5]: 88
```

3.18 Wrap-Up

In this chapter, we discussed Python’s control statements, including `if`, `if...else`, `if...elif...else`, `while`, `for`, `break` and `continue`. We used pseudocode and top-down, stepwise refinement to develop several algorithms. You saw that many simple algorithms often have three execution phases—initialization, processing and termination.

You saw that the `for` statement performs sequence-controlled iteration—it processes each item in an iterable, such as a range of integers, a string or a list. You used the built-in function `range` to generate sequences of integers from 0 up to, but not including, its argument, and to determine how many times a `for` statement iterates. You used sentinel-controlled repetition with the `while` statement to create a loop that continues executing until a sentinel value is encountered. You used built-in function `range`’s two-argument version to generate sequences of integers from the first argument’s value up to, but not including, the second argument’s value. You also used the three-argument version in which the third argument indicated the step between integers in a range.

We introduced the `Decimal` type for precise monetary calculations and used it to calculate compound interest. You used f-strings and various format specifiers to create formatted output. We introduced the `break` and `continue` statements for altering the flow of control in loops. We discussed the Boolean operators `and`, `or` and `not` for creating conditions that combine simple conditions.

Finally, we continued our discussion of descriptive statistics by introducing measures of central tendency—mean, median and mode—and calculating them with functions from the Python Standard Library’s `statistics` module.

In the next chapter, you’ll create custom functions and use existing functions from Python’s `math` and `random` modules. We show several predefined functional-programming reductions. You’ll learn more of Python’s functional-programming capabilities.

Exercises

Unless specified otherwise, use IPython sessions for each exercise.

3.1 (Validating User Input) Modify the script of Fig. 3.3 to validate its inputs. For any input, if the value entered is other than 1 or 2, keep looping until the user enters a correct

value. Use one counter to keep track of the number of passes, then calculate the number of failures after all the user's inputs have been received.

- 3.2** (*What's Wrong with This Code?*) What is wrong with the following code?

```
a = b = 7
print('a = ', a, '\nb = ', b)
```

First, answer the question, then check your work in an IPython session.

- 3.3** (*What Does This Code Do?*) What does the following program print?

```
for row in range(10):
    for column in range(10):
        print('<' if row % 2 == 1 else '>', end='')
    print()
```

- 3.4** (*Fill in the Missing Code*) In the code below

```
for ***:
    for ***:
        print('@')
    print()
```

replace the `***` so that when you execute the code, it displays two rows, each containing seven @ symbols, as in:

```
@@@@@@@
@@@@@@@
```

- 3.5** (*if...else Statements*) Reimplement the script of Fig. 2.1 using three `if...else` statements rather than six `if` statements. [Hint: For example, think of `==` and `!=` as “opposite” tests.]

- 3.6** (*Turing Test*) The great British mathematician Alan Turing proposed a simple test to determine whether machines could exhibit intelligent behavior. A user sits at a computer and does the same text chat with a human sitting at a computer and a computer operating by itself. The user doesn't know if the responses are coming back from the human or the independent computer. If the user can't distinguish which responses are coming from the human and which are coming from the computer, then it's reasonable to say that the computer is exhibiting intelligence.

Create a script that plays the part of the independent computer, giving its user a simple medical diagnosis. The script should prompt the user with 'What is your problem?' When the user answers and presses *Enter*, the script should simply ignore the user's input, then prompt the user again with 'Have you had this problem before (yes or no)?' If the user enters 'yes', print 'Well, you have it again.' If the user answers 'no', print 'Well, you have it now.'

Would this conversation convince the user that the entity at the other end exhibited intelligent behavior? Why or why not?

- 3.7** (*Table of Squares and Cubes*) In Exercise 2.8, you wrote a script to calculate the squares and cubes of the numbers from 0 through 5, then printed the resulting values in table format. Reimplement your script using a `for` loop and the f-string capabilities you learned in this chapter to produce the following table with the numbers right aligned in each column.

number	square	cube
0	0	0
1	1	1
2	4	8
3	9	27
4	16	64
5	25	125

3.8 (Arithmetic, Smallest and Largest) In Exercise 2.10, you wrote a script that input three integers, then displayed the sum, average, product, smallest and largest of those values. Reimplement your script with a loop that inputs four integers.

3.9 (Separating the Digits in an Integer) In Exercise 2.11, you wrote a script that separated a five-digit integer into its individual digits and displayed them. Reimplement your script to use a loop that in each iteration “picks off” one digit (left to right) using the `//` and `%` operators, then displays that digit.

3.10 (7% Investment Return) Reimplement Exercise 2.12 to use a loop that calculates and displays the amount of money you’ll have each year at the ends of years 1 through 30.

3.11 (Miles Per Gallon) Drivers are concerned with the mileage obtained by their automobiles. One driver has kept track of several tankfuls of gasoline by recording miles driven and gallons used for each tankful. Develop a sentinel-controlled-repetition script that prompts the user to input the miles driven and gallons used for each tankful. The script should calculate and display the miles per gallon obtained for each tankful. After processing all input information, the script should calculate and display the combined miles per gallon obtained for all tankfuls (that is, total miles driven divided by total gallons used).

```
Enter the gallons used (-1 to end): 12.8
Enter the miles driven: 287
The miles/gallon for this tank was 22.421875
Enter the gallons used (-1 to end): 10.3
Enter the miles driven: 200
The miles/gallon for this tank was 19.417475
Enter the gallons used (-1 to end): 5
Enter the miles driven: 120
The miles/gallon for this tank was 24.000000
Enter the gallons used (-1 to end): -1
The overall average miles/gallon was 21.601423
```

3.12 (Palindromes) A palindrome is a number, word or text phrase that reads the same backwards or forwards. For example, each of the following five-digit integers is a palindrome: 12321, 55555, 45554 and 11611. Write a script that reads in a five-digit integer and determines whether it’s a palindrome. [Hint: Use the `//` and `%` operators to separate the number into its digits.]

3.13 (Factorials) Factorial calculations are common in probability. The factorial of a nonnegative integer n is written $n!$ (pronounced “ n factorial”) and is defined as follows:

$$n! = n \cdot (n - 1) \cdot (n - 2) \cdot \dots \cdot 1$$

for values of n greater than or equal to 1, with $0!$ defined to be 1. So,

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$$

which is 120. Factorials increase in size very rapidly. Write a script that inputs a nonnegative integer and computes and displays its factorial. Try your script on the integers 10, 20,

30 and even larger values. Did you find any integer input for which Python could not produce an integer factorial value?

3.14 (Challenge: Approximating the Mathematical Constant π) Write a script that computes the value of π from the following infinite series. Print a table that shows the value of π approximated by one term of this series, by two terms, by three terms, and so on. How many terms of this series do you have to use before you first get 3.14? 3.141? 3.1415? 3.14159?

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \dots$$

3.15 (Challenge: Approximating the Mathematical Constant e) Write a script that estimates the value of the mathematical constant e by using the formula below. Your script can stop after summing 10 terms.

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \dots$$

3.16 (Nested Control Statements) Use a loop to find the *two* largest values of 10 numbers entered.

3.17 (Nested Loops) Write a script that displays the following triangle patterns separately, one below the other. Separate each pattern from the next by one blank line. Use `for` loops to generate the patterns. Display all asterisks (*) with a single statement of the form

```
print('* ', end='')
```

which causes the asterisks to display side by side. [Hint: For the last two patterns, begin each line with zero or more space characters.]

(a)	(b)	(c)	(d)
*	*****	*****	*
**	*****	*****	**
***	*****	*****	***
****	*****	*****	****
*****	*****	*****	*****
*****	*****	*****	*****
*****	***	***	*****
*****	**	**	*****
*****	*	*	*****

3.18 (Challenge: Nested Looping) Modify your script from Exercise 3.17 to display all four patterns side-by-side (as shown above) by making clever use of nested `for` loops. Separate each triangle from the next by three horizontal spaces. [Hint: One `for` loop should control the row number. Its nested `for` loops should calculate from the row number the appropriate number of asterisks and spaces for each of the four patterns.]

3.19 (Brute-Force Computing: Pythagorean Triples) A right triangle can have sides that are all integers. The set of three integer values for the sides of a right triangle is called a Pythagorean triple. These three sides must satisfy the relationship that the sum of the squares of two of the sides is equal to the square of the hypotenuse. Find all Pythagorean triples for `side1`, `side2` and `hypotenuse` (such as 3, 4 and 5) all no larger than 20. Use a triple-nested `for`-loop that tries all possibilities. This is an example of “brute-force” computing. You’ll

learn in more advanced computer science courses that there are many interesting problems for which there is no known algorithmic approach other than sheer brute force.

3.20 (Binary-to-Decimal Conversion) Input an integer containing 0s and 1s (i.e., a “binary” integer) and display its decimal equivalent. The online appendix, “Number Systems,” discusses the binary number system. [Hint: Use the modulus and division operators to pick off the “binary” number’s digits one at a time from right to left. Just as in the decimal number system, where the rightmost digit has the positional value 1 and the next digit to the left has the positional value 10, then 100, then 1000, etc., in the binary number system, the rightmost digit has the positional value 1, the next digit to the left has the positional value 2, then 4, then 8, etc. Thus, the decimal number 234 can be interpreted as $2 * 100 + 3 * 10 + 4 * 1$. The decimal equivalent of binary 1101 is $1 * 8 + 1 * 4 + 0 * 2 + 1 * 1$.]

3.21 (Calculate Change Using Fewest Number of Coins) Write a script that inputs a purchase price of a dollar or less for an item. Assume the purchaser pays with a dollar bill. Determine the amount of change the cashier should give back to the purchaser. Display the change using the *fewest* number of pennies, nickels, dimes and quarters. For example, if the purchaser is due 73 cents in change, the script would output:

```
Your change is:  
2 quarters  
2 dimes  
3 pennies
```

3.22 (Optional else Clause of a Loop) The `while` and `for` statements each have an optional `else` clause. In a `while` statement, the `else` clause executes when the condition becomes `False`. In a `for` statement, the `else` clause executes when there are no more items to process. If you break out of a `while` or `for` that has an `else`, the `else` part does *not* execute. Execute the following code to see that the `else` clause executes only if the `break` statement does not:

```
for i in range(2):  
    value = int(input('Enter an integer (-1 to break): '))  
    print('You entered:', value)  
  
    if value == -1:  
        break  
    else:  
        print('The loop terminated without executing the break')
```

For more information on loop `else` clauses, see

<https://docs.python.org/3/tutorial/controlflow.html#break-and-continue-statements-and-else-clauses-on-loops>

3.23 (Validating Indentation) The file `validate_indent.py` in this chapter’s `ch03` examples folder contains the following code with incorrect indentation:

```
grade = 93  
  
if grade >= 90:  
    print('A')  
    print('Great Job!')  
    print('Take a break from studying')
```

The Python Standard Library includes a code indentation validator module named **tabnanny**, which you can run as a script to check your code for proper indentation—this is one of many static code analysis tools. Execute the following command in the ch03 folder to see the results of analyzing `validate_indent.py`:

```
python -m tabnanny validate_indent.py
```

Suppose you accidentally aligned the second `print` statement under the `i` in the `if` keyword. What kind of error would that be? Would you expect `tabnanny` to flag that as an error?

3.24 (Project: Using the prospector Static Code Analysis Tool) The `prospector` tool runs several popular static code analysis tools to check your Python code for common errors and to help you improve your code. Check that you've installed `prospector` (see the Before You Begin section that follows the Preface). Run `prospector` on each of the scripts in this chapter. To do so, open the folder containing the scripts in a Terminal (macOS/Linux), Command Prompt (Windows) or shell (Linux), then run the following command from that folder:

```
prospector --strictness veryhigh --doc-warnings
```

Study the output to see the kinds of issues `prospector` locates in Python code. In general, run `prospector` on all new code you create.

3.25 (Project: Using prospector to Analyze Open-Source Code on GitHub) Locate a Python open-source project on GitHub, download its source code and extract it into a folder on your system. Open that folder in a Terminal (macOS/Linux), Command Prompt (Windows) or shell (Linux), then run the following command from that folder:

```
prospector --strictness veryhigh --doc-warnings
```

Study the output to see more of the kinds of issues `prospector` locates in Python code.

3.26 (Research: Anscombe's Quartet) In this book's data science case studies, we'll emphasize the importance of "getting to know your data." The *basic descriptive statistics* that you've seen in this chapter's and the previous chapter's Intro to Data Science sections certainly help you know more about your data. One caution, though, is that different datasets can have identical or nearly identical descriptive statistics and yet the data can be significantly different. For an example of this phenomenon, research *Anscombe's Quartet*. You should find four datasets and the associated visualizations. It's the visualizations that convince you the datasets are quite different. In an exercise in a later chapter, you'll create these visualizations.

3.27 (World Population Growth) World population has grown considerably over the centuries. Continued growth could eventually challenge the limits of breathable air, drinkable water, arable land and other limited resources. There's evidence that growth has been slowing in recent years and that world population could peak some time this century, then start to decline.

For this exercise, research world population growth issues. This is a controversial topic, so be sure to investigate various viewpoints. Get estimates for the current world population and its growth rate. Write a script that calculates world population growth each year for the next 100 years, *using the simplifying assumption that the current growth rate will stay constant*. Print the results in a table. The first column should display the year

from 1 to 100. The second column should display the anticipated world population at the end of that year. The third column should display the numerical increase in the world population that would occur that year. Using your results, determine the years in which the population would be double and eventually quadruple what it is today.

3.28 (*Intro to Data Science: Mean, Median and Mode*) Calculate the mean, median and mode of the values 9, 11, 22, 34, 17, 22, 34, 22 and 40. Suppose the values included another 34. What problem might occur?

3.29 (*Intro to Data Science: Problem with the Median*) For an odd number of values, to get the median you simply arrange them in order and take the middle value. For an even number, you average the two middle values. What problem occurs if those two values are different?

3.30 (*Intro to Data Science: Outliers*) In statistics, outliers are values out of the ordinary and possibly way out of the ordinary. Sometimes, outliers are simply bad data. In the data science case studies, we'll see that outliers can distort results. Which of the three measures of central tendency we discussed—mean, median and mode—is most affected by outliers? Why? Which of these measures are not affected or least affected? Why?

3.31 (*Intro to Data Science: Categorical Data*) Mean, median and mode work well with numerical values. You can use them in calculations and arrange them in meaningful order. Categorical values are descriptive names like Boxer, Poodle, Collie, Beagle, Bulldog and Chihuahua. Normally, you don't use these in calculations nor associate an order with them. Which if any of the descriptive statistics are appropriate for categorical data?

4

Functions



Objectives

In this chapter, you'll

- Create custom functions.
- Import and use Python Standard Library modules, such as `random` and `math`, to reuse code and avoid “reinventing the wheel.”
- Pass data between functions.
- Generate a range of random numbers.
- Learn simulation techniques using random-number generation.
- Pack values into a tuple and unpack values from a tuple.
- Return multiple values from a function via a tuple.
- Understand how an identifier's scope determines where in your program you can use it.
- Create functions with default parameter values.
- Call functions with keyword arguments.
- Create functions that can receive any number of arguments.
- Use methods of an object.

Outline

-
- | | |
|---|---|
| 4.1 Introduction
4.2 Defining Functions
4.3 Functions with Multiple Parameters
4.4 Random-Number Generation
4.5 Case Study: A Game of Chance
4.6 Python Standard Library
4.7 <code>math</code> Module Functions
4.8 Using IPython Tab Completion for Discovery
4.9 Default Parameter Values
4.10 Keyword Arguments
4.11 Arbitrary Argument Lists | 4.12 Methods: Functions That Belong to Objects
4.13 Scope Rules
4.14 <code>import</code> : A Deeper Look
4.15 Passing Arguments to Functions: A Deeper Look
4.16 Function-Call Stack
4.17 Functional-Style Programming
4.18 Intro to Data Science: Measures of Dispersion
4.19 Wrap-Up Exercises |
|---|---|
-

4.1 Introduction

Experience has shown that the best way to develop and maintain a large program is to construct it from smaller, more manageable pieces. This technique is called **divide and conquer**. Using existing functions as building blocks for creating new programs is a key aspect of **software reusability**—it’s also a major benefit of object-oriented programming. Packaging code as a function allows you to execute it from various locations in your program just by calling the function, rather than duplicating the possibly lengthy code. This also makes programs easier to modify. When you change a function’s code, all calls to the function execute the updated version.

4.2 Defining Functions

You’ve called many built-in functions (`int`, `float`, `print`, `input`, `type`, `sum`, `len`, `min` and `max`) and a few functions from the statistics module (`mean`, `median` and `mode`). Each performed a single, well-defined task. You’ll often define and call *custom* functions. The following session defines a `square` function that calculates the square of its argument. Then it calls the function twice—once to square the `int` value 7 (producing the `int` value 49) and once to square the `float` value 2.5 (producing the `float` value 6.25):

```
In [1]: def square(number):
...:     """Calculate the square of number."""
...:     return number ** 2
...:

In [2]: square(7)
Out[2]: 49

In [3]: square(2.5)
Out[3]: 6.25
```

The statements defining the function in the first snippet are written only once, but may be called “to do their job” from many points throughout a program and as often as you like. Calling `square` with a non-numeric argument like ‘hello’ causes a `TypeError` because the exponentiation operator (`**`) works only with numeric values.

Defining a Custom Function

A **function definition** (like `square` in snippet [1]) begins with the **`def` keyword**, followed by the function name (`square`), a set of parentheses and a colon (:). Like variable identifiers, by convention function names should begin with a lowercase letter and in multiword names underscores should separate each word.

The required parentheses contain the function's **parameter list**—a comma-separated list of **parameters** representing the data that the function needs to perform its task. Function `square` has only one parameter named `number`—the value to be squared.

If the parentheses are empty, the function does not use parameters to perform its task. Exercise 4.7 asks you to write a parameterless `date_and_time` function that displays the current date and time by reading it from your computer's system clock.

The indented lines after the colon (:) are the function's **block**, which consists of an optional docstring followed by the statements that perform the function's task. We'll soon point out the difference between a function's block and a control statement's suite.

Specifying a Custom Function's Docstring

The *Style Guide for Python Code* says that the first line in a function's block should be a docstring that briefly explains the function's purpose:

```
"""Calculate the square of number."""
```

To provide more detail, you can use a multiline docstring—the style guide recommends starting with a brief explanation, followed by a blank line and the additional details.

Returning a Result to a Function's Caller

When a function finishes executing, it returns control to its caller—that is, the line of code that called the function. In `square`'s block, the **`return`** statement:

```
return number ** 2
```

first squares `number`, then terminates the function and gives the result back to the caller. In this example, the first caller is `square(7)` in snippet [2], so IPython displays the result in `Out[2]`. Think of the return value, 49, as simply replacing the call `square(7)`. So after the call, you'd have `In [2]: 49`, and that would indeed produce `Out[2]: 49`. The second caller `square(2.5)` is in snippet [3], so IPython displays the result 6.25 in `Out[3]`.

Function calls also can be embedded in expressions. The following code calls `square` first, then `print` displays the result:

```
In [4]: print('The square of 7 is', square(7))
The square of 7 is 49
```

Here, too, think of the return value, 49, as simply replacing the call `square(7)`, which would indeed produce the output shown above.

There are two other ways to return control from a function to its caller:

- Executing a `return` statement without an expression terminates the function and *implicitly* returns the value `None` to the caller. The Python documentation states that `None` represents the absence of a value. `None` evaluates to `False` in conditions.
- When there's no `return` statement in a function, it *implicitly* returns the value `None` after executing the last statement in the function's block.

What Happens When You Call a Function

The expression `square(7)` passes the argument `7` to `square`'s parameter `number`. Then `square` calculates `number ** 2` and returns the result. The parameter `number` exists only during the function call. It's created on each call to the function to receive the argument value, and it's destroyed when the function returns its result to the caller.

Though we did not define variables in `square`'s block, it is possible to do so. A function's parameters and variables defined in its block are all **local variables**—they can be used only inside the function and exist only while the function is executing. Trying to access a local variable outside its function's block causes a `NameError`, indicating that the variable is not defined. We'll soon see how a behind-the-scenes mechanism called the *function-call stack* supports the automatic creation and destruction of a function's local variables—and helps the function return to its caller.

Accessing a Function's Docstring via IPython's Help Mechanism

IPython can help you learn about the modules and functions you intend to use in your code, as well as IPython itself. For example, to view a function's docstring to learn how to use the function, type the function's name followed by a **question mark (?)**:

```
In [5]: square?
Signature: square(number)
Docstring: Calculate the square of number.
File:      ~/Documents/examples/ch04/<ipython-input-1-7268c8ff93a9>
Type:     function
```

For our `square` function, the information displayed includes:

- The function's name and parameter list—known as its **signature**.
- The function's docstring.
- The name of the file containing the function's definition. For a function in an interactive session, this line shows information for the snippet that defined the function—the `1` in "`<ipython-input-1-7268c8ff93a9>`" means snippet [1].
- The type of the item for which you accessed IPython's help mechanism—in this case, a function.

If the function's source code is accessible from IPython—such as a function defined in the current session or imported into the session from a `.py` file—you can use `??` to display the function's full source-code definition:

```
In [6]: square??
Signature: square(number)
Source:
def square(number):
    """Calculate the square of number."""
    return number ** 2
File:      ~/Documents/examples/ch04/<ipython-input-1-7268c8ff93a9>
Type:     function
```

If the source code is not accessible from IPython, `??` simply shows the docstring.

If the docstring fits in the window, IPython displays the next `In []` prompt. If a docstring is too long to fit, IPython indicates that there's more by displaying a colon (`:`) at the bottom of the window—press the `Space` key to display the next screen. You can navigate

backwards and forwards through the docstring with the up and down arrow keys, respectively. IPython displays (END) at the end of the docstring. Press *q* (for “quit”) at any : or the (END) prompt to return to the next In [] prompt. To get a sense of IPython’s features, type ? at any In [] prompt, press *Enter*, then read the help documentation overview.



Self Check

- 1** (*True/False*) The function body is referred to as its suite.

Answer: False. The function body is referred to as its block.

- 2** (*True/False*) A function’s local variables exist after the function returns to its caller.

Answer: False. A function’s local variables exist until the function returns to its caller.

- 3** (*IPython Session*) Define a function `square_root` that receives a number as a parameter and returns the square root of that number. Determine the square root of 6.25.

Answer:

```
In [1]: def square_root(number):
...:     return number ** 0.5 # or number ** (1 / 2)
...:

In [2]: square_root(6.25)
Out[2]: 2.5
```

4.3 Functions with Multiple Parameters

Let’s define a `maximum` function that determines and returns the largest of three values—the following session calls the function three times with integers, floating-point numbers and strings, respectively.

```
In [1]: def maximum(value1, value2, value3):
...:     """Return the maximum of three values."""
...:     max_value = value1
...:     if value2 > max_value:
...:         max_value = value2
...:     if value3 > max_value:
...:         max_value = value3
...:     return max_value
...:

In [2]: maximum(12, 27, 36)
Out[2]: 36

In [3]: maximum(12.3, 45.6, 9.7)
Out[3]: 45.6

In [4]: maximum('yellow', 'red', 'orange')
Out[4]: 'yellow'
```

We did not place blank lines above and below the `if` statements, because pressing `return` on a blank line in interactive mode completes the function’s definition.

You also may call `maximum` with mixed types, such as `ints` and `floats`:

```
In [5]: maximum(13.5, -3, 7)
Out[5]: 13.5
```

The call `maximum(13.5, 'hello', 7)` results in `TypeError` because strings and numbers cannot be compared to one another with the greater-than (`>`) operator.

Function maximum's Definition

Function `maximum` specifies three parameters in a comma-separated list. Snippet [2]'s arguments 12, 27 and 36 are assigned to the parameters `value1`, `value2` and `value3`, respectively.

To determine the largest value, we process one value at a time:

- Initially, we assume that `value1` contains the largest value, so we assign it to the local variable `max_value`. Of course, it's possible that `value2` or `value3` contains the actual largest value, so we still must compare each of these with `max_value`.
- The first `if` statement then tests `value2 > max_value`, and if this condition is `True` assigns `value2` to `max_value`.
- The second `if` statement then tests `value3 > max_value`, and if this condition is `True` assigns `value3` to `max_value`.

Now, `max_value` contains the largest value, so we return it. When control returns to the caller, the parameters `value1`, `value2` and `value3` and the variable `max_value` in the function's block—which are all *local variables*—no longer exist.

Python's Built-In `max` and `min` Functions

For many common tasks, the capabilities you need already exist in Python. For example, built-in `max` and `min` functions know how to determine the largest and smallest of their two or more arguments, respectively:

```
In [6]: max('yellow', 'red', 'orange', 'blue', 'green')
Out[6]: 'yellow'
```

```
In [7]: min(15, 9, 27, 14)
Out[7]: 9
```

Each of these functions also can receive an iterable argument, such as a list or a string. Using built-in functions or functions from the Python Standard Library's modules rather than writing your own can reduce development time and increase program reliability, portability and performance. For a list of Python's built-in functions and modules, see

<https://docs.python.org/3/library/index.html>



Self Check

1 (*Fill-In*) A function with multiple parameters specifies them in a(n) _____.
Answer: comma-separated list.

2 (*True/False*) When defining a function in IPython interactive mode, pressing *Enter* on a blank line causes IPython to display another continuation prompt so you can continue defining the function's block.

Answer: False. When defining a function in IPython interactive mode, pressing *Enter* on a blank line terminates the function definition.

3 (*IPython Session*) Call function `max` with the list `[14, 27, 5, 3]` as an argument, then call function `min` with the string `'orange'` as an argument.

Answer:

```
In [1]: max([14, 27, 5, 3])
Out[1]: 27
```

```
In [2]: min('orange')
Out[2]: 'a'
```

4.4 Random-Number Generation

We now take a brief diversion into a popular type of programming application—simulation and game playing. You can introduce the **element of chance** via the Python Standard Library’s **random module**.

Rolling a Six-Sided Die

Let’s produce 10 random integers in the range 1–6 to simulate rolling a six-sided die:

```
In [1]: import random

In [2]: for roll in range(10):
...:     print(random.randrange(1, 7), end=' ')
...:
4 2 5 5 4 6 4 6 1 5
```

First, we import `random` so we can use the module’s capabilities. The `randrange` function generates an integer from the first argument value up to, but *not* including, the second argument value. Let’s use the up arrow key to recall the `for` statement, then press *Enter* to re-execute it. Notice that *different* values are displayed:

```
In [3]: for roll in range(10):
...:     print(random.randrange(1, 7), end=' ')
...:
4 5 4 5 1 4 1 4 6 5
```

Sometimes, you may want to guarantee **reproducibility** of a random sequence—for debugging, for example. At the end of this section, we’ll show how to do this with the `random` module’s `seed` function.

Rolling a Six-Sided Die 6,000,000 Times

If `randrange` truly produces integers at random, every number in its range has an equal **probability** (or *chance* or *likelihood*) of being returned each time we call it. To show that the die faces 1–6 occur with equal likelihood, the following script simulates 6,000,000 die rolls. When you run the script, each die face should occur approximately 1,000,000 times, as in the sample output.

```
1 # fig04_01.py
2 """Roll a six-sided die 6,000,000 times."""
3 import random
4
5 # Face frequency counters
6 frequency1 = 0
```

Fig. 4.1 | Roll a six-sided die 6,000,000 times. (Part 1 of 2.)

```

7  frequency2 = 0
8  frequency3 = 0
9  frequency4 = 0
10 frequency5 = 0
11 frequency6 = 0
12
13 # 6,000,000 die rolls
14 for roll in range(6_000_000): # note underscore separators
15     face = random.randrange(1, 7)
16
17     # increment appropriate face counter
18     if face == 1:
19         frequency1 += 1
20     elif face == 2:
21         frequency2 += 1
22     elif face == 3:
23         frequency3 += 1
24     elif face == 4:
25         frequency4 += 1
26     elif face == 5:
27         frequency5 += 1
28     elif face == 6:
29         frequency6 += 1
30
31 print(f'Face{"Frequency":>13}')
32 print(f'{1:>4}{frequency1:>13}')
33 print(f'{2:>4}{frequency2:>13}')
34 print(f'{3:>4}{frequency3:>13}')
35 print(f'{4:>4}{frequency4:>13}')
36 print(f'{5:>4}{frequency5:>13}')
37 print(f'{6:>4}{frequency6:>13}')

```

Face	Frequency
1	998686
2	1001481
3	999900
4	1000453
5	999953
6	999527

Fig. 4.1 | Roll a six-sided die 6,000,000 times. (Part 2 of 2.)

The script uses *nested* control statements (an `if...elif` statement nested in the `for` statement) to determine the number of times each die face appears. The `for` statement iterates 6,000,000 times. We used Python’s underscore (`_`) digit separator to make the value `6000000` more readable. The expression `range(6,000,000)` would be incorrect. Commas separate arguments in function calls, so Python would treat `range(6,000,000)` as a call to `range` with the *three* arguments `6`, `0` and `0`.

For each die roll, the script adds `1` to the appropriate counter variable. Run the program, and observe the results. This program might take a few seconds to complete execution. As you’ll see, each execution produces *different* results.

Note that we did not provide an `else` clause in the `if...elif` statement. Exercise 4.1 asks you to comment on the possible consequences of this.

Seeding the Random-Number Generator for Reproducibility

Function `randrange` actually generates **pseudorandom numbers**, based on an internal calculation that begins with a numeric value known as a **seed**. Repeatedly calling `randrange` produces a sequence of numbers that *appear* to be random, because each time you start a new interactive session or execute a script that uses the `random` module's functions, Python internally uses a *different* seed value.¹ When you're debugging logic errors in programs that use randomly generated data, it can be helpful to use the *same* sequence of random numbers until you've eliminated the logic errors, before testing the program with other values. To do this, you can use the `random` module's `seed` function to **seed the random-number generator** yourself—this forces `randrange` to begin calculating its pseudorandom number sequence from the seed you specify. In the following session, snippets [5] and [8] produce the same results, because snippets [4] and [7] use the same seed (32):

```
In [4]: random.seed(32)

In [5]: for roll in range(10):
...:     print(random.randrange(1, 7), end=' ')
...
1 2 2 3 6 2 4 1 6 1
In [6]: for roll in range(10):
...:     print(random.randrange(1, 7), end=' ')
...
1 3 5 3 1 5 6 4 3 5
In [7]: random.seed(32)

In [8]: for roll in range(10):
...:     print(random.randrange(1, 7), end=' ')
...
1 2 2 3 6 2 4 1 6 1
```

Snippet [6] generates *different* values because it simply continues the pseudorandom number sequence that began in snippet [5].



Self Check

1 (*Fill-In*) The element of chance can be introduced into computer applications using module _____.

Answer: `random`.

2 (*Fill-In*) The `random` module's _____ function enables reproducibility of random sequences.

Answer: `seed`.

-
- According to the documentation, Python bases the seed value on the system clock or an operating-system-dependent randomness source. For applications requiring secure random numbers, such as cryptography, the documentation recommends using the `secrets` module, rather than the `random` module.

3 (IPython Session) Requirements statement: Use a `for` statement, `randrange` and a conditional expression (introduced in the preceding chapter) to simulate 20 coin flips, displaying H for heads and T for tails all on the same line, each separated by a space.

Answer:

```
In [1]: import random
In [2]: for i in range(20):
...:     print('H' if random.randrange(2) == 0 else 'T', end=' ')
...
T H T T H T T T H T H H T H T H H H H
```

In snippet [2]’s output, an equal number of Ts and Hs appeared—that will not always be the case with random-number generation.

4.5 Case Study: A Game of Chance

In this section, we simulate the popular dice game known as “craps.” Here is the requirements statement:

You roll two six-sided dice, each with faces containing one, two, three, four, five and six spots, respectively. When the dice come to rest, the sum of the spots on the two upward faces is calculated. If the sum is 7 or 11 on the first roll, you win. If the sum is 2, 3 or 12 on the first roll (called “craps”), you lose (i.e., the “house” wins). If the sum is 4, 5, 6, 8, 9 or 10 on the first roll, that sum becomes your “point.” To win, you must continue rolling the dice until you “make your point” (i.e., roll that same point value). You lose by rolling a 7 before making your point.

The following script simulates the game and shows several sample executions, illustrating winning on the first roll, losing on the first roll, winning on a subsequent roll and losing on a subsequent roll.

```

1 # fig04_02.py
2 """Simulating the dice game Craps."""
3 import random
4
5 def roll_dice():
6     """Roll two dice and return their face values as a tuple."""
7     die1 = random.randrange(1, 7)
8     die2 = random.randrange(1, 7)
9     return (die1, die2) # pack die face values into a tuple
10
11 def display_dice(dice):
12     """Display one roll of the two dice."""
13     die1, die2 = dice # unpack the tuple into variables die1 and die2
14     print(f'Player rolled {die1} + {die2} = {sum(dice)}')
15
16 die_values = roll_dice() # first roll
17 display_dice(die_values)
18
```

Fig. 4.2 | Game of craps. (Part 1 of 2.)

```
19 # determine game status and point, based on first roll
20 sum_of_dice = sum(die_values)
21
22 if sum_of_dice in (7, 11): # win
23     game_status = 'WON'
24 elif sum_of_dice in (2, 3, 12): # lose
25     game_status = 'LOST'
26 else: # remember point
27     game_status = 'CONTINUE'
28     my_point = sum_of_dice
29     print('Point is', my_point)
30
31 # continue rolling until player wins or loses
32 while game_status == 'CONTINUE':
33     die_values = roll_dice()
34     display_dice(die_values)
35     sum_of_dice = sum(die_values)
36
37     if sum_of_dice == my_point: # win by making point
38         game_status = 'WON'
39     elif sum_of_dice == 7: # lose by rolling 7
40         game_status = 'LOST'
41
42 # display "wins" or "loses" message
43 if game_status == 'WON':
44     print('Player wins')
45 else:
46     print('Player loses')
```

Player rolled 2 + 5 = 7
Player wins

Player rolled 1 + 2 = 3
Player loses

Player rolled 5 + 4 = 9
Point is 9
Player rolled 4 + 4 = 8
Player rolled 2 + 3 = 5
Player rolled 5 + 4 = 9
Player wins

Player rolled 1 + 5 = 6
Point is 6
Player rolled 1 + 6 = 7
Player loses

Fig. 4.2 | Game of craps. (Part 2 of 2.)

Function `roll_dice`—Returning Multiple Values Via a Tuple

Function `roll_dice` (lines 5–9) simulates rolling two dice on each roll. The function is defined once, then called from several places in the program (lines 16 and 33). The empty parameter list indicates that `roll_dice` does not require arguments to perform its task.

The built-in and custom functions you've called so far each return one value. Sometimes it's useful to return more than one value, as in `roll_dice`, which returns both die values (line 9) as a **tuple**—an **immutable** (that is, unmodifiable) sequences of values. To create a tuple, separate its values with commas, as in line 9:

```
(die1, die2)
```

This is known as **packing a tuple**. The parentheses are optional, but we recommend using them for clarity. We discuss tuples in depth in the next chapter.

Function `display_dice`

To use a tuple's values, you can assign them to a comma-separated list of variables, which **unpacks** the tuple. To display each roll of the dice, the function `display_dice` (defined in lines 11–14 and called in lines 17 and 34) unpacks the tuple argument it receives (line 13). The number of variables to the left of `=` must match the number of elements in the tuple; otherwise, a `ValueError` occurs. Line 14 prints a formatted string containing both die values and their sum. We calculate the sum of the dice by passing the tuple to the built-in `sum` function—like a list, a tuple is a sequence.

Note that functions `roll_dice` and `display_dice` each begin their blocks with a docstring that states what the function does. Also, both functions contain local variables `die1` and `die2`. These variables do not “collide,” because they belong to different functions' blocks. Each local variable is accessible only in the block that defined it.

First Roll

When the script begins executing, lines 16–17 roll the dice and display the results. Line 20 calculates the sum of the dice for use in lines 22–29. You can win or lose on the first roll or any subsequent roll. The variable `game_status` keeps track of the win/loss status.

The **in operator** in line 22

```
sum_of_dice in (7, 11)
```

tests whether the tuple `(7, 11)` contains `sum_of_dice`'s value. If this condition is `True`, you rolled a 7 or an 11. In this case, you won on the first roll, so the script sets `game_status` to '`WON`'. The operator's right operand can be any iterable. There's also a **not in** operator to determine whether a value is *not* in an iterable. The preceding concise condition is equivalent to

```
(sum_of_dice == 7) or (sum_of_dice == 11)
```

Similarly, the condition in line 24

```
sum_of_dice in (2, 3, 12)
```

tests whether the tuple `(2, 3, 12)` contains `sum_of_dice`'s value. If so, you lost on the first roll, so the script sets `game_status` to '`LOST`'.

For any other sum of the dice (4, 5, 6, 8, 9 or 10):

- line 27 sets `game_status` to '`CONTINUE`' so you can continue rolling

- line 28 stores the sum of the dice in `my_point` to keep track of what you must roll to win and
- line 29 displays `my_point`.

Subsequent Rolls

If `game_status` is equal to 'CONTINUE' (line 32), you did not win or lose, so the `while` statement's suite (lines 33–40) executes. Each loop iteration calls `roll_dice`, displays the die values and calculates their sum. If `sum_of_dice` is equal to `my_point` (line 37) or 7 (line 39), the script sets `game_status` to 'WON' or 'LOST', respectively, and the loop terminates. Otherwise, the `while` loop continues executing with the next roll.

Displaying the Final Results

When the loop terminates, the script proceeds to the `if...else` statement (lines 43–46), which prints 'Player wins' if `game_status` is 'WON', or 'Player loses' otherwise.



Self Check

1 (*Fill-In*) The _____ operator tests whether its right operand's iterable contains its left operand's value.

Answer: `in`.

2 (*IPython Session*) Pack a student tuple with the name 'Sue' and the list [89, 94, 85], display the tuple, then unpack it into variables `name` and `grades`, and display their values.

Answer:

```
In [1]: student = ('Sue', [89, 94, 85])

In [2]: student
Out[2]: ('Sue', [89, 94, 85])

In [3]: name, grades = student

In [4]: print(f'{name}: {grades}')
Sue: [89, 94, 85]
```

4.6 Python Standard Library

Typically, you write Python programs by combining functions and classes (that is, custom types) that you create with preexisting functions and classes defined in modules, such as those in the Python Standard Library and other libraries. A key programming goal is to avoid “reinventing the wheel.”

A module is a file that groups related functions, data and classes. The type `Decimal` from the Python Standard Library’s `decimal` module is actually a class. We introduced classes briefly in Chapter 1 and discuss them in detail in the “Object-Oriented Programming” chapter. A **package** groups related modules. In this book, you’ll work with many preexisting modules and packages, and you’ll create your own modules—in fact, every Python source-code (.py) file you create is a module. Creating packages is beyond this book’s scope. They’re typically used to organize a large library’s functionality into smaller subsets that are easier to maintain and can be imported separately for convenience. For example, the `matplotlib` visualization library that we use in Section 5.17 has extensive

functionality (its documentation is over 2300 pages), so we'll import only the subsets we need in our examples (`pyplot` and `animation`).

The Python Standard Library is provided with the core Python language. Its packages and modules contain capabilities for a wide variety of everyday programming tasks.² You can see a complete list of the standard library modules at

<https://docs.python.org/3/library/>

You've already used capabilities from the `decimal`, `statistics` and `random` modules. In the next section, you'll use mathematics capabilities from the `math` module. You'll see many other Python Standard Library modules throughout the book's examples and exercises, including many of those in the following table:

Some popular Python Standard Library modules

<code>collections</code> —Data structures beyond lists, tuples, dictionaries and sets.	<code>math</code> —Common math constants and operations.
<code>Cryptography</code> modules—Encrypting data for secure transmission.	<code>os</code> —Interacting with the operating system.
<code>csv</code> —Processing comma-separated value files (like those in Excel).	<code>profile</code> , <code>pstats</code> , <code>timeit</code> —Performance analysis.
<code>datetime</code> —Date and time manipulations. Also modules <code>time</code> and <code>calendar</code> .	<code>random</code> —Pseudorandom numbers.
<code>decimal</code> —Fixed-point and floating-point arithmetic, including monetary calculations.	<code>re</code> —Regular expressions for pattern matching.
<code>doctest</code> —Embed validation tests and expected results in docstrings for simple unit testing.	<code>sqlite3</code> —SQLite relational database access.
<code>gettext</code> and <code>locale</code> —Internationalization and localization modules.	<code>statistics</code> —Mathematical statistics functions such as <code>mean</code> , <code>median</code> , <code>mode</code> and <code>variance</code> .
<code>json</code> —JavaScript Object Notation (JSON) processing used with web services and NoSQL document databases.	<code>string</code> —String processing.
	<code>sys</code> —Command-line argument processing; standard input, standard output and standard error streams.
	<code>tkinter</code> —Graphical user interfaces (GUIs) and canvas-based graphics.
	<code>turtle</code> —Turtle graphics.
	<code>webbrowser</code> —For conveniently displaying web pages in Python apps.



Self Check

1 *(Fill-In)* A(n) _____ defines related functions, data and classes. A(n) _____ groups related modules.

Answer: module, package.

2 *(Fill-In)* Every Python source code (.py) file you create is a(n) _____.

Answer: module.

4.7 math Module Functions

The `math module` defines functions for performing various common mathematical calculations. Recall from the previous chapter that an `import` statement of the following form enables you to use a module's definitions via the module's name and a dot (.):

In [1]: `import math`

2. The Python Tutorial refers to this as the “batteries included” approach.

For example, the following snippet calculates the square root of 900 by calling the `math` module's **sqrt function**, which returns its result as a `float` value:

```
In [2]: math.sqrt(900)
Out[2]: 30.0
```

Similarly, the following snippet calculates the absolute value of -10 by calling the `math` module's **fabs function**, which returns its result as a `float` value:

```
In [3]: math.fabs(-10)
Out[3]: 10.0
```

Some `math` module functions are summarized below—you can view the complete list at

<https://docs.python.org/3/library/math.html>

Function	Description	Example
<code>ceil(x)</code>	Rounds x to the smallest integer not less than x	<code>ceil(9.2)</code> is 10.0 <code>ceil(-9.8)</code> is -9.0
<code>floor(x)</code>	Rounds x to the largest integer not greater than x	<code>floor(9.2)</code> is 9.0 <code>floor(-9.8)</code> is -10.0
<code>sin(x)</code>	Trigonometric sine of x (x in radians)	<code>sin(0.0)</code> is 0.0
<code>cos(x)</code>	Trigonometric cosine of x (x in radians)	<code>cos(0.0)</code> is 1.0
<code>tan(x)</code>	Trigonometric tangent of x (x in radians)	<code>tan(0.0)</code> is 0.0
<code>exp(x)</code>	Exponential function e^x	<code>exp(1.0)</code> is 2.718282 <code>exp(2.0)</code> is 7.389056
<code>log(x)</code>	Natural logarithm of x (base e)	<code>log(2.718282)</code> is 1.0 <code>log(7.389056)</code> is 2.0
<code>log10(x)</code>	Logarithm of x (base 10)	<code>log10(10.0)</code> is 1.0 <code>log10(100.0)</code> is 2.0
<code>pow(x, y)</code>	x raised to power y (x^y)	<code>pow(2.0, 7.0)</code> is 128.0 <code>pow(9.0, .5)</code> is 3.0
<code>sqrt(x)</code>	square root of x	<code>sqrt(900.0)</code> is 30.0 <code>sqrt(9.0)</code> is 3.0
<code>fabs(x)</code>	Absolute value of x —always returns a float. Python also has the built-in function <code>abs</code> , which returns an <code>int</code> or a <code>float</code> , based on its argument.	<code>fabs(5.1)</code> is 5.1 <code>fabs(-5.1)</code> is 5.1
<code>fmod(x, y)</code>	Remainder of x/y as a floating-point number	<code>fmod(9.8, 4.0)</code> is 1.8

4.8 Using IPython Tab Completion for Discovery

You can view a module's documentation in IPython interactive mode via **tab completion**—a **discovery** feature that speeds your coding and learning processes. After you type a portion of an identifier and press *Tab*, IPython completes the identifier for you or provides a list of identifiers that begin with what you've typed so far. This may vary based on your operating system platform and what you have imported into your IPython session:

```
In [1]: import math
```

```
In [2]: ma<Tab>
      map          %macro      %%markdown
      math          %magic       %matplotlib
      max()        %man
```

You can scroll through the identifiers with the up and down arrow keys. As you do, IPython highlights an identifier and shows it to the right of the In [] prompt.

Viewing Identifiers in a Module

To view a list of identifiers defined in a module, type the module's name and a dot (.), then press *Tab*:

```
In [3]: math.<Tab>
      acos()      atan()      copysign()   e           expm1()
      acosh()     atan2()     cos()         erf()       fabs()
      asin()       atanh()    cosh()        erfc()     factorial() >
      asinh()     ceil()      degrees()    exp()      floor()
```

If there are more identifiers to display than are currently shown, IPython displays the > symbol (on some platforms) at the right edge, in this case to the right of `factorial()`. You can use the up and down arrow keys to scroll through the list. In the list of identifiers:

- Those followed by parentheses are functions (or methods, as you'll see later).
- Single-word identifiers (such as `Employee`) that begin with an uppercase letter and multiword identifiers in which each word begins with an uppercase letter (such as `CommissionEmployee`) represent class names (there are none in the preceding list). This naming convention, which the *Style Guide for Python Code* recommends, is known as **CamelCase** because the uppercase letters stand out like a camel's humps.
- Lowercase identifiers without parentheses, such as `pi` (not shown in the preceding list) and `e`, are variables. The identifier `pi` evaluates to `3.141592653589793`, and the identifier `e` evaluates to `2.718281828459045`. In the `math` module, `pi` and `e` represent the mathematical constants π and e , respectively.

Python does not have *constants*, although many objects in Python are immutable (non-modifiable). So even though `pi` and `e` are real-world constants, *you must not assign new values to them*, because that would change their values. To help distinguish constants from other variables, the style guide recommends naming your custom constants with all capital letters.

Using the Currently Highlighted Function

As you navigate through the identifiers, if you wish to use a currently highlighted function, simply start typing its arguments in parentheses. IPython then hides the completion list. If you need more information about the currently highlighted item, you can view its docstring by typing a question mark (?) following the name and pressing *Enter* to view the help documentation. The following shows the `fabs` function's docstring:

```
In [4]: math.fabs?  
Docstring:  
fabs(x)  
  
Return the absolute value of the float x.  
Type: builtin_function_or_method
```

The `builtin_function_or_method` shown above indicates that `fabs` is part of a Python Standard Library module. Such modules are considered to be built into Python. In this case, `fabs` is a built-in function from the `math` module.



Self Check

- 1 (*True/False*) In IPython interactive mode, to view a list of identifiers defined in a module, type the module's name and a dot (.) then press *Enter*.

Answer: False. Press *Tab*, not *Enter*.

- 2 (*True/False*) Python does not have constants.

Answer: True.

4.9 Default Parameter Values

When defining a function, you can specify that a parameter has a **default parameter value**. When calling the function, if you omit the argument for a parameter with a default parameter value, the default value for that parameter is automatically passed. Let's define a function `rectangle_area` with default parameter values:

```
In [1]: def rectangle_area(length=2, width=3):  
...:     """Return a rectangle's area."""  
...:     return length * width  
...:
```

You specify a default parameter value by following a parameter's name with an = and a value—in this case, the default parameter values are 2 and 3 for `length` and `width`, respectively. Any parameters with default parameter values must appear in the parameter list to the *right* of parameters that do not have defaults.

The following call to `rectangle_area` has no arguments, so IPython uses both default parameter values as if you had called `rectangle_area(2, 3)`:

```
In [2]: rectangle_area()  
Out[2]: 6
```

The following call to `rectangle_area` has only one argument. Arguments are assigned to parameters from left to right, so 10 is used as the `length`. The interpreter passes the default parameter value 3 for the `width` as if you had called `rectangle_area(10, 3)`:

```
In [3]: rectangle_area(10)  
Out[3]: 30
```

The following call to `rectangle_area` has arguments for both `length` and `width`, so IPython ignores the default parameter values:

```
In [4]: rectangle_area(10, 5)  
Out[4]: 50
```



Self Check

- 1** (*True/False*) When an argument with a default parameter value is omitted in a function call, the interpreter automatically passes the default parameter value in the call.

Answer: True.

- 2** (*True/False*) Parameters with default parameter values must be the *leftmost* arguments in a function's parameter list.

Answer: False. Parameters with default parameter values must appear to the *right* of parameters that do not have defaults.

4.10 Keyword Arguments

When calling functions, you can use **keyword arguments** to pass arguments in *any* order. To demonstrate keyword arguments, we redefine the `rectangle_area` function—this time without default parameter values:

```
In [1]: def rectangle_area(length, width):
    ....
    """Return a rectangle's area."""
    ....
    return length * width
    ....
```

Each keyword *argument in a call* has the form `parametername=value`. The following call shows that the order of keyword arguments does not matter—they do not need to match the corresponding parameters' positions in the function definition:

```
In [2]: rectangle_area(width=5, length=10)
Out[3]: 50
```

In each function call, you must place keyword arguments *after* a function's positional arguments—that is, any arguments for which you do not specify the parameter name. Such arguments are assigned to the function's parameters left-to-right, based on the argument's positions in the argument list. Keyword arguments are also helpful for improving the readability of function calls, especially for functions with many arguments.



Self Check

- 1** (*True/False*) You must pass keyword arguments in the same order as their corresponding parameters in the function definition's parameter list.

Answer: False. The order of keyword arguments does not matter.

4.11 Arbitrary Argument Lists

Functions with **arbitrary argument lists**, such as built-in functions `min` and `max`, can receive *any* number of arguments. Consider the following `min` call:

```
min(88, 75, 96, 55, 83)
```

The function's documentation states that `min` has two *required* parameters (named `arg1` and `arg2`) and an optional third parameter of the form `*args`, indicating that the function can receive any number of additional arguments. The `*` before the parameter name tells Python to pack any remaining arguments into a tuple that's passed to the `args` parameter. In the call above, parameter `arg1` receives 88, parameter `arg2` receives 75 and parameter `args` receives the tuple (96, 55, 83).

Defining a Function with an Arbitrary Argument List

Let's define an average function that can receive any number of arguments:

```
In [1]: def average(*args):
...:     return sum(args) / len(args)
...:
```

The parameter name `args` is used by convention, but you may use any identifier. If the function has multiple parameters, the `*args` parameter must be the *rightmost* parameter.

Now, let's call `average` several times with arbitrary argument lists of different lengths:

```
In [2]: average(5, 10)
Out[2]: 7.5
```

```
In [3]: average(5, 10, 15)
Out[3]: 10.0
```

```
In [4]: average(5, 10, 15, 20)
Out[4]: 12.5
```

To calculate the average, divide the sum of the `args` tuple's elements (returned by built-in function `sum`) by the tuple's number of elements (returned by built-in function `len`). Note in our `average` definition that if the length of `args` is 0, a `ZeroDivisionError` occurs. In the next chapter, you'll see how to access a tuple's elements without unpacking them.

Passing an Iterable's Individual Elements as Function Arguments

You can unpack a tuple's, list's or other iterable's elements to pass them as individual function arguments. The `*` operator, when applied to an iterable argument in a function call, unpacks its elements. The following code creates a five-element `grades` list, then uses the expression `*grades` to unpack its elements as `average`'s arguments:

```
In [5]: grades = [88, 75, 96, 55, 83]
```

```
In [6]: average(*grades)
Out[6]: 79.4
```

The call shown above is equivalent to `average(88, 75, 96, 55, 83)`.



Self Check

- 1** (*Fill-In*) To define a function with an arbitrary argument list, specify a parameter of the form _____.

Answer: `*args` (again, the name `args` is used by convention, but is not required).

- 2** (*IPython Session*) Create a function named `calculate_product` that receives an arbitrary argument list and returns the product of all the arguments. Call the function with the arguments 10, 20 and 30, then with the sequence of integers produced by `range(1, 6, 2)`.

Answer:

```
In [1]: def calculate_product(*args):
...:     product = 1
...:     for value in args:
...:         product *= value
...:     return product
...:
```

```
In [2]: calculate_product(10, 20, 30)
Out[2]: 6000

In [3]: calculate_product(*range(1, 6, 2))
Out[3]: 15
```

4.12 Methods: Functions That Belong to Objects

A **method** is simply a function that you call on an object using the form

object_name.method_name(arguments)

For example, the following session creates the string variable `s` and assigns it the string object 'Hello'. Then the session calls the object's `lower` and `upper` methods, which produce *new* strings containing all-lowercase and all-uppercase versions of the original string, leaving `s` unchanged:

```
In [1]: s = 'Hello'

In [2]: s.lower() # call lower method on string object s
Out[2]: 'hello'

In [3]: s.upper()
Out[3]: 'HELLO'

In [4]: s
Out[4]: 'Hello'
```

The *Python Standard Library* reference at

<https://docs.python.org/3/library/index.html>

describes the methods of built-in types and the types in the Python Standard Library. In the “Object-Oriented Programming” chapter, you’ll create *custom* types called classes and define custom methods that you can call on objects of those classes.

4.13 Scope Rules

Each identifier has a **scope** that determines where you can use it in your program. For that portion of the program, the identifier is said to be “in scope.”

Local Scope

A local variable’s identifier has **local scope**. It’s “in scope” only from its definition to the end of the function’s block. It “goes out of scope” when the function returns to its caller. So, a local variable can be used only inside the function that defines it.

Global Scope

Identifiers defined outside any function (or class) have **global scope**—these may include functions, variables and classes. Variables with global scope are known as **global variables**. Identifiers with global scope can be used in a .py file or interactive session anywhere after they’re defined.

Accessing a Global Variable from a Function

You can access a global variable's value inside a function:

```
In [1]: x = 7

In [2]: def access_global():
...:     print('x printed from access_global:', x)
...:

In [3]: access_global()
x printed from access_global: 7
```

However, by default, you cannot *modify* a global variable in a function—when you first assign a value to a variable in a function's block, Python creates a *new* local variable:

```
In [4]: def try_to_modify_global():
...:     x = 3.5
...:     print('x printed from try_to_modify_global:', x)
...:

In [5]: try_to_modify_global()
x printed from try_to_modify_global: 3.5

In [6]: x
Out[6]: 7
```

In function `try_to_modify_global`'s block, the local `x` **shadows** the global `x`, making it inaccessible in the scope of the function's block. Snippet [6] shows that global variable `x` still exists and has its original value (7) after function `try_to_modify_global` executes.

To modify a global variable in a function's block, you must use a `global` statement to declare that the variable is defined in the global scope:

```
In [7]: def modify_global():
...:     global x
...:     x = 'hello'
...:     print('x printed from modify_global:', x)
...:

In [8]: modify_global()
x printed from modify_global: hello

In [9]: x
Out[9]: 'hello'
```

Blocks vs. Suites

You've now defined function *blocks* and control statement *suites*. When you create a variable in a block, it's *local* to that block. However, when you create a variable in a control statement's suite, the variable's scope depends on where the control statement is defined:

- If the control statement is in the global scope, then any variables defined in the control statement have global scope.
- If the control statement is in a function's block, then any variables defined in the control statement have local scope.

We'll continue our scope discussion in the "Object-Oriented Programming" chapter when we introduce custom classes.

Shadowing Functions

In the preceding chapters, when summing values, we stored the sum in a variable named `total`. The reason we did this is that `sum` is a built-in function. If you define a variable named `sum`, it *shadows* the built-in function, making it inaccessible in your code. When you execute the following assignment, Python binds the identifier `sum` to the `int` object containing 15. At this point, the identifier `sum` no longer references the built-in function. So, when you try to use `sum` as a function, a `TypeError` occurs:

```
In [10]: sum = 10 + 5

In [11]: sum
Out[11]: 15

In [12]: sum([10, 5])
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-12-1237d97a65fb> in <module>()
----> 1 sum([10, 5])

TypeError: 'int' object is not callable
```

Statements at Global Scope

In the scripts you've seen so far, we've written some statements outside functions at the global scope and some statements inside function blocks. Script statements at global scope execute as soon as they're encountered by the interpreter, whereas statements in a block execute only when the function is called.



Self Check

1 (*Fill-In*) An identifier's _____ describes the region of a program in which the identifier's value can be accessed.

Answer: scope.

2 (*True/False*) Once a code block terminates (e.g., when a function returns), all identifiers defined in that block "go out of scope" and can no longer be accessed.

Answer: True.

4.14 import: A Deeper Look

You've imported modules (such as `math` and `random`) with a statement like:

```
import module_name
```

then accessed their features via each module's name and a dot (.). Also, you've imported a specific identifier from a module (such as the `decimal` module's `Decimal` type) with a statement like:

```
from module_name import identifier
```

then used that identifier without having to precede it with the module name and a dot (.).

Importing Multiple Identifiers from a Module

Using the `from...import` statement you can import a comma-separated list of identifiers from a module then use them in your code without having to precede them with the module name and a dot (.):

```
In [1]: from math import ceil, floor
In [2]: ceil(10.3)
Out[2]: 11
In [3]: floor(10.7)
Out[3]: 10
```

Trying to use a function that's not imported causes a `NameError`, indicating that the name is not defined.

Caution: Avoid Wildcard Imports

You can import *all* identifiers defined in a module with a **wildcard import** of the form

```
from modulename import *
```

This makes all of the module's identifiers available for use in your code. Importing a module's identifiers with a wildcard `import` can lead to subtle errors—it's considered a dangerous practice that you should avoid. Consider the following snippets:

```
In [4]: e = 'hello'
In [5]: from math import *
In [6]: e
Out[6]: 2.718281828459045
```

Initially, we assign the string '`'hello'`' to a variable named `e`. After executing snippet [5] though, the variable `e` is replaced, possibly by accident, with the `math` module's constant `e`, representing the mathematical floating-point value e .

Binding Names for Modules and Module Identifiers

Sometimes it's helpful to import a module and use an abbreviation for it to simplify your code. The `import` statement's `as` clause allows you to specify the name used to reference the module's identifiers. For example, in Section 3.17 we could have imported the `statistics` module and accessed its `mean` function as follows:

```
In [7]: import statistics as stats
In [8]: grades = [85, 93, 45, 87, 93]
In [9]: stats.mean(grades)
Out[9]: 80.6
```

As you'll see in later chapters, `import...as` is frequently used to import Python libraries with convenient abbreviations, like `stats` for the `statistics` module. As another example, we'll use the `numpy` module which typically is imported with

```
import numpy as np
```

Library documentation often mentions popular shorthand names.

Typically, when importing a module, you should use `import` or `import...` as statements, then access the module through the module name or the abbreviation following the `as` keyword, respectively. This ensures that you do not accidentally import an identifier that conflicts with one in your code.



Self Check

1 (*True/False*) You must always import all the identifiers of a given module.

Answer: False. You can import only the identifiers you need by using a `from...import` statement.

2 (*IPython Session*) Import the `decimal` module with the shorthand name `dec`, then create a `Decimal` object with the value `2.5` and square its value.

Answer:

```
In [1]: import decimal as dec
```

```
In [2]: dec.Decimal('2.5') ** 2
```

```
Out[2]: Decimal('6.25')
```

4.15 Passing Arguments to Functions: A Deeper Look

Let's take a closer look at how arguments are passed to functions. In many programming languages, there are two ways to pass arguments—**pass-by-value** and **pass-by-reference** (sometimes called **call-by-value** and **call-by-reference**, respectively):

- With pass-by-value, the called function receives a *copy* of the argument's *value* and works exclusively with that copy. Changes to the function's copy do *not* affect the original variable's value in the caller.
- With pass-by-reference, the called function can access the argument's value in the caller directly and modify the value if it's mutable.

Python arguments are always passed by reference. Some people call this **pass-by-object-reference**, because “everything in Python is an object.”³ When a function call provides an argument, Python copies the argument object's *reference*—not the object itself—into the corresponding parameter. This is important for performance. Functions often manipulate large objects—frequently copying them would consume large amounts of computer memory and significantly slow program performance.

Memory Addresses, References and “Pointers”

You interact with an object via a reference, which behind the scenes is that object's address (or location) in the computer's memory—sometimes called a “pointer” in other languages. After an assignment like

```
x = 7
```

the variable `x` does not actually contain the value `7`. Rather, it contains a reference to an *object* containing `7` (and some other data we'll discuss in later chapters) stored *elsewhere* in

3. Even the functions you defined in this chapter and the classes (custom types) you'll define in later chapters are objects in Python.

memory. You might say that `x` “points to” (that is, references) the object containing 7, as in the diagram below:



Built-In Function `id` and Object Identities

Let’s consider how we pass arguments to functions. First, let’s create the integer variable `x` mentioned above—shortly we’ll use `x` as a function argument:

```
In [1]: x = 7
```

Now `x` refers to (or “points to”) the integer object containing 7. No two separate objects can reside at the same address in memory, so every object in memory has a *unique address*. Though we can’t see an object’s address, we can use the built-in **`id` function** to obtain a *unique int* value which identifies only that object while it remains in memory (you’ll likely get a different value when you run this on your computer):

```
In [2]: id(x)
Out[2]: 4350477840
```

The integer result of calling `id` is known as the object’s **`identity`**.⁴ No two objects in memory can have the same *identity*. We’ll use object identities to demonstrate that objects are passed by reference.

Passing an Object to a Function

Let’s define a `cube` function that displays its parameter’s identity, then returns the parameter’s value cubed:

```
In [3]: def cube(number):
...:     print('id(number):', id(number))
...:     return number ** 3
...:
```

Next, let’s call `cube` with the argument `x`, which refers to the integer object containing 7:

```
In [4]: cube(x)
id(number): 4350477840
Out[4]: 343
```

The identity displayed for `cube`’s parameter `number`—4350477840—is the *same* as that displayed for `x` previously. Since every object has a unique identity, both the *argument* `x` and the *parameter* `number` refer to the *same object* while `cube` executes. So when function `cube` uses its parameter `number` in its calculation, it gets the value of `number` from the original object in the caller.

Testing Object Identities with the `is` Operator

You also can prove that the argument and the parameter refer to the same object with Python’s **`is operator`**, which returns `True` if its two operands have the *same identity*:

4. According to the Python documentation, depending on the Python implementation you’re using, an object’s identity may be the object’s actual memory address, but this is not required.

```
In [5]: def cube(number):
....:     print('number is x:', number is x) # x is a global variable
....:     return number ** 3
....:

In [6]: cube(x)
number is x: True
Out[6]: 343
```

Immutable Objects as Arguments

When a function receives as an argument a reference to an *immutable* (unmodifiable) object—such as an `int`, `float`, `string` or `tuple`—even though you have direct access to the original object in the caller, you cannot modify the original immutable object’s value. To prove this, first let’s have `cube` display `id(number)` before and after assigning a new object to the parameter `number` via an augmented assignment:

```
In [7]: def cube(number):
....:     print('id(number) before modifying number:', id(number))
....:     number **= 3
....:     print('id(number) after modifying number:', id(number))
....:     return number
....:

In [8]: cube(x)
id(number) before modifying number: 4350477840
id(number) after modifying number: 4396653744
Out[8]: 343
```

When we call `cube(x)`, the first `print` statement shows that `id(number)` initially is the same as `id(x)` in snippet [2]. Numeric values are immutable, so the statement

`number **= 3`

actually creates a *new object* containing the cubed value, then assigns that object’s reference to parameter `number`. Recall that if there are no more references to the *original* object, it will be *garbage collected*. Function `cube`’s second `print` statement shows the *new* object’s identity. Object identities must be unique, so `number` must refer to a *different* object. To show that `x` was not modified, we display its value and identity again:

```
In [9]: print(f'x = {x}; id(x) = {id(x)}')
x = 7; id(x) = 4350477840
```

Mutable Objects as Arguments

In the next chapter, we’ll show that when a reference to a *mutable* object like a list is passed to a function, the function *can* modify the original object in the caller.



Self Check

1 (*Fill-In*) The built-in function _____ returns an object’s unique identifier.

Answer: `id`.

2 (*True/False*) Attempts to modify mutable objects create new objects.

Answer: False. This is true for immutable objects.

3 (IPython Session) Create a variable `width` with the value `15.5`, then show that modifying the variable creates a new object. Display `width`'s identity and value before and after modifying its value.

Answer:

```
In [1]: width = 15.5

In [2]: print('id:', id(width), ' value:', width)
id: 4397553776  value: 15.5

In [3]: width = width * 3

In [4]: print('id:', id(width), ' value:', width)
id: 4397554208  value: 46.5
```

4.16 Function-Call Stack

To understand how Python performs function calls, consider a data structure (that is, a collection of related data items) known as a **stack**, which is like a pile of dishes. When you add a dish to the pile, you place it on the *top*. Similarly, when you remove a dish from the pile, you take it from the top. Stacks are known as **last-in, first-out (LIFO) data structures**—the last item **pushed** (that is, placed) onto the stack is the first item **popped** (that, is removed) from the stack.

Stacks and Your Web Browser's Back Button

A stack is working for you when you visit websites with your web browser. A stack of web-page addresses supports a browser's back button. For each new web page you visit, the browser *pushes* the address of the page you were viewing onto the back button's stack. This allows the browser to "remember" the web page you came from if you decide to go back to it later. Pushing onto the back button's stack may happen many times before you decide to go back to a previous web page. When you press the browser's back button, the browser *pops* the top stack element to get the prior web page's address, then displays that web page. Each time you press the back button the browser *pops* the top stack element and displays that page. This continues until the stack is empty, meaning that there are no more pages for you to go back to via the back button.

Stack Frames

Similarly, the **function-call stack** supports the function call/return mechanism. Eventually, each function must return program control to the point at which it was called. For each function call, the interpreter *pushes* an entry called a **stack frame** (or an **activation record**) onto the stack. This entry contains the *return location* that the called function needs so it can return control to its caller. When the function finishes executing, the interpreter *pops* the function's stack frame, and control transfers to the *return location* that was popped.

The *top* stack frame *always* contains the information the currently executing function needs to return control to its caller. If before a function returns it makes a call to another function, the interpreter *pushes* a stack frame for that function call onto the stack. Thus, the return address required by the newly called function to return to its caller is now on *top* of the stack.

Local Variables and Stack Frames

Most functions have one or more *parameters* and possibly *local variables* that need to:

- exist while the function is executing,
- remain active if the function makes calls to other functions, and
- “go away” when the function returns to its caller.

A called function’s stack frame is the perfect place to reserve memory for the function’s local variables. That stack frame is pushed when the function is called and exists while the function is executing. When that function returns, it no longer needs its local variables, so its stack frame is *popped* from the stack, and its local variables no longer exist.

Stack Overflow

Of course, the amount of memory in a computer is finite, so only a certain amount of memory can be used to store stack frames on the function-call stack. If the function-call stack runs out of memory as a result of too many function calls, a fatal error known as **stack overflow** occurs.⁵ Stack overflows actually are rare unless you have a logic error that keeps calling functions that never return.

Principle of Least Privilege

The **principle of least privilege** is fundamental to good software engineering. It states that code should be granted *only* the amount of privilege and access that it needs to accomplish its designated task, but no more. An example of this is the scope of a local variable, which should not be visible when it’s not needed. This is why a function’s local variables are placed in stack frames on the function-call stack, so they can be used by that function while it executes and go away when it returns. Once the stack frame is popped, the memory that was occupied by it can be reused for new stack frames. Also, there is no access between stack frames, so functions cannot see each other’s local variables. The principle of least privilege makes your programs more robust by preventing code from accidentally (or maliciously) modifying variable values that should not be accessible to it.



Self Check

1 (Fill-In) The stack operations for adding an item to a stack and removing an item from a stack are known as _____ and _____, respectively.

Answer: push, pop.

2 (Fill-In) A stack’s items are removed in _____ order.

Answer: last-in, first-out (LIFO).

4.17 Functional-Style Programming

Like other popular languages, such as Java and C#, Python is not a purely functional language. Rather, it offers “functional-style” features that help you write code which is less likely to contain errors, more concise and easier to read, debug and modify. Functional-style programs also can be easier to parallelize to get better performance on today’s multi-

5. This is how the website stackoverflow.com got its name—a good website for getting answers to your programming questions.

core processors. The chart below lists most of Python’s key functional-style programming capabilities and shows in parentheses the chapters in which we initially cover many of them.

Functional-style programming topics

avoiding side effects	generator functions (12)	lazy evaluation (5)
closures	higher-order functions (5)	list comprehensions (5)
declarative programming (4)	immutability (4)	operator module (5, 13, 18)
decorators (10)	internal iteration (4)	pure functions (4)
dictionary comprehensions (6)	iterators (3)	range function (3, 4)
filter/map/reduce (5)	itertools module (18)	reductions (3, 5)
functools module	lambda expressions (5)	set comprehensions (6)
generator expressions (5)		

We cover most of these features throughout the book—many with code examples and others from a literacy perspective. You’ve already used list, string and built-in function `range` *iterators* with the `for` statement, and several *reductions* (functions `sum`, `len`, `min` and `max`). We discuss declarative programming, immutability and internal iteration below.

What vs. How

As the tasks you perform get more complicated, your code can become harder to read, debug and modify, and more likely to contain errors. Specifying *how* the code works can become complex.

Functional-style programming lets you simply say *what* you want to do. It hides many details of *how* to perform each task. Typically, library code handles the *how* for you. As you’ll see, this can eliminate many errors.

Consider the `for` statement in many other programming languages. Typically, *you* must specify all the details of counter-controlled iteration: a control variable, its initial value, how to increment it and a loop-continuation condition that uses the control variable to determine whether to continue iterating. This style of iteration is known as **external iteration** and is error-prone. For example, you might provide an incorrect initializer, increment or loop-continuation condition. External iteration **mutates** (that is, modifies) the control variable, and the `for` statement’s suite often mutates other variables as well. Every time you modify variables you could introduce errors. Functional-style programming emphasizes **immutability**. That is, it avoids operations that modify variables’ values. We’ll say more in the next chapter.

Python’s `for` statement and `range` function *hide* most counter-controlled iteration details. You specify *what* values `range` should produce and the variable that should receive each value as it’s produced. Function `range` *knows how* to produce those values. Similarly, the `for` statement *knows how* to get each value from `range` and *how* to stop iterating when there are no more values. Specifying *what*, but not *how*, is an important aspect of **internal iteration**—a key functional-style programming concept.

The Python built-in functions `sum`, `min` and `max` each use internal iteration. To total the elements of the list `grades`, you simply declare *what* you want to do—that is, `sum(grades)`. Function `sum` *knows how* to iterate through the list and add each element to

the running total. Stating what you *want* done rather than programming *how* to do it is known as **declarative programming**.

Pure Functions

In pure functional programming language you focus on writing pure functions. A **pure function's** result depends only on the argument(s) you pass to it. Also, given a particular argument (or arguments), a pure function always produces the same result. For example, built-in function `sum`'s return value depends only on the iterable you pass to it. Given a list `[1, 2, 3]`, `sum` *always* returns 6 no matter how many times you call it. Also, a pure function does not have *side effects*. For example, even if you pass a *mutable* list to a pure function, the list will contain the same values before and after the function call. When you call the pure function `sum`, it does not modify its argument.

```
In [1]: values = [1, 2, 3]

In [2]: sum(values)
Out[2]: 6

In [3]: sum(values) # same call always returns same result
Out[3]: 6

In [4]: values
Out[5]: [1, 2, 3]
```

In the next chapter, we'll continue using functional-style programming concepts. Also, you'll see that *functions are objects* that you can pass to other functions as data.

4.18 Intro to Data Science: Measures of Dispersion

In our discussion of descriptive statistics, we've considered the measures of central tendency—mean, median and mode. These help us categorize typical values in a group—such as the mean height of your classmates or the most frequently purchased car brand (the mode) in a given country.

When we're talking about a group, the entire group is called the **population**. Sometimes a population is quite large, such as the people likely to vote in the next U.S. presidential election, which is a number in excess of 100,000,000 people. For practical reasons, the polling organizations trying to predict who will become the next president work with carefully selected small subsets of the population known as **samples**. Many of the polls in the 2016 election had sample sizes of about 1000 people.

In this section, we continue discussing basic descriptive statistics. We introduce **measures of dispersion** (also called **measures of variability**) that help you understand how spread out the values are. For example, in a class of students, there may be a bunch of students whose height is close to the average, with smaller numbers of students who are considerably shorter or taller.

For our purposes, we'll calculate each measure of dispersion both by hand and with functions from the module `statistics`, using the following population of 10 six-sided die rolls:

1, 3, 4, 2, 6, 5, 3, 4, 5, 2

Variance

To determine the **variance**,⁶ we begin with the mean of these values—3.5. You obtain this result by dividing the sum of the face values, 35, by the number of rolls, 10. Next, we subtract the mean from every die value (this produces some negative results):

```
-2.5, -0.5, 0.5, -1.5, 2.5, 1.5, -0.5, 0.5, 1.5, -1.5
```

Then, we square each of these results (yielding only positives):

```
6.25, 0.25, 0.25, 2.25, 6.25, 2.25, 0.25, 0.25, 2.25, 2.25
```

Finally, we calculate the mean of these squares, which is 2.25 ($22.5 / 10$)—this is the **population variance**. Squaring the difference between each die value and the mean of all die values emphasizes **outliers**—the values that are farthest from the mean. As we get deeper into data analytics, sometimes we'll want to pay careful attention to outliers, and sometimes we'll want to ignore them. The following code uses the `statistics` module's **pvariance** function to confirm our manual result:

```
In [1]: import statistics
```

```
In [2]: statistics.pvariance([1, 3, 4, 2, 6, 5, 3, 4, 5, 2])  
Out[2]: 2.25
```

Standard Deviation

The **standard deviation** is the square root of the variance (in this case, 1.5), which tones down the effect of the outliers. The smaller the variance and standard deviation are, the closer the data values are to the mean and the less overall **dispersion** (that is, **spread**) there is between the values and the mean. The following code calculates the **population standard deviation** with the `statistics` module's **pstdev** function, confirming our manual result:

```
In [3]: statistics.pstdev([1, 3, 4, 2, 6, 5, 3, 4, 5, 2])  
Out[3]: 1.5
```

Passing the `pvariance` function's result to the `math` module's `sqrt` function confirms our result of 1.5:

```
In [4]: import math
```

```
In [5]: math.sqrt(statistics.pvariance([1, 3, 4, 2, 6, 5, 3, 4, 5, 2]))  
Out[5]: 1.5
```

Advantage of Population Standard Deviation vs. Population Variance

Suppose you've recorded the March Fahrenheit temperatures in your area. You might have 31 numbers such as 19, 32, 28 and 35. The units for these numbers are degrees. When you square your temperatures to calculate the population variance, the units of the population variance become "degrees squared." When you take the square root of the popula-

6. For simplicity, we're calculating the *population variance*. There is a subtle difference between the *population variance* and the *sample variance*. Instead of dividing by n (the number of die rolls in our example), sample variance divides by $n - 1$. The difference is pronounced for small samples and becomes insignificant as the sample size increases. The `statistics` module provides the functions `pvariance` and `variance` to calculate the population variance and sample variance, respectively. Similarly, the `statistics` module provides the functions `pstdev` and `stdev` to calculate the population standard deviation and sample standard deviation, respectively.

tion variance to calculate the population standard deviation, the units once again become degrees, which are the *same* units as your temperatures.



Self Check

1 (*Discussion*) Why do we often work with a sample rather than the full population?

Answer: Because often the full population is unmanageably large.

2 (*True/False*) An advantage of the population variance over the population standard deviation is that its units are the same as the sample values' units.

Answer: False. This is an advantage of population standard deviation over population variance.

3 (*IPython Session*) In this section, we worked with population variance and population standard deviation. There is a subtle difference between the *population variance* and the *sample variance*. In our example, instead of dividing by 10 (the number of die rolls), sample variance would divide by 9 (which is one less than the sample size). The difference is pronounced for small samples but becomes insignificant as the sample size increases. The `statistics` module provides the functions `variance` and `stdev` to calculate the sample variance and sample standard deviation, respectively. Redo the manual calculations, then use the `statistics` module's functions to confirm this difference between the two methods of calculation.

Answer:

```
In [1]: import statistics
In [2]: statistics.variance([1, 3, 4, 2, 6, 5, 3, 4, 5, 2])
Out[2]: 2.5
In [3]: statistics.stdev([1, 3, 4, 2, 6, 5, 3, 4, 5, 2])
Out[3]: 1.5811388300841898
```

4.19 Wrap-Up

In this chapter, we created custom functions. We imported capabilities from the `random` and `math` modules. We introduced random-number generation and used it to simulate rolling a six-sided die. We packed multiple values into tuples to return more than one value from a function. We also unpacked a tuple to access its values. We discussed using the Python Standard Library's modules to avoid “reinventing the wheel.”

We created functions with default parameter values and called functions with keyword arguments. We also defined functions with arbitrary argument lists. We called methods of objects. We discussed how an identifier's scope determines where in your program you can use it.

You learned more about importing modules. You saw that arguments are passed-by-reference to functions, and how the function-call stack and stack frames support the function-call-and-return mechanism. We've introduced basic list and tuple capabilities over the last two chapters—in the next chapter, we'll discuss them in detail.

Finally, we continued our discussion of descriptive statistics by introducing measures of dispersion—variance and standard deviation—and calculating them with functions from the Python Standard Library's `statistics` module.

For some types of problems, it's useful to have functions call themselves. A **recursive function** calls itself, either directly or indirectly through another function. Recursion is an important topic discussed at length in upper-level computer science courses. We include a detailed treatment in the chapter "Computer Science Thinking: Recursion, Searching, Sorting and Big O."

Exercises

Unless specified otherwise, use IPython sessions for each exercise.

4.1 (*Discussion: else Clause*) In the script of Fig. 4.1, we did not include an `else` clause in the `if...elif` statement. What are the possible consequences of this choice?

4.2 (*Discussion: Function-Call Stack*) What happens if you keep pushing onto a stack, without enough popping?

4.3 (*What's Wrong with This Code?*) What is wrong with the following `cube` function's definition?

```
def cube(x):
    """Calculate the cube of x."""
    x ** 3
    print('The cube of 2 is', cube(2))
```

4.4 (*What's Does This Code Do?*) What does the following `mystery` function do? Assume you pass the list `[1, 2, 3, 4, 5]` as an argument.

```
def mystery(x):
    y = 0
    for value in x:
        y += value ** 2
    return y
```

4.5 (*Fill in the Missing Code?*) Replace the `***`'s in the `seconds_since_midnight` function so that it returns the number of seconds since midnight. The function should receive three integers representing the current time of day. Assume that the hour is a value from 0 (midnight) through 23 (11 PM) and that the minute and second are values from 0 to 59. Test your function with actual times. For example, if you call the function for 1:30:45 PM by passing 13, 30 and 45, the function should return 48645.

```
def seconds_since_midnight(***)�
    hour_in_seconds = ***
    minute_in_seconds = ***
    return ***
```

4.6 (*Modified average Function*) The `average` function we defined in Section 4.11 can receive any number of arguments. If you call it with no arguments, however, the function causes a `ZeroDivisionError`. Reimplement `average` to receive one required argument *and* the arbitrary argument list argument `*args`, and update its calculation accordingly. Test your function. The function will always require at least one argument, so you'll no longer be able to get a `ZeroDivisionError`. When you call `average` with no arguments, Python should issue a `TypeError` indicating "average() missing 1 required positional argument."

4.7 (Date and Time) Python's `datetime` module contains a `datetime` type with a method `today` that returns the current date and time as a `datetime` object. Write a *parameterless* `date_and_time` function containing the following statement, then call that function to display the current date and time:

```
print(datetime.datetime.today())
```

On our system, the date and time display in the following format:

```
2018-06-08 13:04:19.214180
```

4.8 (Rounding Numbers) Investigate built-in function `round` at

<https://docs.python.org/3/library/functions.html#round>

then use it to round the `float` value `13.56449` to the nearest integer, tenths, hundredths and thousandths positions.

4.9 (Temperature Conversion) Implement a `fahrenheit` function that returns the Fahrenheit equivalent of a Celsius temperature. Use the following formula:

$$F = (9 / 5) * C + 32$$

Use this function to print a chart showing the Fahrenheit equivalents of all Celsius temperatures in the range 0–100 degrees. Use one digit of precision for the results. Print the outputs in a neat tabular format.

4.10 (Guess the Number) Write a script that plays “guess the number.” Choose the number to be guessed by selecting a random integer in the range 1 to 1000. Do not reveal this number to the user. Display the prompt “Guess my number between 1 and 1000 with the fewest guesses:”. The player inputs a first guess. If the guess is incorrect, display “Too high. Try again.” or “Too low. Try again.” as appropriate to help the player “zero in” on the correct answer, then prompt the user for the next guess. When the user enters the correct answer, display “Congratulations. You guessed the number!”, and allow the user to choose whether to play again.

4.11 (Guess-the-Number Modification) Modify the previous exercise to count the number of guesses the player makes. If the number is 10 or fewer, display “Either you know the secret or you got lucky!” If the player makes more than 10 guesses, display “You should be able to do better!” Why should it take no more than 10 guesses? Well, with each “good guess,” the player should be able to eliminate half of the numbers, then half of the remaining numbers, and so on. Doing this 10 times narrows down the possibilities to a single number. This kind of “halving” appears in many computer science applications. For example, in the “Computer Science Thinking: Recursion, Searching, Sorting and Big O” chapter, we’ll present the high-speed binary search and merge sort algorithms, and you’ll attempt the quicksort exercise—each of these cleverly uses halving to achieve high performance.

4.12 (Simulation: The Tortoise and the Hare) In this problem, you’ll re-create the classic race of the tortoise and the hare. You’ll use random-number generation to develop a simulation of this memorable event.

Our contenders begin the race at square 1 of 70 squares. Each square represents a position along the race course. The finish line is at square 70. The first contender to reach or pass square 70 is rewarded with a pail of fresh carrots and lettuce. The course weaves its way up the side of a slippery mountain, so occasionally the contenders lose ground.

A clock ticks once per second. With each tick of the clock, your application should adjust the position of the animals according to the rules in the table below. Use variables to keep track of the positions of the animals (i.e., position numbers are 1–70). Start each animal at position 1 (the “starting gate”). If an animal slips left before square 1, move it back to square 1.

Animal	Move type	Percentage of the time	Actual move
Tortoise	Fast plod	50%	3 squares to the right
	Slip	20%	6 squares to the left
	Slow plod	30%	1 square to the right
Hare	Sleep	20%	No move at all
	Big hop	20%	9 squares to the right
	Big slip	10%	12 squares to the left
	Small hop	30%	1 square to the right
	Small slip	20%	2 squares to the left

Create two functions that generate the percentages in the table for the tortoise and the hare, respectively, by producing a random integer i in the range $1 \leq i \leq 10$. In the function for the tortoise, perform a “fast plod” when $1 \leq i \leq 5$, a “slip” when $6 \leq i \leq 7$ or a “slow plod” when $8 \leq i \leq 10$. Use a similar technique in the function for the hare.

Begin the race by displaying

BANG !!!!
AND THEY'RE OFF !!!!

Then, for each tick of the clock (i.e., each iteration of a loop), display a 70-position line showing the letter “T” in the position of the tortoise and the letter “H” in the position of the hare. Occasionally, the contenders will land on the same square. In this case, the tortoise bites the hare, and your application should display “OUCH!!!” at that position. All positions other than the “T”, the “H” or the “OUCH!!!” (in case of a tie) should be blank.

After each line is displayed, test for whether either animal has reached or passed square 70. If so, display the winner and terminate the simulation. If the tortoise wins, display TORTOISE WINS!!! YAY!!! If the hare wins, display Hare wins. Yuch. If both animals win on the same tick of the clock, you may want to favor the tortoise (the “underdog”), or you may want to display “It's a tie”. If neither animal wins, perform the loop again to simulate the next tick of the clock. When you're ready to run your application, assemble a group of fans to watch the race. You'll be amazed at how involved your audience gets!

4.13 (Arbitrary Argument List) Calculate the product of a series of integers that are passed to the function `product`, which receives an arbitrary argument list. Test your function with several calls, each with a different number of arguments.

4.14 (Computer-Assisted Instruction) Computer-assisted instruction (CAI) refers to the use of computers in education. Write a script to help an elementary school student learn multiplication. Create a function that randomly generates and returns a tuple of two pos-

itive one-digit integers. Use that function's result in your script to prompt the user with a question, such as

How much is 6 times 7?

For a correct answer, display the message "Very good!" and ask another multiplication question. For an incorrect answer, display the message "No. Please try again." and let the student try the same question repeatedly until the student finally gets it right.

4.15 (*Computer-Assisted Instruction: Reducing Student Fatigue*) Varying the computer's responses can help hold the student's attention. Modify the previous exercise so that various comments are displayed for each answer. Possible responses to a correct answer should include 'Very good!', 'Nice work!', and 'Keep up the good work!' Possible responses to an incorrect answer should include 'No. Please try again.', 'Wrong. Try once more.' and 'No. Keep trying.' Choose a number from 1 to 3, then use that value to select one of the three appropriate responses to each correct or incorrect answer.

4.16 (*Computer-Assisted Instruction: Difficulty Levels*) Modify the previous exercise to allow the user to enter a difficulty level. At a difficulty level of 1, the program should use only single-digit numbers in the problems and at a difficulty level of 2, numbers as large as two digits.

4.17 (*Computer-Assisted Instruction: Varying the Types of Problems*) Modify the previous exercise to allow the user to pick a type of arithmetic problem to study—1 means addition problems only, 2 means subtraction problems only, 3 means multiplication problems only, 4 means division problems only (avoid dividing by 0) and 5 means a random mixture of all these types.

4.18 (*Functional-Style Programming: Internal vs. External Iteration*) Why is internal iteration preferable to external iteration in functional-style programming?

4.19 (*Functional-Style Programming: What vs. How*) Why is programming that emphasizes "what" preferable to programming that emphasizes "how"? What is it that makes "what" programming feasible?

4.20 (*Intro to Data Science: Population Variance vs. Sample Variance*) We mentioned in the Intro to Data Science section that there's a slight difference between the way the `statistics` module's functions calculate the population variance and the sample variance. The same is true for the population standard deviation and the sample standard deviation. Research the reason for these differences.

5

Sequences: Lists and Tuples



Objectives

In this chapter, you'll:

- Create and initialize lists and tuples.
- Refer to elements of lists, tuples and strings.
- Sort and search lists, and search tuples.
- Pass lists and tuples to functions and methods.
- Use list methods to perform common manipulations, such as searching for items, sorting a list, inserting items and removing items.
- Use additional Python functional-style programming capabilities, including lambdas and the functional-style programming operations filter, map and reduce.
- Use functional-style list comprehensions to create lists quickly and easily, and use generator expressions to generate values on demand.
- Use two-dimensional lists.
- Enhance your analysis and presentation skills with the Seaborn and Matplotlib visualization libraries.

Outline

-
- | | |
|---|---|
| 5.1 Introduction
5.2 Lists
5.3 Tuples
5.4 Unpacking Sequences
5.5 Sequence Slicing
5.6 <code>del</code> Statement
5.7 Passing Lists to Functions
5.8 Sorting Lists
5.9 Searching Sequences
5.10 Other List Methods
5.11 Simulating Stacks with Lists | 5.12 List Comprehensions
5.13 Generator Expressions
5.14 Filter, Map and Reduce
5.15 Other Sequence Processing Functions
5.16 Two-Dimensional Lists
5.17 Intro to Data Science: Simulation and Static Visualizations <ul style="list-style-type: none"> 5.17.1 Sample Graphs for 600, 60,000 and 6,000,000 Die Rolls 5.17.2 Visualizing Die-Roll Frequencies and Percentages 5.18 Wrap-Up Exercises |
|---|---|
-

5.1 Introduction

In the last two chapters, we briefly introduced the list and tuple sequence types for representing ordered collections of items. **Collections** are prepackaged data structures consisting of related data items. Examples of collections include your favorite songs on your smartphone, your contacts list, a library’s books, your cards in a card game, your favorite sports team’s players, the stocks in an investment portfolio, patients in a cancer study and a shopping list. Python’s built-in collections enable you to store and access data conveniently and efficiently. In this chapter, we discuss lists and tuples in more detail.

We’ll demonstrate common list and tuple manipulations. You’ll see that lists (which are modifiable) and tuples (which are not) have many common capabilities. Each can hold items of the same or different types. Lists can **dynamically resize** as necessary, growing and shrinking at execution time. We discuss one-dimensional and two-dimensional lists.

In the preceding chapter, we demonstrated random-number generation and simulated rolling a six-sided die. We conclude this chapter with our next Intro to Data Science section, which uses the visualization libraries Seaborn and Matplotlib to interactively develop static bar charts containing the die frequencies. In the next chapter’s Intro to Data Science section, we’ll present an animated visualization in which the bar chart changes *dynamically* as the number of die rolls increases—you’ll see the law of large numbers “in action.”

5.2 Lists

Here, we discuss lists in more detail and explain how to refer to particular list **elements**. Many of the capabilities shown in this section apply to all sequence types.

Creating a List

Lists typically store **homogeneous data**, that is, values of the *same* data type. Consider the list `c`, which contains five integer elements:

```
In [1]: c = [-45, 6, 0, 72, 1543]
```

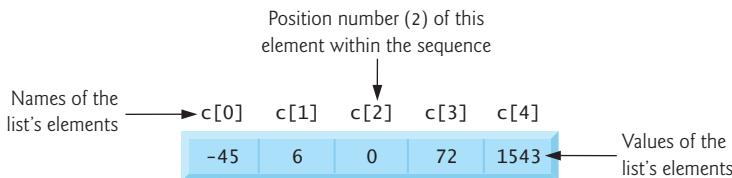
```
In [2]: c
Out[2]: [-45, 6, 0, 72, 1543]
```

They also may store **heterogeneous data**, that is, data of many different types. For example, the following list contains a student's first name (a string), last name (a string), grade point average (a `float`) and graduation year (an `int`):

```
[ 'Mary', 'Smith', 3.57, 2022]
```

Accessing Elements of a List

You reference a list element by writing the list's name followed by the element's **index** (that is, its **position number**) enclosed in square brackets (`[]`), known as the **subscription operator**). The following diagram shows the list `c` labeled with its element names:



The first element in a list has the index 0. So, in the five-element list `c`, the first element is named `c[0]` and the last is `c[4]`:

```
In [3]: c[0]
Out[3]: -45
```

```
In [4]: c[4]
Out[4]: 1543
```

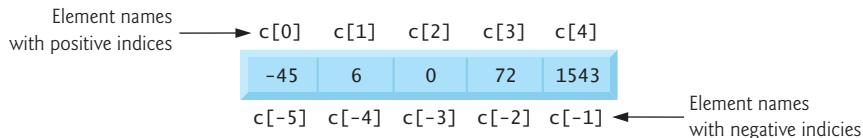
Determining a List's Length

To get a list's length, use the built-in `len` function:

```
In [5]: len(c)
Out[5]: 5
```

Accessing Elements from the End of the List with Negative Indices

Lists also can be accessed from the end by using **negative indices**:



So, list `c`'s last element (`c[4]`), can be accessed with `c[-1]` and its first element with `c[-5]`:

```
In [6]: c[-1]
Out[6]: 1543
```

```
In [7]: c[-5]
Out[7]: -45
```

Indices Must Be Integers or Integer Expressions

An index must be an integer or integer expression (or a *slice*, as we'll soon see):

```
In [8]: a = 1
In [9]: b = 2
In [10]: c[a + b]
Out[10]: 72
```

Using a non-integer index value causes a `TypeError`.

Lists Are Mutable

Lists are mutable—their elements can be modified:

```
In [11]: c[4] = 17
In [12]: c
Out[12]: [-45, 6, 0, 72, 17]
```

You'll soon see that you also can insert and delete elements, changing the list's length.

Some Sequences Are Immutable

Python's string and tuple sequences are immutable—they cannot be modified. You can get the individual characters in a string, but attempting to assign a new value to one of the characters causes a `TypeError`:

```
In [13]: s = 'hello'
In [14]: s[0]
Out[14]: 'h'
In [15]: s[0] = 'H'
-----
TypeError                                         Traceback (most recent call last)
<ipython-input-15-812ef2514689> in <module>()
----> 1 s[0] = 'H'

TypeError: 'str' object does not support item assignment
```

Attempting to Access a Nonexistent Element

Using an out-of-range list, tuple or string index causes an `IndexError`:

```
In [16]: c[100]
-----
IndexError                                         Traceback (most recent call last)
<ipython-input-19-9a31ea1e1a13> in <module>()
----> 1 c[100]

IndexError: list index out of range
```

Using List Elements in Expressions

List elements may be used as variables in expressions:

```
In [17]: c[0] + c[1] + c[2]
Out[17]: -39
```

Appending to a List with `+=`

Let's start with an *empty* list `[]`, then use a `for` statement and `+=` to append the values 1 through 5 to the list—the list grows dynamically to accommodate each item:

```
In [18]: a_list = []

In [19]: for number in range(1, 6):
...:     a_list += [number]
...:

In [20]: a_list
Out[20]: [1, 2, 3, 4, 5]
```

When the left operand of `+=` is a list, the right operand must be an *iterable*; otherwise, a `TypeError` occurs. In snippet [19]'s suite, the square brackets around `number` create a one-element list, which we append to `a_list`. If the right operand contains multiple elements, `+=` appends them all. The following appends the characters of 'Python' to the list `letters`:

```
In [21]: letters = []

In [22]: letters += 'Python'

In [23]: letters
Out[23]: ['P', 'y', 't', 'h', 'o', 'n']
```

If the right operand of `+=` is a tuple, its elements also are appended to the list. Later in the chapter, we'll use the list method `append` to add items to a list.

Concatenating Lists with +

You can **concatenate** two lists, two tuples or two strings using the `+` operator. The result is a *new* sequence of the same type containing the left operand's elements followed by the right operand's elements. The original sequences are unchanged:

```
In [24]: list1 = [10, 20, 30]

In [25]: list2 = [40, 50]

In [26]: concatenated_list = list1 + list2

In [27]: concatenated_list
Out[27]: [10, 20, 30, 40, 50]
```

A `TypeError` occurs if the `+` operator's operands are difference sequence types—for example, concatenating a list and a tuple is an error.

Using for and range to Access List Indices and Values

List elements also can be accessed via their indices and the subscription operator (`[]`):

```
In [28]: for i in range(len(concatenated_list)):
...:     print(f'{i}: {concatenated_list[i]}')
...:

0: 10
1: 20
2: 30
3: 40
4: 50
```

The function call `range(len(concatenated_list))` produces a sequence of integers representing `concatenated_list`'s indices (in this case, 0 through 4). When looping in this manner, you must ensure that indices remain in range. Soon, we'll show a safer way to access element indices and values using built-in function `enumerate`.

Comparison Operators

You can compare entire lists element-by-element using comparison operators:

```
In [29]: a = [1, 2, 3]
In [30]: b = [1, 2, 3]
In [31]: c = [1, 2, 3, 4]
In [32]: a == b # True: corresponding elements in both are equal
Out[32]: True
In [33]: a == c # False: a and c have different elements and lengths
Out[33]: False
In [34]: a < c # True: a has fewer elements than c
Out[34]: True
In [35]: c >= b # True: elements 0-2 are equal but c has more elements
Out[35]: True
```



Self Check

1 (*Fill-In*) Python's string and tuple sequences are _____—they cannot be modified.
Answer: immutable.

2 (*True/False*) The + operator's sequence operands may be of any sequence type.
Answer: False. The + operator's operand sequences must have the *same* type; otherwise, a TypeError occurs.

3 (*IPython Session*) Create a function cube_list that cubes each element of a list. Call the function with the list numbers containing 1 through 10. Show numbers after the call.
Answer:

```
In [1]: def cube_list(values):
...:     for i in range(len(values)):
...:         values[i] **= 3
...:
In [2]: numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
In [3]: cube_list(numbers)
In [4]: numbers
Out[4]: [1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

4 (*IPython Session*) Use an empty list named characters and a += augmented assignment statement to convert the string 'Birthday' into a list of its characters.

Answer:

```
In [5]: characters = []
In [6]: characters += 'Birthday'
In [7]: characters
Out[7]: ['B', 'i', 'r', 't', 'h', 'd', 'a', 'y']
```

5.3 Tuples

As discussed in the preceding chapter, tuples are immutable and typically store heterogeneous data, but the data can be homogeneous. A tuple's length is its number of elements and cannot change during program execution.

Creating Tuples

To create an empty tuple, use empty parentheses:

```
In [1]: student_tuple = ()
```

```
In [2]: student_tuple
```

```
Out[2]: ()
```

```
In [3]: len(student_tuple)
```

```
Out[3]: 0
```

Recall that you can pack a tuple by separating its values with commas:

```
In [4]: student_tuple = 'John', 'Green', 3.3
```

```
In [5]: student_tuple
```

```
Out[5]: ('John', 'Green', 3.3)
```

```
In [6]: len(student_tuple)
```

```
Out[6]: 3
```

When you output a tuple, Python always displays its contents in parentheses. You may surround a tuple's comma-separated list of values with optional parentheses:

```
In [7]: another_student_tuple = ('Mary', 'Red', 3.3)
```

```
In [8]: another_student_tuple
```

```
Out[8]: ('Mary', 'Red', 3.3)
```

The following code creates a one-element tuple:

```
In [9]: a_singleton_tuple = ('red',) # note the comma
```

```
In [10]: a_singleton_tuple
```

```
Out[10]: ('red',)
```

The comma (,) that follows the string 'red' identifies `a_singleton_tuple` as a tuple—the parentheses are optional. If the comma were omitted, the parentheses would be redundant, and `a_singleton_tuple` would simply refer to the string 'red' rather than a tuple.

Accessing Tuple Elements

A tuple's elements, though related, are often of multiple types. Usually, you do not iterate over them. Rather, you access each individually. Like list indices, tuple indices start at 0. The following code creates `time_tuple` representing an hour, minute and second, displays the tuple, then uses its elements to calculate the number of seconds since midnight—note that we perform a *different* operation with each value in the tuple:

```
In [11]: time_tuple = (9, 16, 1)
```

```
In [12]: time_tuple
```

```
Out[12]: (9, 16, 1)
```

```
In [13]: time_tuple[0] * 3600 + time_tuple[1] * 60 + time_tuple[2]
Out[13]: 33361
```

Assigning a value to a tuple element causes a `TypeError`.

Adding Items to a String or Tuple

As with lists, the `+=` augmented assignment statement can be used with strings and tuples, even though they're *immutable*. In the following code, after the two assignments, `tuple1` and `tuple2` refer to the *same* tuple object:

```
In [14]: tuple1 = (10, 20, 30)
```

```
In [15]: tuple2 = tuple1
```

```
In [16]: tuple2
```

```
Out[16]: (10, 20, 30)
```

Concatenating the tuple `(40, 50)` to `tuple1` creates a *new* tuple, then assigns a reference to it to the variable `tuple1`—`tuple2` still refers to the original tuple:

```
In [17]: tuple1 += (40, 50)
```

```
In [18]: tuple1
```

```
Out[18]: (10, 20, 30, 40, 50)
```

```
In [19]: tuple2
```

```
Out[19]: (10, 20, 30)
```

For a string or tuple, the item to the right of `+=` must be a string or tuple, respectively—mixing types causes a `TypeError`.

Appending Tuples to Lists

You can use `+=` to append a tuple to a list:

```
In [20]: numbers = [1, 2, 3, 4, 5]
```

```
In [21]: numbers += (6, 7)
```

```
In [22]: numbers
```

```
Out[22]: [1, 2, 3, 4, 5, 6, 7]
```

Tuples May Contain Mutable Objects

Let's create a `student_tuple` with a first name, last name and list of grades:

```
In [23]: student_tuple = ('Amanda', 'Blue', [98, 75, 87])
```

Even though the tuple is immutable, its list element is mutable:

```
In [24]: student_tuple[2][1] = 85
```

```
In [25]: student_tuple
```

```
Out[25]: ('Amanda', 'Blue', [98, 85, 87])
```

In the *double-subscripted* name `student_tuple[2][1]`, Python views `student_tuple[2]` as the element of the tuple containing the list `[98, 75, 87]`, then uses `[1]` to access the list element containing 75. The assignment in snippet [24] replaces that grade with 85.



Self Check

1 (*True/False*) A `+=` augmented assignment statement may not be used with strings and tuples, because they're immutable.

Answer: False. A `+=` augmented assignment statement also may be used with strings and tuples, even though they're immutable. The result is a *new* string or tuple, respectively.

2 (*True/False*) Tuples can contain only immutable objects.

Answer: False. Even though a tuple is immutable, its elements can be mutable objects, such as lists.

3 (*IPython Session*) Create a single-element tuple containing 123.45, then display it.

Answer:

```
In [1]: single = (123.45,)
```

```
In [2]: single
```

```
Out[2]: (123.45,)
```

4 (*IPython Session*) Show what happens when you attempt to concatenate sequences of different types—the list `[1, 2, 3]` and the tuple `(4, 5, 6)`—using the `+` operator.

Answer:

```
In [3]: [1, 2, 3] + (4, 5, 6)
```

```
-----
```

```
TypeError
```

```
Traceback (most recent call last)
```

```
<ipython-input-3-1ac3d3041bfa> in <module>()
```

```
----> 1 [1, 2, 3] + (4, 5, 6)
```

```
TypeError: can only concatenate list (not "tuple") to list
```

5.4 Unpacking Sequences

The previous chapter introduced tuple unpacking. You can unpack any sequence's elements by assigning the sequence to a comma-separated list of variables. A `ValueError` occurs if the number of variables to the left of the assignment symbol is not identical to the number of elements in the sequence on the right:

```
In [1]: student_tuple = ('Amanda', [98, 85, 87])
```

```
In [2]: first_name, grades = student_tuple
```

```
In [3]: first_name
```

```
Out[3]: 'Amanda'
```

```
In [4]: grades
```

```
Out[4]: [98, 85, 87]
```

The following code unpacks a string, a list and a sequence produced by `range`:

```
In [5]: first, second = 'hi'
```

```
In [6]: print(f'{first} {second}')
```

```
h i
```

```
In [7]: number1, number2, number3 = [2, 3, 5]
In [8]: print(f'{number1} {number2} {number3}')
2 3 5
In [9]: number1, number2, number3 = range(10, 40, 10)
In [10]: print(f'{number1} {number2} {number3}')
10 20 30
```

Swapping Values Via Packing and Unpacking

You can swap two variables' values using sequence packing and unpacking:

```
In [11]: number1 = 99
In [12]: number2 = 22
In [13]: number1, number2 = (number2, number1)
In [14]: print(f'number1 = {number1}; number2 = {number2}')
number1 = 22; number2 = 99
```

Accessing Indices and Values Safely with Built-in Function enumerate

Earlier, we called `range` to produce a sequence of index values, then accessed list elements in a `for` loop using the index values and the subscription operator (`[]`). This is error-prone because you could pass the wrong arguments to `range`. If any value produced by `range` is an out-of-bounds index, using it as an index causes an `IndexError`.

The preferred mechanism for accessing an element's index *and* value is the built-in function `enumerate`. This function receives an iterable and creates an iterator that, for each element, returns a tuple containing the element's index and value. The following code uses the built-in function `list` to create a list containing `enumerate`'s results:

```
In [15]: colors = ['red', 'orange', 'yellow']
In [16]: list(enumerate(colors))
Out[16]: [(0, 'red'), (1, 'orange'), (2, 'yellow')]
```

Similarly the built-in function `tuple` creates a tuple from a sequence:

```
In [17]: tuple(enumerate(colors))
Out[17]: ((0, 'red'), (1, 'orange'), (2, 'yellow'))
```

The following `for` loop unpacks each tuple returned by `enumerate` into the variables `index` and `value` and displays them:

```
In [18]: for index, value in enumerate(colors):
    ...:     print(f'{index}: {value}')
    ...:
0: red
1: orange
2: yellow
```

Creating a Primitive Bar Chart

The script in Fig. 5.1 creates a primitive **bar chart** where each bar's length is made of asterisks (*) and is proportional to the list's corresponding element value. We use the function

`enumerate` to access the list's indices and values safely. To run this example, change to this chapter's `ch05` examples folder, then enter:

```
ipython fig05_01.py
```

or, if you're in IPython already, use the command:

```
run fig05_01.py
```

```
1 # fig05_01.py
2 """Displaying a bar chart"""
3 numbers = [19, 3, 15, 7, 11]
4
5 print('\nCreating a bar chart from numbers:')
6 print(f'Index{"Value":>8} Bar')
7
8 for index, value in enumerate(numbers):
9     print(f'{index:>5}{value:>8} {"*"* value}')
```

```
Creating a bar chart from numbers:
Index  Value  Bar
  0      19  *****
  1       3  ***
  2      15  *****
  3       7  ****
  4      11  *****
```

Fig. 5.1 | Displaying a bar chart.

The `for` statement uses `enumerate` to get each element's index and value, then displays a formatted line containing the index, the element value and the corresponding bar of asterisks. The expression

```
"*" * value
```

creates a string consisting of `value` asterisks. When used with a sequence, the multiplication operator (`*`) repeats the sequence—in this case, the string `"*"`—`value` times. Later in this chapter, we'll use the open-source Seaborn and Matplotlib libraries to display a publication-quality bar chart visualization.



Self Check

- 1 (*Fill-In*) A sequence's elements can be _____ by assigning the sequence to a comma-separated list of variables.

Answer: unpacked.

- 2 (*True/False*) The following expression causes an error:

```
'-' * 10
```

Answer: False: In this context, the multiplication operator (`*`) repeats the string `(' - ')` 10 times.

3 (IPython Session) Create a tuple `high_low` representing a day of the week (a string) and its high and low temperatures (integers), display its string representation, then perform the following tasks in an interactive IPython session:

- Use the `[]` operator to access and display the `high_low` tuple's elements.
- Unpack the `high_low` tuple into the variables `day` and `high`. What happens and why?

Answer: For Part (b) an error occurs because you must unpack *all* the elements of a sequence.

```
In [1]: high_low = ('Monday', 87, 65)

In [2]: high_low
Out[2]: ('Monday', 87, 65)

In [3]: print(f'{high_low[0]}: High={high_low[1]}, Low={high_low[2]}')
Monday: High=87, Low=65

In [4]: day, high = high_low
-----
ValueError                                     Traceback (most recent call last)
<ipython-input-3-0c3ad5c97284> in <module>()
----> 1 day, high = high_low

ValueError: too many values to unpack (expected 2)
```

4 (IPython Session) Create the list `names` containing three name strings. Use a `for` loop and the `enumerate` function to iterate through the elements and display each element's index and value.

Answer:

```
In [4]: names = ['Amanda', 'Sam', 'David']

In [5]: for i, name in enumerate(names):
...:     print(f'{i}: {name}')
...:
0: Amanda
1: Sam
2: David
```

5.5 Sequence Slicing

You can **slice** sequences to create new sequences of the same type containing *subsets* of the original elements. Slice operations can modify mutable sequences—those that do *not* modify a sequence work identically for lists, tuples and strings.

Specifying a Slice with Starting and Ending Indices

Let's create a slice consisting of the elements at indices 2 through 5 of a list:

```
In [1]: numbers = [2, 3, 5, 7, 11, 13, 17, 19]

In [2]: numbers[2:6]
Out[2]: [5, 7, 11, 13]
```

The slice *copies* elements from the *starting index* to the left of the colon (2) up to, but not including, the *ending index* to the right of the colon (6). The original list is not modified.

Specifying a Slice with Only an Ending Index

If you omit the starting index, 0 is assumed. So, the slice `numbers[:6]` is equivalent to the slice `numbers[0:6]`:

```
In [3]: numbers[:6]
Out[3]: [2, 3, 5, 7, 11, 13]

In [4]: numbers[0:6]
Out[4]: [2, 3, 5, 7, 11, 13]
```

Specifying a Slice with Only a Starting Index

If you omit the ending index, Python assumes the sequence's length (8 here), so snippet [5]'s slice contains the elements of `numbers` at indices 6 and 7:

```
In [5]: numbers[6:]
Out[5]: [17, 19]

In [6]: numbers[6:len(numbers)]
Out[6]: [17, 19]
```

Specifying a Slice with No Indices

Omitting both the start and end indices copies the entire sequence:

```
In [7]: numbers[:]
Out[7]: [2, 3, 5, 7, 11, 13, 17, 19]
```

Though slices create new objects, slices make *shallow* copies of the elements—that is, they copy the elements' references but not the objects they point to. So, in the snippet above, the new list's elements refer to the *same objects* as the original list's elements, rather than to separate copies. In the “Array-Oriented Programming with NumPy” chapter, we'll explain *deep* copying, which actually copies the referenced objects themselves, and we'll point out when deep copying is preferred.

Slicing with Steps

The following code uses a *step* of 2 to create a slice with every other element of `numbers`:

```
In [8]: numbers[::2]
Out[8]: [2, 5, 11, 17]
```

We omitted the start and end indices, so 0 and `len(numbers)` are assumed, respectively.

Slicing with Negative Indices and Steps

You can use a negative step to select slices in *reverse* order. The following code concisely creates a new list in reverse order:

```
In [9]: numbers[::-1]
Out[9]: [19, 17, 13, 11, 7, 5, 3, 2]
```

This is equivalent to:

```
In [10]: numbers[-1:-9:-1]
Out[10]: [19, 17, 13, 11, 7, 5, 3, 2]
```

Modifying Lists Via Slices

You can modify a list by assigning to a slice of it—the rest of the list is unchanged. The following code replaces `numbers`' first three elements, leaving the rest unchanged:

```
In [11]: numbers[0:3] = ['two', 'three', 'five']
```

```
In [12]: numbers
```

```
Out[12]: ['two', 'three', 'five', 7, 11, 13, 17, 19]
```

The following deletes only the first three elements of `numbers` by assigning an *empty* list to the three-element slice:

```
In [13]: numbers[0:3] = []
```

```
In [14]: numbers
```

```
Out[14]: [7, 11, 13, 17, 19]
```

The following assigns a list's elements to a slice of every other element of `numbers`:

```
In [15]: numbers = [2, 3, 5, 7, 11, 13, 17, 19]
```

```
In [16]: numbers[::-2] = [100, 100, 100, 100]
```

```
In [17]: numbers
```

```
Out[17]: [100, 3, 100, 7, 100, 13, 100, 19]
```

```
In [18]: id(numbers)
```

```
Out[18]: 4434456648
```

Let's delete all the elements in `numbers`, leaving the *existing* list empty:

```
In [19]: numbers[:] = []
```

```
In [20]: numbers
```

```
Out[20]: []
```

```
In [21]: id(numbers)
```

```
Out[21]: 4434456648
```

Deleting `numbers`' contents (snippet [19]) is different from assigning `numbers` a *new* empty list `[]` (snippet [22]). To prove this, we display `numbers`' identity after each operation. The identities are different, so they represent separate objects in memory:

```
In [22]: numbers = []
```

```
In [23]: numbers
```

```
Out[23]: []
```

```
In [24]: id(numbers)
```

```
Out[24]: 4406030920
```

When you assign a new object to a variable (as in snippet [21]), the original object will be garbage collected if no other variables refer to it.



Self Check

- (True/False)* Slice operations that modify a sequence work identically for lists, tuples and strings.

Answer: False. Slice operations that *do not* modify a sequence work identically for lists, tuples and strings.

- (Fill-In)* Assume you have a list called `names`. The slice expression _____ creates a new list with the elements of `names` in reverse order.

Answer: `names[::-1]`

3 (IPython Session) Create a list called `numbers` containing the values from 1 through 15, then use *slices* to perform the following operations consecutively:

- Select `number`'s even integers.
- Replace the elements at indices 5 through 9 with 0s, then show the resulting list.
- Keep only the first five elements, then show the resulting list.
- Delete all the remaining elements by assigning to a slice. Show the resulting list.

Answer:

```
In [1]: numbers = list(range(1, 16))

In [2]: numbers
Out[2]: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]

In [3]: numbers[1:len(numbers):2]
Out[3]: [2, 4, 6, 8, 10, 12, 14]

In [4]: numbers[5:10] = [0] * len(numbers[5:10])

In [5]: numbers
Out[5]: [1, 2, 3, 4, 5, 0, 0, 0, 0, 0, 11, 12, 13, 14, 15]

In [6]: numbers[5:] = []

In [7]: numbers
Out[7]: [1, 2, 3, 4, 5]

In [8]: numbers[:] = []

In [9]: numbers
Out[9]: []
```

Recall that multiplying a sequence repeats that sequence the specified number of times.

5.6 del Statement

The **del statement** also can be used to remove elements from a list and to delete variables from the interactive session. You can remove the element at any valid index or the element(s) from any valid slice.

Deleting the Element at a Specific List Index

Let's create a list, then use `del` to remove its last element:

```
In [1]: numbers = list(range(0, 10))

In [2]: numbers
Out[2]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [3]: del numbers[-1]

In [4]: numbers
Out[4]: [0, 1, 2, 3, 4, 5, 6, 7, 8]
```

Deleting a Slice from a List

The following deletes the list's first two elements:

```
In [5]: del numbers[0:2]

In [6]: numbers
Out[6]: [2, 3, 4, 5, 6, 7, 8]
```

The following uses a step in the slice to delete every other element from the entire list:

```
In [7]: del numbers[::-2]
```

```
In [8]: numbers
```

```
Out[8]: [3, 5, 7]
```

Deleting a Slice Representing the Entire List

The following code deletes all of the list's elements:

```
In [9]: del numbers[:]
```

```
In [10]: numbers
```

```
Out[10]: []
```

Deleting a Variable from the Current Session

The `del` statement can delete any variable. Let's delete `numbers` from the interactive session, then attempt to display the variable's value, causing a `NameError`:

```
In [11]: del numbers
```

```
In [12]: numbers
```

```
NameError
```

```
Traceback (most recent call last)
```

```
<ipython-input-12-426f8401232b> in <module>()
```

```
----> 1 numbers
```

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



Self Check

- 1 (Fill-In)** Given a list `numbers` containing 1 through 10, `del numbers[-2]` removes the value _____ from the list.

Answer: 9.

- 2 (IPython Session)** Create a list called `numbers` containing the values from 1 through 15, then use the `del` statement to perform the following operations consecutively:

- Delete a slice containing the first four elements, then show the resulting list.
- Starting with the first element, use a slice to delete every other element of the list, then show the resulting list.

Answer:

```
In [1]: numbers = list(range(1, 16))
```

```
In [2]: numbers
```

```
Out[2]: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
```

```
In [3]: del numbers[0:4]
```

```
In [4]: numbers
```

```
Out[4]: [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
```

```
In [5]: del numbers[::-2]
```

```
In [6]: numbers
```

```
Out[6]: [6, 8, 10, 12, 14]
```

5.7 Passing Lists to Functions

In the last chapter, we mentioned that all objects are passed by reference and demonstrated passing an immutable object as a function argument. Here, we discuss references further by examining what happens when a program passes a mutable list object to a function.

Passing an Entire List to a Function

Consider the function `modify_elements`, which receives a reference to a list and multiplies each of the list's element values by 2:

```
In [1]: def modify_elements(items):
...:     """Multiplies all element values in items by 2."""
...:     for i in range(len(items)):
...:         items[i] *= 2
...:

In [2]: numbers = [10, 3, 7, 1, 9]

In [3]: modify_elements(numbers)

In [4]: numbers
Out[4]: [20, 6, 14, 2, 18]
```

Function `modify_elements`'s `items` parameter receives a reference to the *original* list, so the statement in the loop's suite modifies each element in the original list object.

Passing a Tuple to a Function

When you pass a tuple to a function, attempting to modify the tuple's immutable elements results in a `TypeError`:

```
In [5]: numbers_tuple = (10, 20, 30)

In [6]: numbers_tuple
Out[6]: (10, 20, 30)

In [7]: modify_elements(numbers_tuple)
-----
TypeError                                         Traceback (most recent call last)
<ipython-input-27-9339741cd595> in <module>()
----> 1 modify_elements(numbers_tuple)

<ipython-input-25-27acb8f8f44c> in modify_elements(items)
      2     """Multiplies all element values in items by 2."""
      3     for i in range(len(items)):
----> 4         items[i] *= 2
      5
      6

TypeError: 'tuple' object does not support item assignment
```

Recall that tuples may contain mutable objects, such as lists. Those objects still can be modified when a tuple is passed to a function.

A Note Regarding Tracebacks

The previous traceback shows the *two* snippets that led to the `TypeError`. The first is snippet [7]'s function call. The second is snippet [1]'s function definition. Line numbers pre-

cede each snippet’s code. We’ve demonstrated mostly single-line snippets. When an exception occurs in such a snippet, it’s always preceded by ----> 1, indicating that line 1 (the snippet’s only line) caused the exception. Multiline snippets like the definition of `modify_elements` show consecutive line numbers starting at 1. The notation ----> 4 above indicates that the exception occurred in line 4 of `modify_elements`. No matter how long the traceback is, the last line of code with ----> caused the exception.



Self Check

1 (*True/False*) You cannot modify a list’s contents when you pass it to a function.

Answer: False. When you pass a list (a mutable object) to a function, the function receives a reference to the original list object and can use that reference to modify the original list’s contents.

2 (*True/False*) Tuples can contain lists and other mutable objects. Those mutable objects can be modified when a tuple is passed to a function.

Answer: True.

5.8 Sorting Lists

A common computing task called **sorting** enables you to arrange data either in ascending or descending order. Sorting is an intriguing problem that has attracted intense computer-science research efforts. It’s studied in detail in data-structures and algorithms courses. We discuss sorting in more detail in the “Computer Science Thinking: Recursion, Searching, Sorting and Big O” chapter.

Sorting a List in Ascending Order

List method `sort` modifies a list to arrange its elements in ascending order:

```
In [1]: numbers = [10, 3, 7, 1, 9, 4, 2, 8, 5, 6]
In [2]: numbers.sort()
In [3]: numbers
Out[3]: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Sorting a List in Descending Order

To sort a list in descending order, call list method `sort` with the optional keyword argument `reverse` set to True (`False` is the default):

```
In [4]: numbers.sort(reverse=True)
In [5]: numbers
Out[5]: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Built-In Function `sorted`

Built-in function `sorted` returns a new list containing the sorted elements of its argument *sequence*—the original sequence is *unmodified*. The following code demonstrates function `sorted` for a list, a string and a tuple:

```
In [6]: numbers = [10, 3, 7, 1, 9, 4, 2, 8, 5, 6]
In [7]: ascending_numbers = sorted(numbers)
```

```
In [8]: ascending_numbers
Out[8]: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

In [9]: numbers
Out[9]: [10, 3, 7, 1, 9, 4, 2, 8, 5, 6]

In [10]: letters = 'fadgchjebi'

In [11]: ascending_letters = sorted(letters)

In [12]: ascending_letters
Out[12]: ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']

In [13]: letters
Out[13]: 'fadgchjebi'

In [14]: colors = ('red', 'orange', 'yellow', 'green', 'blue')

In [15]: ascending_colors = sorted(colors)

In [16]: ascending_colors
Out[16]: ['blue', 'green', 'orange', 'red', 'yellow']

In [17]: colors
Out[17]: ('red', 'orange', 'yellow', 'green', 'blue')
```

Use the optional keyword argument `reverse` with the value `True` to sort the elements in descending order.



Self Check

- 1** (*Fill-In*) To sort a list in descending order, call list method `sort` with the optional keyword argument _____ set to `True`.

Answer: `reverse`.

- 2** (*True/False*) All sequences provide a `sort` method.

Answer: False. Immutable sequences like tuples and strings do not provide a `sort` method. However, you can sort *any* sequence *without modifying it* by using built-in function `sorted`, which returns a *new* list containing the sorted elements of its argument sequence.

- 3** (*IPython Session*) Create a `foods` list containing '`Cookies`', '`pizza`', '`Grapes`', '`apples`', '`steak`' and '`Bacon`'. Use list method `sort` to sort the list in ascending order. Are the strings in alphabetical order?

Answer:

```
In [1]: foods = ['Cookies', 'pizza', 'Grapes',
...:             'apples', 'steak', 'Bacon']
...:

In [2]: foods.sort()

In [3]: foods
Out[3]: ['Bacon', 'Cookies', 'Grapes', 'apples', 'pizza', 'steak']
```

They're probably not in what you'd consider alphabetical order, but they are in order as defined by the underlying character set—known as **lexicographical order**. As you'll see later in the chapter, strings are compared by their character's numerical values, not their letters, and the values of uppercase letters are *lower* than the values of lowercase letters.

5.9 Searching Sequences

Often, you'll want to determine whether a sequence (such as a list, tuple or string) contains a value that matches a particular **key** value. **Searching** is the process of locating a key.

List Method `index`

List method `index` takes as an argument a search key—the value to locate in the list—then searches through the list from index 0 and returns the index of the *first* element that matches the search key:

```
In [1]: numbers = [3, 7, 1, 4, 2, 8, 5, 6]
```

```
In [2]: numbers.index(5)
Out[2]: 6
```

A `ValueError` occurs if the value you're searching for is not in the list.

Specifying the Starting Index of a Search

Using method `index`'s optional arguments, you can search a subset of a list's elements. You can use `*=` to *multiply a sequence*—that is, append a sequence to itself multiple times. After the following snippet, `numbers` contains two copies of the original list's contents:

```
In [3]: numbers *= 2
```

```
In [4]: numbers
Out[4]: [3, 7, 1, 4, 2, 8, 5, 6, 3, 7, 1, 4, 2, 8, 5, 6]
```

The following code searches the updated list for the value 5 starting from index 7 and continuing through the end of the list:

```
In [5]: numbers.index(5, 7)
Out[5]: 14
```

Specifying the Starting and Ending Indices of a Search

Specifying the starting and ending indices causes `index` to search from the starting index up to but not including the ending index location. The call to `index` in snippet [5]:

```
numbers.index(5, 7)
```

assumes the length of `numbers` as its optional third argument and is equivalent to:

```
numbers.index(5, 7, len(numbers))
```

The following looks for the value 7 in the range of elements with indices 0 through 3:

```
In [6]: numbers.index(7, 0, 4)
Out[6]: 1
```

Operators `in` and `not in`

Operator `in` tests whether its right operand's iterable contains the left operand's value:

```
In [7]: 1000 in numbers
Out[7]: False
```

```
In [8]: 5 in numbers
Out[8]: True
```

Similarly, operator `not in` tests whether its right operand's iterable does *not* contain the left operand's value:

```
In [9]: 1000 not in numbers
Out[9]: True
```

```
In [10]: 5 not in numbers
Out[10]: False
```

Using Operator `in` to Prevent a `ValueError`

You can use the operator `in` to ensure that calls to method `index` do not result in `ValueError`s for search keys that are not in the corresponding sequence:

```
In [11]: key = 1000
```

```
In [12]: if key in numbers:
...:     print(f'found {key} at index {numbers.index(search_key)}')
...: else:
...:     print(f'{key} not found')
...:
1000 not found
```

Built-In Functions `any` and `all`

Sometimes you simply need to know whether *any* item in an iterable is `True` or whether *all* the items are `True`. The built-in function `any` returns `True` if any item in its iterable argument is `True`. The built-in function `all` returns `True` if all items in its iterable argument are `True`. Recall that nonzero values are `True` and `0` is `False`. Non-empty iterable objects also evaluate to `True`, whereas any empty iterable evaluates to `False`. Functions `any` and `all` are additional examples of internal iteration in functional-style programming.



Self Check

1 (Fill-In) The _____ operator can be used to extend a list with copies of itself.

Answer: `*=`.

2 (Fill-In) Operators _____ and _____ determine whether a sequence contains or does not contain a value, respectively.

Answer: `in`, `not in`.

3 (IPython Session) Create a five-element list containing 67, 12, 46, 43 and 13, then use list method `index` to search for a 43 and 44. Ensure that no `ValueError` occurs when searching for 44.

Answer:

```
In [1]: numbers = [67, 12, 46, 43, 13]
In [2]: numbers.index(43)
Out[2]: 3
In [3]: if 44 in numbers:
...:     print(f'Found 44 at index: {numbers.index(44)}')
...: else:
...:     print('44 not found')
...:
44 not found
```

5.10 Other List Methods

Lists also have methods that add and remove elements. Consider the list `color_names`:

```
In [1]: color_names = ['orange', 'yellow', 'green']
```

Inserting an Element at a Specific List Index

Method `insert` adds a new item at a specified index. The following inserts 'red' at index 0:

```
In [2]: color_names.insert(0, 'red')
```

```
In [3]: color_names
```

```
Out[3]: ['red', 'orange', 'yellow', 'green']
```

Adding an Element to the End of a List

You can add a new item to the end of a list with method `append`:

```
In [4]: color_names.append('blue')
```

```
In [5]: color_names
```

```
Out[5]: ['red', 'orange', 'yellow', 'green', 'blue']
```

Adding All the Elements of a Sequence to the End of a List

Use list method `extend` to add all the elements of another sequence to the end of a list:

```
In [6]: color_names.extend(['indigo', 'violet'])
```

```
In [7]: color_names
```

```
Out[7]: ['red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet']
```

This is the equivalent of using `+=`. The following code adds all the characters of a string then all the elements of a tuple to a list:

```
In [8]: sample_list = []
```

```
In [9]: s = 'abc'
```

```
In [10]: sample_list.extend(s)
```

```
In [11]: sample_list
```

```
Out[11]: ['a', 'b', 'c']
```

```
In [12]: t = (1, 2, 3)
```

```
In [13]: sample_list.extend(t)
```

```
In [14]: sample_list
```

```
Out[14]: ['a', 'b', 'c', 1, 2, 3]
```

Rather than creating a temporary variable, like `t`, to store a tuple before appending it to a list, you might want to pass a tuple directly to `extend`. In this case, the tuple's parentheses are required, because `extend` expects one iterable argument:

```
In [15]: sample_list.extend((4, 5, 6)) # note the extra parentheses
```

```
In [16]: sample_list
```

```
Out[16]: ['a', 'b', 'c', 1, 2, 3, 4, 5, 6]
```

A `TypeError` occurs if you omit the required parentheses.

Removing the First Occurrence of an Element in a List

Method `remove` deletes the first element with a specified value—a `ValueError` occurs if `remove`'s argument is not in the list:

```
In [17]: color_names.remove('green')

In [18]: color_names
Out[18]: ['red', 'orange', 'yellow', 'blue', 'indigo', 'violet']
```

Emptying a List

To delete all the elements in a list, call method `clear`:

```
In [19]: color_names.clear()

In [20]: color_names
Out[20]: []
```

This is the equivalent of the previously shown slice assignment

```
color_names[:] = []
```

Counting the Number of Occurrences of an Item

List method `count` searches for its argument and returns the number of times it is found:

```
In [21]: responses = [1, 2, 5, 4, 3, 5, 2, 1, 3, 3,
...:                 1, 4, 3, 3, 3, 2, 3, 3, 2, 2]
...:

In [22]: for i in range(1, 6):
...:     print(f'{i} appears {responses.count(i)} times in responses')
...:
1 appears 3 times in responses
2 appears 5 times in responses
3 appears 8 times in responses
4 appears 2 times in responses
5 appears 2 times in responses
```

Reversing a List's Elements

List method `reverse` reverses the contents of a list in place, rather than creating a reversed copy, as we did with a slice previously:

```
In [23]: color_names = ['red', 'orange', 'yellow', 'green', 'blue']

In [24]: color_names.reverse()

In [25]: color_names
Out[25]: ['blue', 'green', 'yellow', 'orange', 'red']
```

Copying a List

List method `copy` returns a *new* list containing a *shallow* copy of the original list:

```
In [26]: copied_list = color_names.copy()

In [27]: copied_list
Out[27]: ['blue', 'green', 'yellow', 'orange', 'red']
```

This is equivalent to the previously demonstrated slice operation:

```
copied_list = color_names[:]
```



Self Check

- 1 (*Fill-In*) To add all the elements of a sequence to the end of a list, use list method _____, which is equivalent to using `+=`.

Answer: `extend`.

- 2 (*Fill-In*) For a list `numbers`, calling method _____ is equivalent to `numbers[:] = []`.
Answer: `clear`.

- 3 (*IPython Session*) Create a list called `rainbow` containing 'green', 'orange' and 'violet'. Perform the following operations consecutively using list methods and show the list's contents after each operation:

- Determine the index of 'violet', then use it to insert 'red' before 'violet'.
- Append 'yellow' to the end of the list.
- Reverse the list's elements.
- Remove the element 'orange'.

Answer:

```
In [1]: rainbow = ['green', 'orange', 'violet']

In [2]: rainbow.insert(rainbow.index('violet'), 'red')

In [3]: rainbow
Out[3]: ['green', 'orange', 'red', 'violet']

In [4]: rainbow.append('yellow')

In [5]: rainbow
Out[5]: ['green', 'orange', 'red', 'violet', 'yellow']

In [6]: rainbow.reverse()

In [7]: rainbow
Out[7]: ['yellow', 'violet', 'red', 'orange', 'green']

In [8]: rainbow.remove('orange')

In [9]: rainbow
Out[9]: ['yellow', 'violet', 'red', 'green']
```

5.11 Simulating Stacks with Lists

The preceding chapter introduced the function-call stack. Python does not have a built-in stack type, but you can think of a stack as a constrained list. You *push* using list method `append`, which adds a new element to the *end* of the list. You *pop* using list method `pop` with no arguments, which removes and returns the item at the *end* of the list.

Let's create an empty list called `stack`, push (append) two strings onto it, then pop the strings to confirm they're retrieved in last-in, first-out (LIFO) order:

```
In [1]: stack = []
In [2]: stack.append('red')
In [3]: stack
Out[3]: ['red']
```

```
In [4]: stack.append('green')

In [5]: stack
Out[5]: ['red', 'green']

In [6]: stack.pop()
Out[6]: 'green'

In [7]: stack
Out[7]: ['red']

In [8]: stack.pop()
Out[8]: 'red'

In [9]: stack
Out[9]: []

In [10]: stack.pop()
```

`IndexError` Traceback (most recent call last)
`<ipython-input-10-50ea7ec13fbe> in <module>()`
`----> 1 stack.pop()`

`IndexError: pop from empty list`

For each `pop` snippet, the value that `pop` removes and returns is displayed. Popping from an empty stack causes an `IndexError`, just like accessing a nonexistent list element with `[]`. To prevent an `IndexError`, ensure that `len(stack)` is greater than 0 before calling `pop`. You can run out of memory if you keep pushing items faster than you `pop` them.

In the exercises, you'll use a list to simulate another popular collection called a `queue` in which you insert at the back and delete from the front. Items are retrieved from queues in **first-in, first-out (FIFO) order**.



Self Check

1 (*Fill-In*) You can simulate a stack with a list, using methods _____ and _____ to add and remove elements, respectively, only at the end of the list.

Answer: `append`, `pop`.

2 (*Fill-In*) To prevent an `IndexError` when calling `pop` on a list, first ensure that _____.

Answer: the list's length is greater than 0.

5.12 List Comprehensions

Here, we continue discussing *functional-style* features with **list comprehensions**—a concise and convenient notation for creating new lists. List comprehensions can replace many `for` statements that iterate over existing sequences and create new lists, such as:

```
In [1]: list1 = []

In [2]: for item in range(1, 6):
    ....      list1.append(item)
    ....
```

```
In [3]: list1
Out[3]: [1, 2, 3, 4, 5]
```

Using a List Comprehension to Create a List of Integers

We can accomplish the same task in a single line of code with a list comprehension:

```
In [4]: list2 = [item for item in range(1, 6)]
```

```
In [5]: list2
Out[5]: [1, 2, 3, 4, 5]
```

Like snippet [2]’s for statement, the list comprehension’s **for clause**

```
for item in range(1, 6)
```

iterates over the sequence produced by `range(1, 6)`. For each `item`, the list comprehension evaluates the expression to the left of the `for` clause and places the expression’s value (in this case, the `item` itself) in the new list. Snippet [4]’s particular comprehension could have been expressed more concisely using the function `list`:

```
list2 = list(range(1, 6))
```

Mapping: Performing Operations in a List Comprehension’s Expression

A list comprehension’s expression can perform tasks, such as calculations, that `map` elements to new values (possibly of different types). Mapping is a common functional-style programming operation that produces a result with the *same* number of elements as the original data being mapped. The following comprehension maps each value to its cube with the expression `item ** 3`:

```
In [6]: list3 = [item ** 3 for item in range(1, 6)]
```

```
In [7]: list3
Out[7]: [1, 8, 27, 64, 125]
```

Filtering: List Comprehensions with if Clauses

Another common functional-style programming operation is `filtering` elements to select only those that match a condition. This typically produces a list with *fewer* elements than the data being filtered. To do this in a list comprehension, use the **if clause**. The following includes in `list4` only the even values produced by the `for` clause:

```
In [8]: list4 = [item for item in range(1, 11) if item % 2 == 0]
```

```
In [9]: list4
Out[9]: [2, 4, 6, 8, 10]
```

List Comprehension That Processes Another List’s Elements

The `for` clause can process any iterable. Let’s create a list of lowercase strings and use a list comprehension to create a new list containing their uppercase versions:

```
In [10]: colors = ['red', 'orange', 'yellow', 'green', 'blue']
```

```
In [11]: colors2 = [item.upper() for item in colors]
```

```
In [12]: colors2
Out[12]: ['RED', 'ORANGE', 'YELLOW', 'GREEN', 'BLUE']
```

```
In [13]: colors
Out[13]: ['red', 'orange', 'yellow', 'green', 'blue']
```



Self Check

- 1 (*Fill-In*) A list comprehension's _____ clause iterates over the specified sequence.

Answer: `for`.

- 2 (*Fill-In*) A list comprehension's _____ clause filters sequence elements to select only those that match a condition.

Answer: `if`.

- 3 (*IPython Session*) Use a list comprehension to create a list of tuples containing the numbers 1–5 and their cubes—that is, [(1, 1), (2, 8), (3, 27), ...]. To create tuples, place parentheses around the expression to the left of the list comprehension’s `for` clause.

Answer:

```
In [1]: cubes = [(x, x ** 3) for x in range(1, 6)]
```

```
In [2]: cubes
Out[2]: [(1, 1), (2, 8), (3, 27), (4, 64), (5, 125)]
```

- 4 (*IPython Session*) Use a list comprehension and the `range` function with a step to create a list of the multiples of 3 that are less than 30.

Answer:

```
In [3]: multiples = [x for x in range(3, 30, 3)]
```

```
In [4]: multiples
Out[4]: [3, 6, 9, 12, 15, 18, 21, 24, 27]
```

5.13 Generator Expressions

A **generator expression** is similar to a list comprehension, but creates an iterable **generator object** that produces values *on demand*. This is known as **lazy evaluation**. List comprehensions use **greedy evaluation**—they create lists *immediately* when you execute them. For large numbers of items, creating a list can take substantial memory and time. So generator expressions can reduce your program’s memory consumption and improve performance if the whole list is not needed at once.

Generator expressions have the same capabilities as list comprehensions, but you define them in parentheses instead of square brackets. The generator expression in snippet [2] squares and returns only the odd values in `numbers`:

```
In [1]: numbers = [10, 3, 7, 1, 9, 4, 2, 8, 5, 6]
```

```
In [2]: for value in (x ** 2 for x in numbers if x % 2 != 0):
    ...:     print(value, end=' ')
...
9 49 1 81 25
```

To show that a generator expression does not create a list, let’s assign the preceding snippet’s generator expression to a variable and evaluate the variable:

```
In [3]: squares_of_odds = (x ** 2 for x in numbers if x % 2 != 0)
```

```
In [3]: squares_of_odds
Out[3]: <generator object <genexpr> at 0x1085e84c0>
```

The text "generator object <genexpr>" indicates that `squares_of_odds` is a generator object that was created from a generator expression (`genexpr`).



Self Check

- 1 (Fill-In) A generator expression is _____—it produces values on demand.

Answer: lazy.

- 2 (IPython Session) Create a generator expression that cubes the even integers in a list containing 10, 3, 7, 1, 9, 4 and 2. Use function `list` to create a list of the results. Note that the function call's parentheses also act as the generator expression's parentheses.

Answer:

```
In [1]: list(x ** 3 for x in [10, 3, 7, 1, 9, 4, 2] if x % 2 == 0)
Out[1]: [1000, 64, 8]
```

5.14 Filter, Map and Reduce

The preceding section introduced several functional-style features—list comprehensions, filtering and mapping. Here we demonstrate the built-in `filter` and `map` functions for filtering and mapping, respectively. We continue discussing reductions in which you process a collection of elements into a *single* value, such as their count, total, product, average, minimum or maximum.

Filtering a Sequence’s Values with the Built-In `filter` Function

Let’s use built-in function `filter` to obtain the odd values in `numbers`:

```
In [1]: numbers = [10, 3, 7, 1, 9, 4, 2, 8, 5, 6]

In [2]: def is_odd(x):
...:     """Returns True only if x is odd."""
...:     return x % 2 != 0
...:

In [3]: list(filter(is_odd, numbers))
Out[3]: [3, 7, 1, 9, 5]
```

Like data, Python functions are objects that you can assign to variables, pass to other functions and return from functions. Functions that receive other functions as arguments are a functional-style capability called **higher-order functions**. For example, `filter`’s first argument must be a function that receives one argument and returns `True` if the value should be included in the result. The function `is_odd` returns `True` if its argument is odd. The `filter` function calls `is_odd` once for each value in its second argument’s iterable (`numbers`). Higher-order functions may also return a function as a result.

Function `filter` returns an iterator, so `filter`’s results are not produced until you iterate through them. This is another example of lazy evaluation. In snippet [3], function

`list` iterates through the results and creates a list containing them. We can obtain the same results as above by using a list comprehension with an `if` clause:

```
In [4]: [item for item in numbers if is_odd(item)]
Out[4]: [3, 7, 1, 9, 5]
```

Using a Lambda Rather than a Function

For simple functions like `is_odd` that return only a *single expression's value*, you can use a **lambda expression** (or simply a `lambda`) to define the function inline where it's needed—typically as it's passed to another function:

```
In [5]: list(filter(lambda x: x % 2 != 0, numbers))
Out[5]: [3, 7, 1, 9, 5]
```

We pass `filter`'s return value (an iterator) to function `list` here to convert the results to a list and display them.

A lambda expression is an *anonymous function*—that is, a *function without a name*. In the `filter` call

```
filter(lambda x: x % 2 != 0, numbers)
```

the first argument is the lambda

```
lambda x: x % 2 != 0
```

A lambda begins with the `lambda` keyword followed by a comma-separated parameter list, a colon (`:`) and an expression. In this case, the parameter list has one parameter named `x`. A lambda *implicitly* returns its expression's value. So any simple function of the form

```
def function_name(parameter_list):
    return expression
```

may be expressed as a more concise lambda of the form

```
lambda parameter_list: expression
```

Mapping a Sequence's Values to New Values

Let's use built-in function `map` with a lambda to square each value in `numbers`:

```
In [6]: numbers
Out[6]: [10, 3, 7, 1, 9, 4, 2, 8, 5, 6]
```

```
In [7]: list(map(lambda x: x ** 2, numbers))
Out[7]: [100, 9, 49, 1, 81, 16, 4, 64, 25, 36]
```

Function `map`'s first argument is a function that receives one value and returns a new value—in this case, a lambda that squares its argument. The second argument is an iterable of values to map. Function `map` uses lazy evaluation. So, we pass to the `list` function the iterator that `map` returns. This enables us to iterate through and create a list of the mapped values. Here's an equivalent list comprehension:

```
In [8]: [item ** 2 for item in numbers]
Out[8]: [100, 9, 49, 1, 81, 16, 4, 64, 25, 36]
```

Combining filter and map

You can combine the preceding `filter` and `map` operations as follows:

```
In [9]: list(map(lambda x: x ** 2,
...:     filter(lambda x: x % 2 != 0, numbers)))
...:
Out[9]: [9, 49, 1, 81, 25]
```

There is a lot going on in snippet [9], so let's take a closer look at it. First, `filter` returns an iterable representing only the odd values of `numbers`. Then `map` returns an iterable representing the squares of the filtered values. Finally, `list` uses `map`'s iterable to create the list. You might prefer the following list comprehension to the preceding snippet:

```
In [10]: [x ** 2 for x in numbers if x % 2 != 0]
Out[10]: [9, 49, 1, 81, 25]
```

For each value of `x` in `numbers`, the expression `x ** 2` is performed only if the condition `x % 2 != 0` is True.

Reduction: Totaling the Elements of a Sequence with sum

As you know reductions process a sequence's elements into a single value. You've performed reductions with the built-in functions `len`, `sum`, `min` and `max`. You also can create custom reductions using the `functools` module's `reduce` function. See <https://docs.python.org/3/library/functools.html> for a code example. When we investigate big data and Hadoop (introduced briefly in Chapter 1), we'll demonstrate MapReduce programming, which is based on the `filter`, `map` and `reduce` operations in functional-style programming.



Self Check

1 (Fill-In) _____, _____ and _____ are common operations used in functional-style programming.

Answer: Filter, map, reduce.

2 (Fill-In) A(n) _____ processes a sequence's elements into a single value, such as their count, total or average.

Answer: reduction.

3 (IPython Session) Create a list called `numbers` containing 1 through 15, then perform the following tasks:

- Use the built-in function `filter` with a lambda to select only `numbers`' even elements. Create a new list containing the result.
- Use the built-in function `map` with a lambda to square the values of `numbers`' elements. Create a new list containing the result.
- Filter `numbers`' even elements, then map them to their squares. Create a new list containing the result.

Answer:

```
In [1]: numbers = list(range(1, 16))
In [2]: numbers
Out[2]: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
In [3]: list(filter(lambda x: x % 2 == 0, numbers))
```

```
Out[3]: [2, 4, 6, 8, 10, 12, 14]  
In [4]: list(map(lambda x: x ** 2, numbers))  
Out[4]: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225]  
In [5]: list(map(lambda x: x**2, filter(lambda x: x % 2 == 0, numbers)))  
Out[5]: [4, 16, 36, 64, 100, 144, 196]
```

- 4 (*IPython Session*) Map a list of the three Fahrenheit temperatures 41, 32 and 212 to a list of tuples containing the Fahrenheit temperatures and their Celsius equivalents. Convert Fahrenheit temperatures to Celsius with the following formula:

```
Celsius = (Fahrenheit - 32) * (5 / 9)
```

Answer:

```
In [6]: fahrenheit = [41, 32, 212]  
In [7]: list(map(lambda x: (x, (x - 32) * 5 / 9), fahrenheit))  
Out[7]: [(41, 5.0), (32, 0.0), (212, 100.0)]
```

The lambda's expression— $(x, (x - 32) * 5 / 9)$ —uses parentheses to create a tuple containing the original Fahrenheit temperature (x) and the corresponding Celsius temperature, as calculated by $(x - 32) * 5 / 9$.

5.15 Other Sequence Processing Functions

Python provides other built-in functions for manipulating sequences.

Finding the Minimum and Maximum Values Using a Key Function

We've previously shown the built-in reduction functions `min` and `max` using arguments, such as `ints` or lists of `ints`. Sometimes you'll need to find the minimum and maximum of more complex objects, such as strings. Consider the following comparison:

```
In [1]: 'Red' < 'orange'  
Out[1]: True
```

The letter 'R' "comes after" 'o' in the alphabet, so you might expect 'Red' to be less than 'orange' and the condition above to be `False`. However, strings are compared by their characters' underlying *numerical values*, and lowercase letters have *higher* numerical values than uppercase letters. You can confirm this with built-in function `ord`, which returns the numerical value of a character:

```
In [2]: ord('R')  
Out[2]: 82  
  
In [3]: ord('o')  
Out[3]: 111
```

Consider the list `colors`, which contains strings with uppercase and lowercase letters:

```
In [4]: colors = ['Red', 'orange', 'Yellow', 'green', 'Blue']
```

Let's assume that we'd like to determine the minimum and maximum strings using *alphabetical* order, not *numerical* (lexicographical) order. If we arrange colors alphabetically

```
'Blue', 'green', 'orange', 'Red', 'Yellow'
```

you can see that 'Blue' is the minimum (that is, closest to the beginning of the alphabet), and 'Yellow' is the maximum (that is, closest to the end of the alphabet).

Since Python compares strings using numerical values, you must first convert each string to all lowercase or all uppercase letters. Then their numerical values will also represent *alphabetical* ordering. The following snippets enable `min` and `max` to determine the minimum and maximum strings alphabetically:

```
In [5]: min(colors, key=lambda s: s.lower())
Out[5]: 'Blue'
```

```
In [6]: max(colors, key=lambda s: s.lower())
Out[6]: 'Yellow'
```

The `key` keyword argument must be a one-parameter function that returns a value. In this case, it's a `lambda` that calls string method `lower` to get a string's lowercase version. Functions `min` and `max` call the `key` argument's function for each element and use the results to compare the elements.

Iterating Backward Through a Sequence

Built-in function `reversed` returns an iterator that enables you to iterate over a sequence's values backward. The following list comprehension creates a new list containing the squares of numbers' values in reverse order:

```
In [7]: numbers = [10, 3, 7, 1, 9, 4, 2, 8, 5, 6]
In [7]: reversed_numbers = [item for item in reversed(numbers)]
In [8]: reversed_numbers
Out[8]: [36, 25, 64, 4, 16, 81, 1, 49, 9, 100]
```

Combining Iterables into Tuples of Corresponding Elements

Built-in function `zip` enables you to iterate over *multiple* iterables of data at the *same* time. The function receives as arguments any number of iterables and returns an iterator that produces tuples containing the elements at the same index in each. For example, snippet [11]'s call to `zip` produces the tuples ('Bob', 3.5), ('Sue', 4.0) and ('Amanda', 3.75) consisting of the elements at index 0, 1 and 2 of each list, respectively:

```
In [9]: names = ['Bob', 'Sue', 'Amanda']
In [10]: grade_point_averages = [3.5, 4.0, 3.75]
In [11]: for name, gpa in zip(names, grade_point_averages):
...:     print(f'Name={name}; GPA={gpa}')
...
Name=Bob; GPA=3.5
Name=Sue; GPA=4.0
Name=Amanda; GPA=3.75
```

We unpack each tuple into `name` and `gpa` and display them. Function `zip`'s shortest argument determines the number of tuples produced. Here both have the same length.



Self Check

- 1** (*True/False*) The letter 'V' "comes after" the letter 'g' in the alphabet, so the comparison 'Violet' < 'green' yields False.

Answer: False. Strings are compared by their characters' underlying numerical values. Lowercase letters have *higher* numerical values than uppercase. So, the comparison is True.

- 2** (*Fill-In*) Built-in function _____ returns an iterator that enables you to iterate over a sequence's values backward.

Answer: reversed.

- 3** (*IPython Session*) Create the list foods containing 'Cookies', 'pizza', 'Grapes', 'apples', 'steak' and 'Bacon'. Find the smallest string with min, then reimplement the min call using the key function to ignore the strings' case. Do you get the same results? Why or why not?

Answer: The min result was different because 'apples' is the smallest string when you compare them without case sensitivity.

```
In [1]: foods = ['Cookies', 'pizza', 'Grapes',
...:             'apples', 'steak', 'Bacon']
...:

In [2]: min(foods)
Out[2]: 'Bacon'

In [3]: min(foods, key=lambda s: s.lower())
Out[3]: 'apples'
```

- 4** (*IPython Session*) Use zip with two integer lists to create a new list containing the sum of the elements from corresponding indices in both lists (that is, add the elements at index 0, add the elements at index 1, ...).

Answer:

```
In [4]: [(a + b) for a, b in zip([10, 20, 30], [1, 2, 3])]
Out[4]: [11, 22, 33]
```

5.16 Two-Dimensional Lists

Lists can contain other lists as elements. A typical use of such nested (or multidimensional) lists is to represent **tables** of values consisting of information arranged in **rows** and **columns**. To identify a particular table element, we specify *two* indices—by convention, the first identifies the element's row, the second the element's column.

Lists that require two indices to identify an element are called **two-dimensional lists** (or **double-indexed lists** or **double-subscripted lists**). Multidimensional lists can have more than two indices. Here, we introduce two-dimensional lists.

Creating a Two-Dimensional List

Consider a two-dimensional list with three rows and four columns (i.e., a 3-by-4 list) that might represent the grades of three students who each took four exams in a course:

```
In [1]: a = [[77, 68, 86, 73], [96, 87, 89, 81], [70, 90, 86, 81]]
```

Writing the list as follows makes its row and column tabular structure clearer:

```
a = [[77, 68, 86, 73], # first student's grades
      [96, 87, 89, 81], # second student's grades
      [70, 90, 86, 81]] # third student's grades
```

Illustrating a Two-Dimensional List

The diagram below shows the list `a`, with its rows and columns of exam grade values:

	Column 0	Column 1	Column 2	Column 3
Row 0	77	68	86	73
Row 1	96	87	89	81
Row 2	70	90	86	81

Identifying the Elements in a Two-Dimensional List

The following diagram shows the names of list `a`'s elements:

	Column 0	Column 1	Column 2	Column 3
Row 0	<code>a[0][0]</code>	<code>a[0][1]</code>	<code>a[0][2]</code>	<code>a[0][3]</code>
Row 1	<code>a[1][0]</code>	<code>a[1][1]</code>	<code>a[1][2]</code>	<code>a[1][3]</code>
Row 2	<code>a[2][0]</code>	<code>a[2][1]</code>	<code>a[2][2]</code>	<code>a[2][3]</code>

Column index
 Row index
 List name

Every element is identified by a name of the form `a[i][j]`—`a` is the list's name, and i and j are the indices that uniquely identify each element's row and column, respectively. The element names in row 0 all have 0 as the first index. The element names in column 3 all have 3 as the second index.

In the two-dimensional list `a`:

- 77, 68, 86 and 73 initialize `a[0][0]`, `a[0][1]`, `a[0][2]` and `a[0][3]`, respectively,
- 96, 87, 89 and 81 initialize `a[1][0]`, `a[1][1]`, `a[1][2]` and `a[1][3]`, respectively, and
- 70, 90, 86 and 81 initialize `a[2][0]`, `a[2][1]`, `a[2][2]` and `a[2][3]`, respectively.

A list with m rows and n columns is called an **m-by-n list** and has $m \times n$ elements.

The following nested `for` statement outputs the rows of the preceding two-dimensional list one row at a time:

```
In [2]: for row in a:
...:   for item in row:
...:     print(item, end=' ')
...:   print()
...
77 68 86 73
96 87 89 81
70 90 86 81
```

How the Nested Loops Execute

Let's modify the nested loop to display the list's name and the row and column indices and value of each element:

```
In [3]: for i, row in enumerate(a):
...:     for j, item in enumerate(row):
...:         print(f'a[{i}][{j}]={item} ', end=' ')
...:     print()
...
a[0][0]=77  a[0][1]=68  a[0][2]=86  a[0][3]=73
a[1][0]=96  a[1][1]=87  a[1][2]=89  a[1][3]=81
a[2][0]=70  a[2][1]=90  a[2][2]=86  a[2][3]=81
```

The outer `for` statement iterates over the two-dimensional list's rows one row at a time. During each iteration of the outer `for` statement, the inner `for` statement iterates over *each* column in the current row. So in the first iteration of the outer loop, `row 0` is

`[77, 68, 86, 73]`

and the nested loop iterates through this list's four elements `a[0][0]=77`, `a[0][1]=68`, `a[0][2]=86` and `a[0][3]=73`.

In the second iteration of the outer loop, `row 1` is

`[96, 87, 89, 81]`

and the nested loop iterates through this list's four elements `a[1][0]=96`, `a[1][1]=87`, `a[1][2]=89` and `a[1][3]=81`.

In the third iteration of the outer loop, `row 2` is

`[70, 90, 86, 81]`

and the nested loop iterates through this list's four elements `a[2][0]=70`, `a[2][1]=90`, `a[2][2]=86` and `a[2][3]=81`.

In the “Array-Oriented Programming with NumPy” chapter, we'll cover the NumPy library's `ndarray` collection and the Pandas library's `DataFrame` collection. These enable you to manipulate multidimensional collections more concisely and conveniently than the two-dimensional list manipulations you've seen in this section.



Self Check

- 1 (*Fill-In*) In a two-dimensional list, the first index by convention identifies the _____ of an element and the second index identifies the _____ of an element.

Answer: row, column.

- 2 (*Label the Elements*) Label the elements of the two-by-three list `sales` to indicate the order in which they're set to zero by the following program segment:

```
for row in range(len(sales)):
    for col in range(len(sales[row])):
        sales[row][col] = 0
```

Answer: `sales[0][0], sales[0][1], sales[0][2], sales[1][0], sales[1][1], sales[1][2]`.

- 3 (*Two-Dimensional Array*) Consider a two-by-three integer list `t`.

- How many rows does `t` have?
- How many columns does `t` have?

- c) How many elements does `t` have?
- d) What are the names of the elements in row 1?
- e) What are the names of the elements in column 2?
- f) Set the element in row 0 and column 1 to 10.
- g) Write a nested `for` statement that sets each element to the sum of its indices.

Answer:

- a) 2.
- b) 3.
- c) 6.
- d) `t[1][0], t[1][1], t[1][2]`.
- e) `t[0][2], t[1][2]`.
- f) `t[0][1] = 10`.
- g) `for row in range(len(t)):`
 `for column in range(len(t[row])):`
 `t[row][column] = row + column`

4 (*IPython Session*) Given the two-by-three integer list `t`

```
t = [[10, 7, 3], [20, 4, 17]]
```

- a) Determine and display the average of `t`'s elements using nested `for` statements to iterate through the elements.
- b) Write a `for` statement that determines and displays the average of `t`'s elements using the reductions `sum` and `len` to calculate the sum of each row's elements and the number of elements in each row.

Answer:

```
In [1]: t = [[10, 7, 3], [20, 4, 17]]  
  
In [2]: total = 0  
  
In [3]: items = 0  
  
In [4]: for row in t:  
...:     for item in row:  
...:         total += item  
...:         items += 1  
...:  
  
In [5]: total / items  
Out[5]: 10.16666666666666  
  
In [6]: total = 0  
  
In [7]: items = 0  
  
In [8]: for row in t:  
...:     total += sum(row)  
...:     items += len(row)  
...:  
  
In [9]: total / items  
Out[9]: 10.16666666666666
```

5.17 Intro to Data Science: Simulation and Static Visualizations

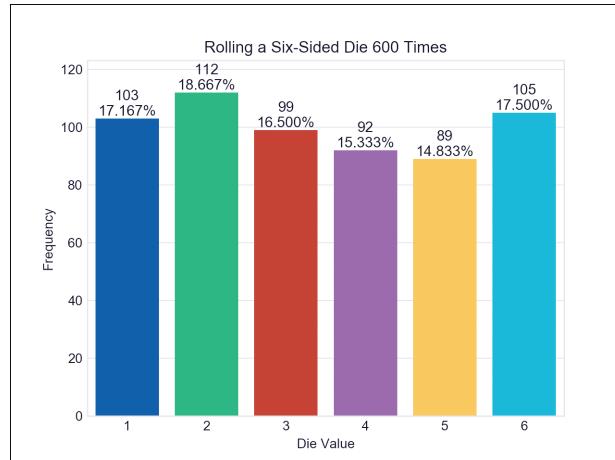
The last few chapters' Intro to Data Science sections discussed basic descriptive statistics. Here, we focus on visualizations, which help you "get to know" your data. Visualizations give you a powerful way to understand data that goes beyond simply looking at raw data.

We use two open-source visualization libraries—Seaborn and Matplotlib—to display *static* bar charts showing the final results of a six-sided-die-rolling simulation. The **Seaborn visualization library** is built over the **Matplotlib visualization library** and simplifies many Matplotlib operations. We'll use aspects of both libraries, because some of the Seaborn operations return objects from the Matplotlib library.

In the next chapter's Intro to Data Science section, we'll make things "come alive" with *dynamic visualizations*. In this chapter's exercises, you'll use simulation techniques and explore the characteristics of some popular card and dice games.

5.17.1 Sample Graphs for 600, 60,000 and 6,000,000 Die Rolls

The screen capture below shows a vertical bar chart that for 600 die rolls summarizes the frequencies with which each of the six faces appear, and their percentages of the total. Seaborn refers to this type of graph as a **bar plot**:



Here we expect about 100 occurrences of each die face. However, with such a small number of rolls, none of the frequencies is exactly 100 (though several are close) and most of the percentages are not close to 16.667% (about 1/6th). As we run the simulation for 60,000 die rolls, the bars will become much closer in size. At 6,000,000 die rolls, they'll appear to be exactly the same size. This is the "law of large numbers" at work. The next chapter will show the lengths of the bars changing dynamically.

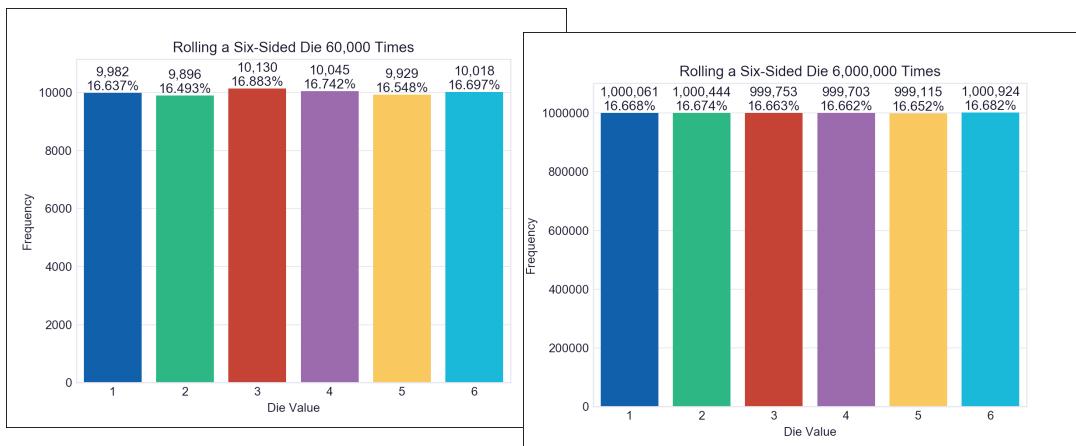
We'll discuss how to control the plot's appearance and contents, including:

- the graph title inside the window (**Rolling a Six-Sided Die 600 Times**),
- the descriptive labels **Die Value** for the *x*-axis and **Frequency** for the *y*-axis,

- the text displayed above each bar, representing the *frequency* and *percentage* of the total rolls, and
- the bar colors.

We'll use various Seaborn default options. For example, Seaborn determines the text labels along the *x*-axis from the die face values 1–6 and the text labels along the *y*-axis from the actual die frequencies. Behind the scenes, Matplotlib determines the positions and sizes of the bars, based on the window size and the magnitudes of the values the bars represent. It also positions the **Frequency** axis's numeric labels based on the actual die frequencies that the bars represent. There are many more features you can customize. You should tweak these attributes to your personal preferences.

The first screen capture below shows the results for 60,000 die rolls—imagine trying to do this by hand. In this case, we expect about 10,000 of each face. The second screen capture below shows the results for 6,000,000 rolls—surely something you'd never do by hand!¹ In this case, we expect about 1,000,000 of each face, and the frequency bars appear to be identical in length (they're close but not exactly the same length). Note that with more die rolls, the frequency percentages are much closer to the expected 16.667%.



Self Check

I *(Discussion)* If you toss a coin a large odd number of times and associate the value 1 with heads and 2 with tails, what would you expect the mean to be? What would you expect the median and mode to be?

Answer: We'd expect the mean to be 1.5, which seems strange because it's not one of the possible outcomes. As the number of coin tosses increases, the percentages of heads and tails should each approach 50% of the total. However, at any given time, they are not likely to be identical. You're just as likely to have a few more heads than tails as vice versa, and as the number of rolls increases, the face with the larger number of rolls could change repeatedly. There are only two possible outcomes, so the median and mode values will be

- When we taught die rolling in our first programming book in the mid-1970s, computers were so much slower that we had to limit our simulations to 6000 rolls. In writing this book's examples, we went to 6,000,000 rolls, which the program completed in a few seconds. We then went to 60,000,000 rolls, which took about a minute.

whichever value there is more of at a given time. So if there are currently more heads than tails, both the median and mode will be heads; otherwise, they'll both be tails. Similar observations apply to die rolling.

5.17.2 Visualizing Die-Roll Frequencies and Percentages

In this section, you'll interactively develop the bar plots shown in the preceding section.

Launching IPython for Interactive Matplotlib Development

IPython has built-in support for interactively developing Matplotlib graphs, which you also need to develop Seaborn graphs. Simply launch IPython with the command:

```
ipython --matplotlib
```

Importing the Libraries

First, let's import the libraries we'll use:

```
In [1]: import matplotlib.pyplot as plt
In [2]: import numpy as np
In [3]: import random
In [4]: import seaborn as sns
```

1. The **matplotlib.pyplot module** contains the Matplotlib library's graphing capabilities that we use. This module typically is imported with the name `plt`.
2. The NumPy (Numerical Python) library includes the function `unique` that we'll use to summarize the die rolls. The **numpy module** typically is imported as `np`.
3. The `random` module contains Python's random-number generation functions.
4. The **seaborn module** contains the Seaborn library's graphing capabilities we use. This module typically is imported with the name `sns`. Search for why this curious abbreviation was chosen.

Rolling the Die and Calculating Die Frequencies

Next, let's use a *list comprehension* to create a list of 600 random die values, then use NumPy's `unique` function to determine the unique roll values (most likely all six possible face values) and their frequencies:

```
In [5]: rolls = [random.randrange(1, 7) for i in range(600)]
In [6]: values, frequencies = np.unique(rolls, return_counts=True)
```

The NumPy library provides the high-performance **ndarray** collection, which is typically much faster than lists.² Though we do not use ndarray directly here, the NumPy `unique` function expects an ndarray argument and returns an ndarray. If you pass a list (like `rolls`), NumPy converts it to an ndarray for better performance. The ndarray that `unique` returns we'll simply assign to a variable for use by a Seaborn plotting function.

Specifying the keyword argument `return_counts=True` tells `unique` to count each unique value's number of occurrences. In this case, `unique` returns a tuple of two one-

2. We'll run a performance comparison in Chapter 7 where we discuss ndarray in depth.

dimensional ndarrays containing the sorted unique values and the corresponding frequencies, respectively. We unpack the tuple's ndarrays into the variables `values` and `frequencies`. If `return_counts` is `False`, only the list of unique values is returned.

Creating the Initial Bar Plot

Let's create the bar plot's title, set its style, then graph the die faces and frequencies:

```
In [7]: title = f'Rolling a Six-Sided Die {len(rolls):,} Times'
In [8]: sns.set_style('whitegrid')
In [9]: axes = sns.barplot(x=values, y=frequencies, palette='bright')
```

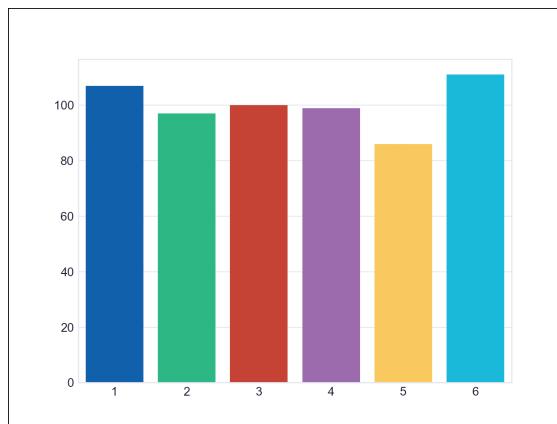
Snippet [7]'s f-string includes the number of die rolls in the bar plot's title. The comma (,) format specifier in

```
{len(rolls):,}
```

displays the number with *thousands separators*—so, 60000 would be displayed as 60,000.

By default, Seaborn plots graphs on a plain white background, but it provides several styles to choose from ('darkgrid', 'whitegrid', 'dark', 'white' and 'ticks'). Snippet [8] specifies the 'whitegrid' style, which displays light-gray horizontal lines in the vertical bar plot. These help you see more easily how each bar's height corresponds to the numeric frequency labels at the bar plot's left side.

Snippet [9] graphs the die frequencies using Seaborn's **barplot** function. When you execute this snippet, the following window appears (because you launched IPython with the --matplotlib option):



Seaborn interacts with Matplotlib to display the bars by creating a Matplotlib **Axes** object, which manages the content that appears in the window. Behind the scenes, Seaborn uses a Matplotlib **Figure** object to manage the window in which the Axes will appear. Function `barplot`'s first two arguments are ndarrays containing the *x*-axis and *y*-axis values, respectively. We used the optional `palette` keyword argument to choose Seaborn's pre-defined color palette 'bright'. You can view the palette options at:

https://seaborn.pydata.org/tutorial/color_palettes.html

Function `barplot` returns the `Axes` object that it configured. We assign this to the variable `axes` so we can use it to configure other aspects of our final plot. Any changes you make

to the bar plot after this point will appear *immediately* when you execute the corresponding snippet.

Setting the Window Title and Labeling the x- and y-Axes

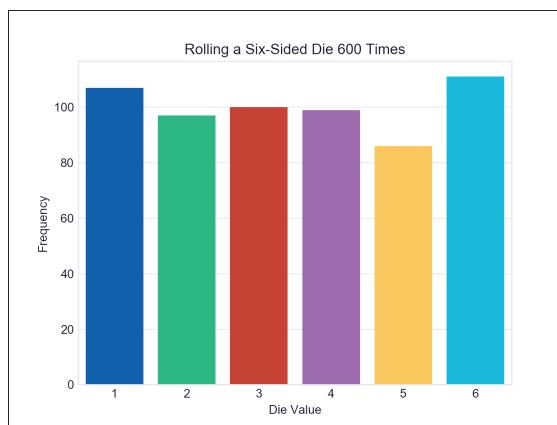
The next two snippets add some descriptive text to the bar plot:

```
In [10]: axes.set_title(title)
Out[10]: Text(0.5,1,'Rolling a Six-Sided Die 600 Times')

In [11]: axes.set(xlabel='Die Value', ylabel='Frequency')
Out[11]: [Text(92.6667,0.5,'Frequency'), Text(0.5,58.7667,'Die Value')]
```

Snippet [10] uses the `axes` object's `set_title` method to display the `title` string centered above the plot. This method returns a `Text` object containing the title and its *location* in the window, which IPython simply displays as output for confirmation. You can ignore the `Out[]`s in the snippets above.

Snippet [11] add labels to each axis. The `set` method receives keyword arguments for the `Axes` object's properties to set. The method displays the `xlabel` text along the *x*-axis, and the `ylabel` text along the *y*-axis, and returns a list of `Text` objects containing the labels and their locations. The bar plot now appears as follows:



Finalizing the Bar Plot

The next two snippets complete the graph by making room for the text above each bar, then displaying it:

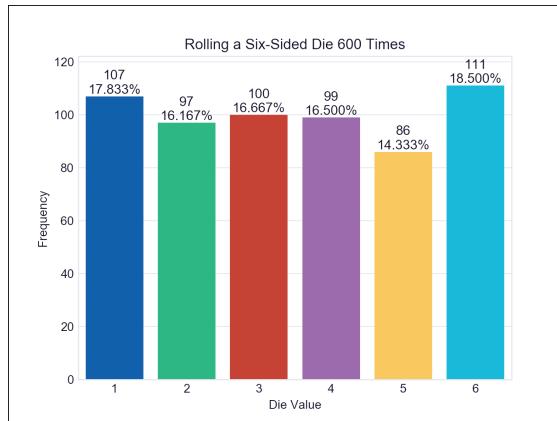
```
In [12]: axes.set_ylim(top=max(frequencies) * 1.10)
Out[12]: (0.0, 122.10000000000001)

In [13]: for bar, frequency in zip(axes.patches, frequencies):
...:     text_x = bar.get_x() + bar.get_width() / 2.0
...:     text_y = bar.get_height()
...:     text = f'{frequency},\n{frequency / len(rolls):.3%}'
...:     axes.text(text_x, text_y, text,
...:               fontsize=11, ha='center', va='bottom')
...:
```

To make room for the text above the bars, snippet [12] scales the *y*-axis by 10%. We chose this value via experimentation. The `Axes` object's `set_ylim` method has many optional keyword arguments. Here, we use only `top` to change the maximum value represented by the *y*-axis. We multiplied the largest frequency by 1.10 to ensure that the *y*-axis is 10% taller than the tallest bar.

Finally, snippet [13] displays each bar's frequency value and percentage of the total rolls. The `axes` object's `patches` collection contains two-dimensional colored shapes that represent the plot's bars. The `for` statement uses `zip` to iterate through the `patches` and their corresponding `frequency` values. Each iteration unpacks into `bar` and `frequency` one of the tuples `zip` returns. The `for` statement's suite operates as follows:

- The first statement calculates the center *x*-coordinate where the text will appear. We calculate this as the sum of the bar's left-edge *x*-coordinate (`bar.get_x()`) and half of the bar's width (`bar.get_width() / 2.0`).
- The second statement gets the *y*-coordinate where the text will appear—`bar.get_y()` represents the bar's top.
- The third statement creates a two-line string containing that bar's frequency and the corresponding percentage of the total die rolls.
- The last statement calls the `Axes` object's `text` method to display the text above the bar. This method's first two arguments specify the text's *x*–*y* position, and the third argument is the text to display. The keyword argument `ha` specifies the *horizontal alignment*—we centered text horizontally around the *x*-coordinate. The keyword argument `va` specifies the *vertical alignment*—we aligned the bottom of the text with at the *y*-coordinate. The final bar plot is shown below:



Rolling Again and Updating the Bar Plot—Introducing IPython Magics

Now that you've created a nice bar plot, you probably want to try a different number of die rolls. First, clear the existing graph by calling Matplotlib's `c1a` (clear axes) function:

In [14]: `plt.cla()`

IPython provides special commands called **magics** for conveniently performing various tasks. Let's use the `%recall` magic to get snippet [5], which created the `rolls` list, and place the code at the next In [] prompt:

```
In [15]: %recall 5
```

```
In [16]: rolls = [random.randrange(1, 7) for i in range(600)]
```

You can now edit the snippet to change the number of rolls to 60000, then press *Enter* to create a new list:

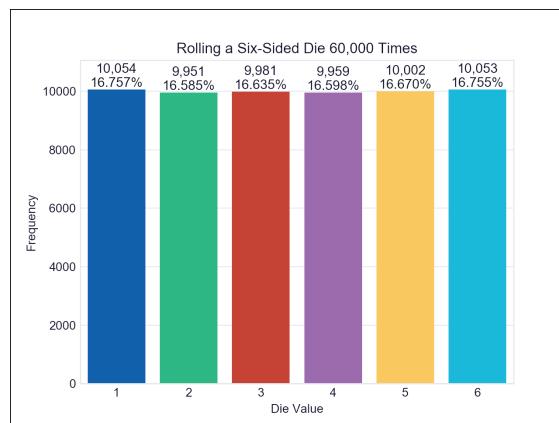
```
In [16]: rolls = [random.randrange(1, 7) for i in range(60000)]
```

Next, recall snippets [6] through [13]. This displays all the snippets in the specified range in the next In [] prompt. Press *Enter* to re-execute these snippets:

```
In [17]: %recall 6-13
```

```
In [18]: values, frequencies = np.unique(rolls, return_counts=True)
....: title = f'Rolling a Six-Sided Die {len(rolls)} Times'
....: sns.set_style('whitegrid')
....: axes = sns.barplot(x=values, y=frequencies, palette='bright')
....: axes.set_title(title)
....: axes.set_xlabel('Die Value', ylabel='Frequency')
....: axes.set_ylim(top=max(frequencies) * 1.10)
....: for bar, frequency in zip(axes.patches, frequencies):
....:     text_x = bar.get_x() + bar.get_width() / 2.0
....:     text_y = bar.get_height()
....:     text = f'{frequency}\n{frequency / len(rolls):.3%}'
....:     axes.text(text_x, text_y, text,
....:               fontsize=11, ha='center', va='bottom')
....:
```

The updated bar plot is shown below:



Saving Snippets to a File with the %save Magic

Once you've interactively created a plot, you may want to save the code to a file so you can turn it into a script and run it in the future. Let's use the **%save magic** to save snippets 1 through 13 to a file named `RollDie.py`. IPython indicates the file to which the lines were written, then displays the lines that it saved:

```
In [19]: %save RollDie.py 1-13
```

The following commands were written to file `RollDie.py`:

```
import matplotlib.pyplot as plt
```

```
import numpy as np
```

```

import random
import seaborn as sns
rolls = [random.randrange(1, 7) for i in range(600)]
values, frequencies = np.unique(rolls, return_counts=True)
title = f'Rolling a Six-Sided Die {len(rolls):,} Times'
sns.set_style("whitegrid")
axes = sns.barplot(values, frequencies, palette='bright')
axes.set_title(title)
axes.set(xlabel='Die Value', ylabel='Frequency')
axes.set_ylim(top=max(frequencies) * 1.10)
for bar, frequency in zip(axes.patches, frequencies):
    text_x = bar.get_x() + bar.get_width() / 2.0
    text_y = bar.get_height()
    text = f'{frequency:,}\n{frequency / len(rolls):.3%}'
    axes.text(text_x, text_y, text,
              fontsize=11, ha='center', va='bottom')

```

Command-Line Arguments; Displaying a Plot from a Script

Provided with this chapter's examples is an edited version of the `RollDie.py` file you saved above. We added comments and a two modifications so you can run the script with an argument that specifies the number of die rolls, as in:

```
ipython RollDie.py 600
```

The Python Standard Library's **sys module** enables a script to receive *command-line arguments* that are passed into the program. These include the script's name and any values that appear to the right of it when you execute the script. The `sys` module's **argv** list contains the arguments. In the command above, `argv[0]` is the *string* '`RollDie.py`' and `argv[1]` is the *string* '`600`'. To control the number of die rolls with the command-line argument's value, we modified the statement that creates the `rolls` list as follows:

```
rolls = [random.randrange(1, 7) for i in range(int(sys.argv[1]))]
```

Note that we converted the `argv[1]` string to an `int`.

Matplotlib and Seaborn do not automatically display the plot for you when you create it in a script. So at the end of the script we added the following call to Matplotlib's **show** function, which displays the window containing the graph:

```
plt.show()
```



Self Check

1 *(Fill-In)* The _____ format specifier indicates that a number should be displayed with thousands separators.

Answer: comma (,),

2 *(Fill-In)* A Matplotlib _____ object manages the content that appears in a Matplotlib window.

Answer: Axes.

3 *(Fill-In)* The Seaborn function _____ displays data as a bar chart.

Answer: barplot.

4 *(Fill-In)* The Matplotlib function _____ displays a plot window from a script.

Answer: show.

5 (IPython Session) Use the %recall magic to repeat the steps in snippets [14] through [18] to redraw the bar plot for 6,000,000 die rolls. This exercise assumes that you’re continuing this section’s IPython session. Notice that the heights of the six bars look the same, although each frequency is close to 1,000,000 and each percentage is close to 16.667%.

Answer:

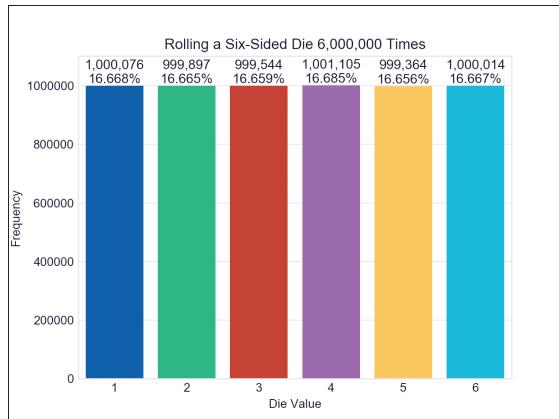
```
In [20]: plt.cla()

In [21]: %recall 5

In [22]: rolls = [random.randrange(1, 7) for i in range(6000000)]

In [23]: %recall 6-13

In [24]: values, frequencies = np.unique(rolls, return_counts=True)
....: title = f'Rolling a Six-Sided Die {len(rolls)} Times'
....: sns.set_style('whitegrid')
....: axes = sns.barplot(values, frequencies, palette='bright')
....: axes.set_title(title)
....: axes.set_xlabel('Die Value', ylabel='Frequency')
....: axes.set_ylim(top=max(frequencies) * 1.10)
....: for bar, frequency in zip(axes.patches, frequencies):
....:     text_x = bar.get_x() + bar.get_width() / 2.0
....:     text_y = bar.get_height()
....:     text = f'{frequency},\n{frequency / len(rolls):.3%}'
....:     axes.text(text_x, text_y, text,
....:               fontsize=11, ha='center', va='bottom')
....:
```



5.18 Wrap-Up

This chapter presented more details of the list and tuple sequences. You created lists, accessed their elements and determined their length. You saw that lists are mutable, so you can modify their contents, including growing and shrinking the lists as your programs execute. You saw that accessing a nonexistent element causes an `IndexError`. You used `for` statements to iterate through list elements.

We discussed tuples, which like lists are sequences, but are immutable. You unpacked a tuple’s elements into separate variables. You used `enumerate` to create an iterable of tuples, each with a list index and corresponding element value.

You learned that all sequences support slicing, which creates new sequences with subsets of the original elements. You used the `del` statement to remove elements from lists and delete variables from interactive sessions. We passed lists, list elements and slices of lists to functions. You saw how to search and sort lists, and how to search tuples. We used list methods to insert, append and remove elements, and to reverse a list’s elements and copy lists.

We showed how to simulate stacks with lists—in an exercise, you’ll use the same list methods to simulate a queue with a list. We used the concise list-comprehension notation to create new lists. We used additional built-in methods to sum list elements, iterate backward through a list, find the minimum and maximum values, filter values and map values to new values. We showed how nested lists can represent two-dimensional tables in which data is arranged in rows and columns. You saw how nested `for` loops process two-dimensional lists.

The chapter concluded with an Intro to Data Science section that presented a die-rolling simulation and static visualizations. A detailed code example used the Seaborn and Matplotlib visualization libraries to create a *static* bar plot visualization of the simulation’s final results. In the next Intro to Data Science section, we use a die-rolling simulation with a *dynamic* bar plot visualization to make the plot “come alive.”

In the next chapter, “Dictionaries and Sets,” we’ll continue our discussion of Python’s built-in collections. We’ll use dictionaries to store unordered collections of key–value pairs that map immutable keys to values, just as a conventional dictionary maps words to definitions. We’ll use sets to store unordered collections of unique elements.

In the “Array-Oriented Programming with NumPy” chapter, we’ll discuss NumPy’s `ndarray` collection in more detail. You’ll see that while lists are fine for small amounts of data, they are not efficient for the large amounts of data you’ll encounter in big data analytics applications. For such cases, the NumPy library’s highly optimized `ndarray` collection should be used. `ndarray` (*n*-dimensional array) can be much faster than lists. We’ll run Python profiling tests to see just how much faster. As you’ll see, NumPy also includes many capabilities for conveniently and efficiently manipulating arrays of *many* dimensions. In big data analytics applications, the processing demands can be humongous, so everything we can do to improve performance significantly matters. In our “Big Data: Hadoop, Spark, NoSQL and IoT” chapter, you’ll use one of the most popular big-data databases—MongoDB.³

Exercises

Use IPython sessions for each exercise where practical.

5.1 (What’s Wrong with This Code?) What, if anything, is wrong with each of the following code segments?

- `day, high_temperature = ('Monday', 87, 65)`
- `numbers = [1, 2, 3, 4, 5]`
`numbers[10]`

3. The database’s name is rooted in the word “humongous.”

- c) `name = 'amanda'`
`name[0] = 'A'`
- d) `numbers = [1, 2, 3, 4, 5]`
`numbers[3.4]`
- e) `student_tuple = ('Amanda', 'Blue', [98, 75, 87])`
`student_tuple[0] = 'Ariana'`
- f) `('Monday', 87, 65) + 'Tuesday'`
- g) `'A' += ('B', 'C')`
- h) `x = 7`
`del x`
`print(x)`
- i) `numbers = [1, 2, 3, 4, 5]`
`numbers.index(10)`
- j) `numbers = [1, 2, 3, 4, 5]`
`numbers.extend(6, 7, 8)`
- k) `numbers = [1, 2, 3, 4, 5]`
`numbers.remove(10)`
- l) `values = []`
`values.pop()`

5.2 (*What's Does This Code Do?*) What does the following function do, based on the sequence it receives as an argument?

```
def mystery(sequence):
    return sequence == sorted(sequence)
```

5.3 (*Fill in the Missing Code*) Replace the ***'s in the following list comprehension and map function call, such that given a list of heights in inches, the code maps the list to a list of tuples containing the original height values and their corresponding values in meters. For example, if one element in the original list contains the height 69 inches, the corresponding element in the new list will contain the tuple (69, 1.7526), representing both the height in inches and the height in meters. There are 0.0254 meters per inch.

```
[*** for x in [69, 77, 54]]
list(map(lambda ***, [69, 77, 54]))
```

5.4 (*Iteration Order*) Create a 2-by-3 list, then use a nested loop to:

- a) Set each element's value to an integer indicating the order in which it was processed by the nested loop.
- b) Display the elements in tabular format. Use the column indices as headings across the top, and the row indices to the left of each row.

5.5 (*IPython Session: Slicing*) Create a string called `alphabet` containing 'abcdefghijklmnopqrstuvwxyz', then perform the following separate slice operations to obtain:

- a) The first half of the string using starting and ending indices.
- b) The first half of the string using only the ending index.
- c) The second half of the string using starting and ending indices.
- d) The second half of the string using only the starting index.
- e) Every second letter in the string starting with 'a'.
- f) The entire string in reverse.
- g) Every third letter of the string in reverse starting with 'z'.

5.6 (Functions Returning Tuples) Define a function `rotate` that receives three arguments and returns a tuple in which the first argument is at index 1, the second argument is at index 2 and the third argument is at index 0. Define variables `a`, `b` and `c` containing 'Doug', 22 and 1984. Then call the function three times. For each call, unpack its result into `a`, `b` and `c`, then display their values.

5.7 (Duplicate Elimination) Create a function that receives a list and returns a (possibly shorter) list containing only the unique values in sorted order. Test your function with a list of numbers and a list of strings.

5.8 (Sieve of Eratosthenes) A prime number is an integer greater than 1 that's evenly divisible only by itself and 1. The Sieve of Eratosthenes is an elegant, straightforward method of finding prime numbers. The process for finding all primes less than 1000 is:

- a) Create a 1000-element list `primes` with all elements initialized to `True`. List elements with prime indices (like 2, 3, 5, 7, 11, ...) will remain `True`. All other list elements will eventually be set to `False`.
- b) Starting with index 2, if a given element is `True` iterate through the rest of the list and set to `False` every element in `primes` whose index is a *multiple* of the index for the element we're currently processing. For list index 2, all elements beyond element 2 in the list that have indices which are multiples of 2 (i.e., 4, 6, 8, 10, ..., 998) will be set to `False`.
- c) Repeat Step (b) for the next `True` element. For list index 3 (which was initialized to `True`), all elements beyond element 3 in the list that have indices which are multiples of 3 (i.e., 6, 9, 12, 15, ..., 999) will be set to `False`; and so on. A subtle observation (think about why this is true): The square root of 999 is 31.6, you'll need to test and set to `False` only all multiples of 2, 3, 5, 7, 9, 11, 13, 17, 19, 23, 29 and 31. This will significantly improve the performance of your algorithm, especially if you decide to look for large prime numbers.

When this process completes, the list elements that are still `True` indicate that the index is a prime number. These indices can be displayed. Use a list of 1000 elements to determine and display the prime numbers less than 1000. Ignore list elements 0 and 1. [As you work through the book, you'll discover other Python capabilities that will enable you to cleverly reimplement this exercise.]

5.9 (Palindrome Tester) A string that's spelled identically backward and forward, like 'radar', is a palindrome. Write a function `is_palindrome` that takes a string and returns `True` if it's a palindrome and `False` otherwise. Use a stack (simulated with a list as we did in Section 5.11) to help determine whether a string is a palindrome. Your function should ignore case sensitivity (that is, 'a' and 'A' are the same), spaces and punctuation.

5.10 (Anagrams) An anagram of a string is another string formed by rearranging the letters in the first. Write a script that produces all possible anagrams of a given string using only techniques that you've seen to this point in the book. [The `itertools` module provides many functions, including one that produces permutations.]

5.11 (Summarizing Letters in a String) Write a function `summarize_letters` that receives a string and returns a list of tuples containing the unique letters and their frequencies in the string. Test your function and display each letter with its frequency. Your function should ignore case sensitivity (that is, 'a' and 'A' are the same) and ignore spaces

and punctuation. When done, write a statement that says whether the string has all the letters of the alphabet.

5.12 (Telephone-Number Word Generator) You should find this exercise to be entertaining. Standard telephone keypads contain the digits zero through nine. The numbers two through nine each have three letters associated with them, as shown in the following table:

Digit	Letters	Digit	Letters	Digit	Letters
2	A B C	5	J K L	8	T U V
3	D E F	6	M N O	9	W X Y
4	G H I	7	P R S		

Many people find it difficult to memorize phone numbers, so they use the correspondence between digits and letters to develop seven-letter words (or phrases) that correspond to their phone numbers. For example, a person whose telephone number is 686-2377 might use the correspondence indicated in the preceding table to develop the seven-letter word “NUMBERS.” Every seven-letter word or phrase corresponds to exactly one seven-digit telephone number. A budding data science entrepreneur might like to reserve the phone number 244-3282 (“BIGDATA”).

Every seven-digit phone number without 0s or 1s corresponds to many different seven-letter words, but most of these words represent unrecognizable gibberish. A veterinarian with the phone number 738-2273 would be pleased to know that the number corresponds to the letters “PETCARE.”

Write a script that, given a seven-digit number, generates every possible seven-letter word combination corresponding to that number. There are 2,187 (3^7) such combinations. Avoid phone numbers with the digits 0 and 1 (to which no letters correspond). See if your phone number corresponds to meaningful words.

5.13 (Word or Phrase to Phone-Number Generator) Just as people would enjoy knowing what word or phrase their phone number corresponds to, they might choose a word or phrase appropriate for their business and determine what phone numbers correspond to it. These are sometimes called vanity phone numbers, and various websites sell such phone numbers. Write a script similar to the one in the previous exercise that produces the possible phone number for the given seven-letter string.

5.14 (Is a Sequence Sorted?) Create a function `is_ordered` that receives a sequence and returns `True` if the elements are in sorted order. Test your function with sorted and unsorted lists, tuples and strings.

5.15 (Tuples Representing Invoices) When you purchase products or services from a company, you typically receive an invoice listing what you purchased and the total amount of money due. Use tuples to represent hardware store invoices that consist of four pieces of data—a part ID string, a part description string, an integer quantity of the item being purchased and, for simplicity, a `float` item price (in general, `Decimal` should be used for monetary amounts). Use the sample hardware data shown in the following table.

Part number	Part description	Quantity	Price
83	Electric sander	7	57.98
24	Power saw	18	99.99
7	Sledge hammer	11	21.50
77	Hammer	76	11.99
39	Jig saw	3	79.50

Perform the following tasks on a list of invoice tuples:

- a) Use function `sorted` with a `key` argument to sort the tuples by part description, then display the results. To specify the element of the tuple that should be used for sorting, first import the `itemgetter` function from the `operator` module as in

```
from operator import itemgetter
```

Then, for `sorted`'s `key` argument specify `itemgetter(index)` where `index` specifies which element of the tuple should be used for sorting purposes.

- b) Use the `sorted` function with a `key` argument to sort the tuples by price, then display the results.
 c) Map each invoice tuple to a tuple containing the part description and quantity, sort the results by quantity, then display the results.
 d) Map each invoice tuple to a tuple containing the part description and the value of the invoice (the product of the quantity and the item price), sort the results by the invoice value, then display the results.
 e) Modify Part (d) to filter the results to invoice values in the range \$200 to \$500.
 f) Calculate the total of all the invoices.

5.16 (Sorting Letters and Removing Duplicates) Insert 20 random letters in the range 'a' through 'f' into a list. Perform the following tasks and display your results:

- a) Sort the list in ascending order.
 b) Sort the list in descending order.
 c) Get the unique values sort them in ascending order.

5.17 (Filter/Map Performance) With regard to the following code:

```
numbers = [10, 3, 7, 1, 9, 4, 2, 8, 5, 6]
list(map(lambda x: x ** 2,
        filter(lambda x: x % 2 != 0, numbers)))
```

- a) How many times does the `filter` operation call its `lambda` argument?
 b) How many times does the `map` operation call its `lambda` argument?
 c) If you reverse the `filter` and `map` operations, how many times does the `map` operation call its `lambda` argument?

To help you answer the preceding questions, define functions that perform the same tasks as the lambdas. In each function, include a `print` statement so you can see each time the function is called. Finally, replace the lambdas in the preceding code with the names of your functions.

5.18 (Summing the Triples of the Even Integers from 2 through 10) Starting with a list containing 1 through 10, use `filter`, `map` and `sum` to calculate the total of the triples of

the even integers from 2 through 10. Reimplement your code with list comprehensions rather than `filter` and `map`.

5.19 (*Finding the People with a Specified Last Name*) Create a list of tuples containing first and last names. Use `filter` to locate the tuples containing the last name `Jones`. Ensure that several tuples in your list have that last name.

5.20 (*Display a Two-Dimensional List in Tabular Format*) Define a function named `display_table` that receives a two-dimensional list and displays its contents in tabular format. List the column indices as headings across the top, and list the row indices at the left of each row.

5.21 (*Computer-Assisted Instruction: Reducing Student Fatigue*) Re-implement Exercise 4.15 to store the computer's responses in lists. Use random-number generation to select responses using random list indices.

5.22 (*Simulating a Queue with a List*) In this chapter, you simulated a stack using a list. You also can simulate a queue collection with a list. **Queues** represent waiting lines similar to a checkout line in a supermarket. The cashier services the person at the *front* of the line *first*. Other customers enter the line only at the end and wait for service.

In a queue, you insert items at the back (known as the **tail**) and delete items from the front (known as the **head**). For this reason, queues are first-in, first-out (FIFO) collections. The insert and remove operations are commonly known as `enqueue` and `dequeue`.

Queues have many uses in computer systems, such as sharing CPUs among a potentially large number of competing applications and the operating system itself. Applications not currently being serviced sit in a queue until a CPU becomes available. The application at the front of the queue is the next to receive service. Each application gradually advances to the front as the applications before it receive service.

Simulate a queue of integers using list methods `append` (to simulate `enqueue`) and `pop` with the argument 0 (to simulate `dequeue`). Enqueue the values 3, 2 and 1, then dequeue them to show that they're removed in FIFO order.

5.23 (*Functional-Style Programming: Order of `filter` and `map` Calls*) When combining `filter` and `map` operations, the order in which they're performed matters. Consider a list `numbers` containing 10, 3, 7, 1, 9, 4, 2, 8, 5, 6 and the following code:

```
In [1]: numbers = [10, 3, 7, 1, 9, 4, 2, 8, 5, 6]

In [2]: list(map(lambda x: x * 2,
    ...:     filter(lambda x: x % 2 == 0, numbers)))
    ...:
Out[3]: [20, 8, 4, 16, 12]
```

Reorder this code to call `map` first and `filter` second. What happens and why?

Exercises 5.24 through 5.26 are reasonably challenging. Once you've done them, you ought to be able to implement many popular card games.

5.24 (*Card Shuffling and Dealing*) In Exercises 5.24 through 5.26, you'll use lists of tuples in scripts that simulate card shuffling and dealing. Each tuple represents one card in the deck and contains a face (e.g., 'Ace', 'Deuce', 'Three', ..., 'Jack', 'Queen', 'King') and a suit (e.g., 'Hearts', 'Diamonds', 'Clubs', 'Spades'). Create an `initialize_deck` function to initialize the deck of card tuples with 'Ace' through 'King' of each suit, as in

```
deck = [('Ace', 'Hearts'), ..., ('King', 'Hearts'),
        ('Ace', 'Diamonds'), ..., ('King', 'Diamonds'),
        ('Ace', 'Clubs'), ..., ('King', 'Clubs'),
        ('Ace', 'Spades'), ..., ('King', 'Spades')]
```

Before returning the list, use the random module's **shuffle** function to randomly order the list elements. Output the shuffled cards in the following four-column format:

Six of Spades	Eight of Spades	Six of Clubs	Nine of Hearts
Queen of Hearts	Seven of Clubs	Nine of Spades	King of Hearts
Three of Diamonds	Deuce of Clubs	Ace of Hearts	Ten of Spades
Four of Spades	Ace of Clubs	Seven of Diamonds	Four of Hearts
Three of Clubs	Deuce of Hearts	Five of Spades	Jack of Diamonds
King of Clubs	Ten of Hearts	Three of Hearts	Six of Diamonds
Queen of Clubs	Eight of Diamonds	Deuce of Diamonds	Ten of Diamonds
Three of Spades	King of Diamonds	Nine of Clubs	Six of Hearts
Ace of Spades	Four of Diamonds	Seven of Hearts	Eight of Clubs
Deuce of Spades	Eight of Hearts	Five of Hearts	Queen of Spades
Jack of Hearts	Seven of Spades	Four of Clubs	Nine of Diamonds
Ace of Diamonds	Queen of Diamonds	Five of Clubs	King of Spades
Five of Diamonds	Ten of Clubs	Jack of Spades	Jack of Clubs

5.25 (Card Playing: Evaluating Poker Hands) Modify Exercise 5.24 to deal a five-card poker hand as a list of five card tuples. Then create functions (i.e., `is_pair`, `is_two_pair`, `is_three_of_a_kind`, ...) that determine whether the hand they receive as an argument contains groups of cards, such as:

- one pair
- two pairs
- three of a kind (e.g., three jacks)
- a straight (i.e., five cards of consecutive face values)
- a flush (i.e., all five cards of the same suit)
- a full house (i.e., two cards of one face value and three cards of another)
- four of a kind (e.g., four aces)
- straight flush (i.e., a straight with all five cards of the same suit)
- ... and others.

See https://en.wikipedia.org/wiki/List_of_poker_hands for poker-hand types and how they rank with respect to one another. For example, three of a kind beats two pairs.

5.26 (Card Playing: Determining the Winning Hand) Use the methods developed in Exercise 5.25 to write a script that deals two five-card poker hands (i.e., two lists of five card tuples each), evaluates each hand and determines which wins. As each card is dealt, it should be removed from the list of tuples representing the deck.

5.27 (Intro to Data Science: Duplicate Elimination and Counting Frequencies) Use a list comprehension to create a list of 50 random values in the range 1 through 10. Use NumPy's `unique` function to obtain the unique values and their frequencies. Display the results.

5.28 (Intro to Data Science: Survey Response Statistics) Twenty students were asked to rate on a scale of 1 to 5 the quality of the food in the student cafeteria, with 1 being "awful" and 5 being "excellent." Place the 20 responses in a list

```
1, 2, 5, 4, 3, 5, 2, 1, 3, 3, 1, 4, 3, 3, 3, 2, 3, 3, 2, 5
```

Determine and display the frequency of each rating. Use the built-in functions, `statistics` module functions and NumPy functions demonstrated in Section 5.17.2 to display the following response statistics: minimum, maximum, range, mean, median, mode, variance and standard deviation.

5.29 (*Intro to Data Science: Visualizing Survey Response Statistics*) Using the list in Exercise 5.28 and the techniques you learned in Section 5.17.2, display a bar chart showing the response frequencies and their percentages of the total responses.

5.30 (*Intro to Data Science: Removing the Text Above the Bars*) Modify the die-rolling simulation in Section 5.17.2 to omit displaying the frequencies and percentages above each bar. Try to minimize the number of lines of code.

5.31 (*Intro to Data Science: Coin Tossing*) Modify the die-rolling simulation in Section 5.17.2 to simulate the flipping a coin. Use randomly generated 1s and 2s to represent heads and tails, respectively. Initially, do not include the frequencies and percentages above the bars. Then modify your code to include the frequencies and percentages. Run simulations for 200, 20,000 and 200,000 coin flips. Do you get approximately 50% heads and 50% tails? Do you see the “law of large numbers” in operation here?

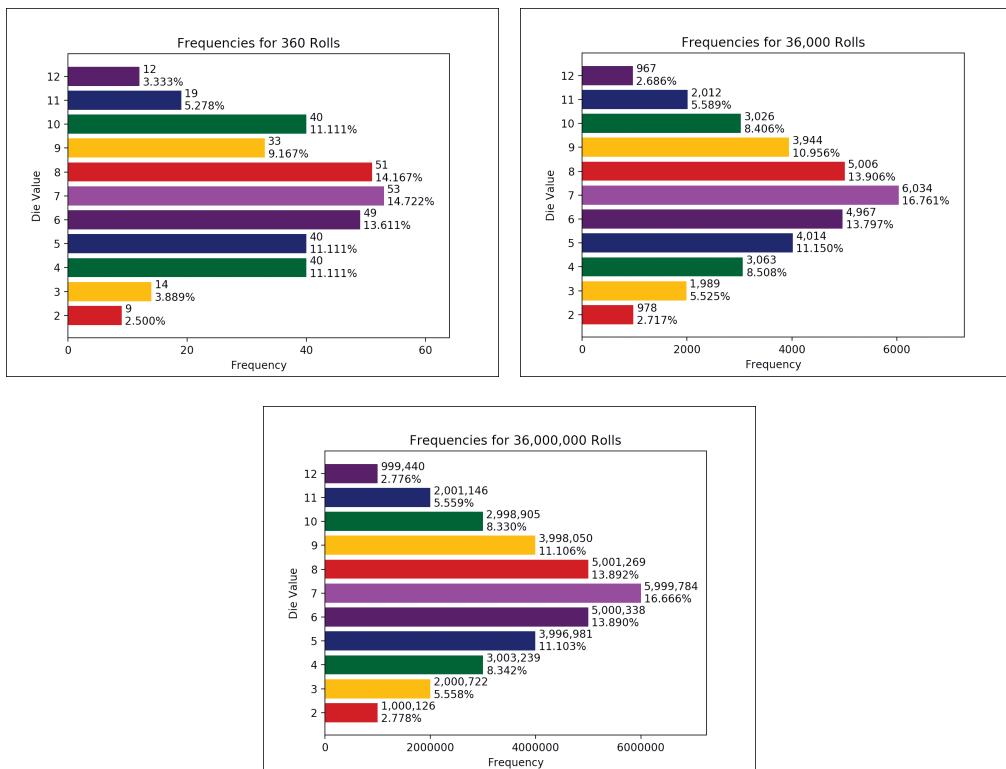
5.32 (*Intro to Data Science: Rolling Two Dice*) Modify the script `RollDie.py` that we provided with this chapter’s examples to simulate rolling two dice. Calculate the sum of the two values. Each die has a value from 1 to 6, so the sum of the values will vary from 2 to 12, with 7 being the most frequent sum, and 2 and 12 the least frequent. The following diagram shows the 36 equally likely possible combinations of the two dice and their corresponding sums:

	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12

If you roll the dice 36,000 times:

- The values 2 and 12 each occur 1/36th (2.778%) of the time, so you should expect about 1000 of each.
- The values 3 and 11 each occur 2/36ths (5.556%) of the time, so you should expect about 2000 of each, and so on.

Use a command-line argument to obtain the number of rolls. Display a bar plot summarizing the roll frequencies. The following screen captures show the final bar plots for sample executions of 360, 36,000 and 36,000,000 rolls. Use the Seaborn `barplot` function’s optional `orient` keyword argument to specify a horizontal bar plot.



5.33 (Intro to Data Science Challenge: Analyzing the Dice Game Craps) In this exercise, you'll modify Chapter 4's script that simulates the dice game craps by using the techniques you learned in Section 5.17.2. The script should receive a command-line argument indicating the number of games of craps to execute and use two lists to track the total numbers of games won and lost on the first roll, second roll, third roll, etc. Summarize the results as follows:

- Display a horizontal bar plot indicating how many games are won and how many are lost on the first roll, second roll, third roll, etc. Since the game could continue indefinitely, you might track wins and losses through the first dozen rolls (of a pair of dice), then maintain two counters that keep track of wins and losses after 12 rolls—no matter how long the game gets. Create separate bars for wins and losses.
- What are the chances of winning at craps? [Note: You should discover that craps is one of the fairest casino games. What do you suppose this means?]
- What is the mean for the length of a game of craps? The median? The mode?
- Do the chances of winning improve with the length of the game?

6

Dictionaries and Sets



Objectives

In this chapter, you'll:

- Use dictionaries to represent unordered collections of key-value pairs.
- Use sets to represent unordered collections of unique values.
- Create, initialize and refer to elements of dictionaries and sets.
- Iterate through a dictionary's keys, values and key–value pairs.
- Add, remove and update a dictionary's key–value pairs.
- Use dictionary and set comparison operators.
- Combine sets with set operators and methods.
- Use operators `in` and `not in` to determine if a dictionary contains a key or a set contains a value.
- Use the mutable set operations to modify a set's contents.
- Use comprehensions to create dictionaries and sets quickly and conveniently.
- Learn how to build dynamic visualizations and implement more of your own in the exercises.
- Enhance your understanding of mutability and immutability.

Outline

6.1	Introduction	6.3	Sets
6.2	Dictionaries	6.3.1	Comparing Sets
6.2.1	Creating a Dictionary	6.3.2	Mathematical Set Operations
6.2.2	Iterating through a Dictionary	6.3.3	Mutable Set Operators and Methods
6.2.3	Basic Dictionary Operations	6.3.4	Set Comprehensions
6.2.4	Dictionary Methods <code>keys</code> and <code>values</code>	6.4	Intro to Data Science: Dynamic Visualizations
6.2.5	Dictionary Comparisons	6.4.1	How Dynamic Visualization Works
6.2.6	Example: Dictionary of Student Grades	6.4.2	Implementing a Dynamic Visualization
6.2.7	Example: Word Counts	6.5	Wrap-Up
6.2.8	Dictionary Method <code>update</code>		Exercises
6.2.9	Dictionary Comprehensions		

6.1 Introduction

We've discussed three built-in sequence collections—strings, lists and tuples. Now, we consider the built-in non-sequence collections—dictionaries and sets. A **dictionary** is an *unordered* collection which stores **key–value pairs** that map immutable keys to values, just as a conventional dictionary maps words to definitions. A **set** is an unordered collection of *unique* immutable elements.

6.2 Dictionaries

A dictionary *associates* keys with values. Each key *maps* to a specific value. The following table contains examples of dictionaries with their keys, key types, values and value types:

Keys	Key type	Values	Value type
Country names	<code>str</code>	Internet country codes	<code>str</code>
Decimal numbers	<code>int</code>	Roman numerals	<code>str</code>
States	<code>str</code>	Agricultural products	list of <code>str</code>
Hospital patients	<code>str</code>	Vital signs	tuple of <code>ints</code> and <code>floats</code>
Baseball players	<code>str</code>	Batting averages	<code>float</code>
Metric measurements	<code>str</code>	Abbreviations	<code>str</code>
Inventory codes	<code>str</code>	Quantity in stock	<code>int</code>

Unique Keys

A dictionary's keys must be *immutable* (such as strings, numbers or tuples) and *unique* (that is, no duplicates). Multiple keys can have the same value, such as two different inventory codes that have the same quantity in stock.

6.2.1 Creating a Dictionary

You can create a dictionary by enclosing in curly braces, `{}`, a comma-separated list of key–value pairs, each of the form `key: value`. You can create an empty dictionary with `{}`.

Let's create a dictionary with the country-name keys 'Finland', 'South Africa' and 'Nepal' and their corresponding Internet country code values 'fi', 'za' and 'np':

```
In [1]: country_codes = {'Finland': 'fi', 'South Africa': 'za',
...:                 'Nepal': 'np'}
...:

In [2]: country_codes
Out[2]: {'Finland': 'fi', 'South Africa': 'za', 'Nepal': 'np'}
```

When you output a dictionary, its comma-separated list of key–value pairs is always enclosed in curly braces. Because dictionaries are *unordered* collections, the display order can differ from the order in which the key–value pairs were added to the dictionary. In snippet [2]’s output the key–value pairs are displayed in the order they were inserted, but do *not* write code that depends on the order of the key–value pairs.

Determining if a Dictionary Is Empty

The built-in function `len` returns the number of key–value pairs in a dictionary:

```
In [3]: len(country_codes)
Out[3]: 3
```

You can use a dictionary as a condition to determine if it’s empty—a non-empty dictionary evaluates to `True`:

```
In [4]: if country_codes:
...:     print('country_codes is not empty')
...: else:
...:     print('country_codes is empty')
...:
country_codes is not empty
```

An empty dictionary evaluates to `False`. To demonstrate this, in the following code we call method `clear` to delete the dictionary’s key–value pairs, then in snippet [6] we recall and re-execute snippet [4]:

```
In [5]: country_codes.clear()

In [6]: if country_codes:
...:     print('country_codes is not empty')
...: else:
...:     print('country_codes is empty')
...:
country_codes is empty
```



Self Check

1 (Fill-In) _____ can be thought of as unordered collections in which each value is accessed through its corresponding key.

Answer: Dictionaries.

2 (True/False) Dictionaries may contain duplicate keys.

Answer: False. Dictionary keys must be unique. However, multiple keys may have the same value.

3 (IPython Session) Create a dictionary named `states` that maps three state abbreviations to their state names, then display the dictionary.

Answer:

```
In [1]: states = {'VT': 'Vermont', 'NH': 'New Hampshire',
...:                 'MA': 'Massachusetts'}
...:

In [2]: states
Out[2]: {'VT': 'Vermont', 'NH': 'New Hampshire', 'MA': 'Massachusetts'}
```

6.2.2 Iterating through a Dictionary

The following dictionary maps month-name strings to `int` values representing the numbers of days in the corresponding month. Note that *multiple* keys can have the *same* value:

```
In [1]: days_per_month = {'January': 31, 'February': 28, 'March': 31}

In [2]: days_per_month
Out[2]: {'January': 31, 'February': 28, 'March': 31}
```

Again, the dictionary's string representation shows the key–value pairs in their insertion order, but this is not guaranteed because dictionaries are *unordered*. We'll show how to process keys in *sorted* order later in this chapter.

The following `for` statement iterates through `days_per_month`'s key–value pairs. Dictionary method `items` returns each key–value pair as a tuple, which we unpack into `month` and `days`:

```
In [3]: for month, days in days_per_month.items():
...:     print(f'{month} has {days} days')
...:
January has 31 days
February has 28 days
March has 31 days
```



Self Check

I (Fill-In) Dictionary method _____ returns each key–value pair as a tuple.

Answer: `items`.

6.2.3 Basic Dictionary Operations

For this section, let's begin by creating and displaying the dictionary `roman_numerals`. We intentionally provide the incorrect value 100 for the key 'X', which we'll correct shortly:

```
In [1]: roman_numerals = {'I': 1, 'II': 2, 'III': 3, 'V': 5, 'X': 100}

In [2]: roman_numerals
Out[2]: {'I': 1, 'II': 2, 'III': 3, 'V': 5, 'X': 100}
```

Accessing the Value Associated with a Key

Let's get the value associated with the key 'V':

```
In [3]: roman_numerals['V']
Out[3]: 5
```

Updating the Value of an Existing Key–Value Pair

You can update a key's associated value in an assignment statement, which we do here to replace the incorrect value associated with the key 'X':

```
In [4]: roman_numerals['X'] = 10
In [5]: roman_numerals
Out[5]: {'I': 1, 'II': 2, 'III': 3, 'V': 5, 'X': 10}
```

Adding a New Key–Value Pair

Assigning a value to a nonexistent key inserts the key–value pair in the dictionary:

```
In [6]: roman_numerals['L'] = 50
In [7]: roman_numerals
Out[7]: {'I': 1, 'II': 2, 'III': 3, 'V': 5, 'X': 10, 'L': 50}
```

String keys are case sensitive. Assigning to a nonexistent key inserts a new key–value pair. This may be what you intend, or it could be a logic error.

Removing a Key–Value Pair

You can delete a key–value pair from a dictionary with the `del` statement:

```
In [8]: del roman_numerals['III']
In [9]: roman_numerals
Out[9]: {'I': 1, 'II': 2, 'V': 5, 'X': 10, 'L': 50}
```

You also can remove a key–value pair with the dictionary method `pop`, which returns the value for the removed key:

```
In [10]: roman_numerals.pop('X')
Out[10]: 10
In [11]: roman_numerals
Out[11]: {'I': 1, 'II': 2, 'V': 5, 'L': 50}
```

Attempting to Access a Nonexistent Key

Accessing a nonexistent key results in a `KeyError`:

```
In [12]: roman_numerals['III']
-----
KeyError Traceback (most recent call last)
<ipython-input-12-cccd50c7f0c8b> in <module>()
----> 1 roman_numerals['III']

KeyError: 'III'
```

You can prevent this error by using dictionary method `get`, which normally returns its argument's corresponding value. If that key is not found, `get` returns `None`. IPython does not display anything when `None` is returned in snippet [13]. If you specify a second argument to `get`, it returns that value if the key is not found:

```
In [13]: roman_numerals.get('III')
In [14]: roman_numerals.get('III', 'III not in dictionary')
Out[14]: 'III not in dictionary'
In [15]: roman_numerals.get('V')
Out[15]: 5
```

Testing Whether a Dictionary Contains a Specified Key

Operators `in` and `not in` can determine whether a dictionary contains a specified key:

```
In [16]: 'V' in roman_numerals
Out[16]: True

In [17]: 'III' in roman_numerals
Out[17]: False

In [18]: 'III' not in roman_numerals
Out[18]: True
```



Self Check

- 1** (*True/False*) Assigning to a nonexistent dictionary key causes an exception.

Answer: False. Assigning to a nonexistent key inserts a new key–value pair. This may be what you intend, or it could be a logic error if you incorrectly specify the key.

- 2** (*Fill-In*) What does an expression of the following form do when the *key* is in the dictionary?

$$\text{dictionaryName}[\text{key}] = \text{value}$$

Answer: It updates the *value* associated with the *key*, replacing the original *value*.

- 3** (*IPython Session*) String dictionary keys are case sensitive. Confirm this by using the following dictionary and assigning 10 to the key 'x'—doing so adds a new key–value pair rather than correcting the value for the key 'X':

```
roman_numerals = {'I': 1, 'II': 2, 'III': 3, 'V': 5, 'X': 100}
```

Answer:

```
In [1]: roman_numerals = {'I': 1, 'II': 2, 'III': 3, 'V': 5, 'X': 100}

In [2]: roman_numerals['x'] = 10

In [3]: roman_numerals
Out[3]: {'I': 1, 'II': 2, 'III': 3, 'V': 5, 'X': 100, 'x': 10}
```

6.2.4 Dictionary Methods **keys** and **values**

Earlier, we used dictionary method `items` to iterate through tuples of a dictionary's key–value pairs. Similarly, methods **keys** and **values** can be used to iterate through only a dictionary's keys or values, respectively:

```
In [1]: months = {'January': 1, 'February': 2, 'March': 3}

In [2]: for month_name in months.keys():
    ...:     print(month_name, end=' ')
    ...:
January February March

In [3]: for month_number in months.values():
    ...:     print(month_number, end=' ')
    ...:
1 2 3
```

Dictionary Views

Dictionary methods `items`, `keys` and `values` each return a view of a dictionary's data. When you iterate over a `view`, it "sees" the dictionary's current contents—it does *not* have its own copy of the data.

To show that views do *not* maintain their own copies of a dictionary's data, let's first save the view returned by `keys` into the variable `months_view`, then iterate through it:

```
In [4]: months_view = months.keys()
In [5]: for key in months_view:
...:     print(key, end=' ')
...:
January February March
```

Next, let's add a new key-value pair to `months` and display the updated dictionary:

```
In [6]: months['December'] = 12
In [7]: months
Out[7]: {'January': 1, 'February': 2, 'March': 3, 'December': 12}
```

Now, let's iterate through `months_view` again. The key we added above is indeed displayed:

```
In [8]: for key in months_view:
...:     print(key, end=' ')
...:
January February March December
```

Do not modify a dictionary while iterating through a view. According to Section 4.10.1 of the Python Standard Library documentation,¹ either you'll get a `RuntimeError` or the loop might not process all of the view's values.

Converting Dictionary Keys, Values and Key-Value Pairs to Lists

You might occasionally need *lists* of a dictionary's keys, values or key-value pairs. To obtain such a list, pass the view returned by `keys`, `values` or `items` to the built-in `list` function. Modifying these lists does *not* modify the corresponding dictionary:

```
In [9]: list(months.keys())
Out[9]: ['January', 'February', 'March', 'December']
In [10]: list(months.values())
Out[10]: [1, 2, 3, 12]
In [11]: list(months.items())
Out[11]: [('January', 1), ('February', 2), ('March', 3), ('December', 12)]
```

Processing Keys in Sorted Order

To process keys in *sorted* order, you can use built-in function `sorted` as follows:

```
In [12]: for month_name in sorted(months.keys()):
...:     print(month_name, end=' ')
...:
February December January March
```

1. <https://docs.python.org/3/library/stdtypes.html#dictionary-view-objects>.

✓ Self Check

1 (Fill-In) Dictionary method _____ returns an unordered list of the dictionary's keys.

Answer: keys.

2 (True/False) A view has its own copy of the corresponding data from the dictionary.
Answer: False. A view does *not* have its own copy of the corresponding data from the dictionary. As the dictionary changes, each view updates dynamically.

3 (IPython Session) For the following dictionary, create lists of its keys, values and items and show those lists.

```
roman_numerals = {'I': 1, 'II': 2, 'III': 3, 'V': 5}
```

Answer:

```
In [1]: roman_numerals = {'I': 1, 'II': 2, 'III': 3, 'V': 5}

In [2]: list(roman_numerals.keys())
Out[2]: ['I', 'II', 'III', 'V']

In [3]: list(roman_numerals.values())
Out[3]: [1, 2, 3, 5]

In [4]: list(roman_numerals.items())
Out[4]: [('I', 1), ('II', 2), ('III', 3), ('V', 5)]
```

6.2.5 Dictionary Comparisons

The comparison operators == and != can be used to determine whether two dictionaries have identical or different contents. An equals (==) comparison evaluates to True if both dictionaries have the same key–value pairs, *regardless* of the order in which those key–value pairs were added to each dictionary:

```
In [1]: country_capitals1 = {'Belgium': 'Brussels',
...: 'Haiti': 'Port-au-Prince'}
...:

In [2]: country_capitals2 = {'Nepal': 'Kathmandu',
...: 'Uruguay': 'Montevideo'}
...:

In [3]: country_capitals3 = {'Haiti': 'Port-au-Prince',
...: 'Belgium': 'Brussels'}
...:

In [4]: country_capitals1 == country_capitals2
Out[4]: False

In [5]: country_capitals1 == country_capitals3
Out[5]: True

In [6]: country_capitals1 != country_capitals2
Out[6]: True
```

✓ Self Check

I (*True/False*) The `==` comparison evaluates to `True` only if both dictionaries have the same key–value pairs in the same order.

Answer: False. The `==` comparison evaluates to `True` if both dictionaries have the same key–value pairs, regardless of their order.

6.2.6 Example: Dictionary of Student Grades

The script in Fig. 6.1 represents an instructor’s grade book as a dictionary that maps each student’s name (a string) to a list of integers containing that student’s grades on three exams. In each iteration of the loop that displays the data (lines 13–17), we unpack a key–value pair into the variables `name` and `grades` containing one student’s name and the corresponding list of three grades. Line 14 uses built-in function `sum` to total a given student’s grades, then line 15 calculates and displays that student’s average by dividing `total` by the number of grades for that student (`len(grades)`). Lines 16–17 keep track of the total of all four students’ grades and the number of grades for all the students, respectively. Line 19 prints the class average of all the students’ grades on all the exams.

```

1 # fig06_01.py
2 """Using a dictionary to represent an instructor's grade book."""
3 grade_book = {
4     'Susan': [92, 85, 100],
5     'Eduardo': [83, 95, 79],
6     'Azizi': [91, 89, 82],
7     'Pantipa': [97, 91, 92]
8 }
9
10 all_grades_total = 0
11 all_grades_count = 0
12
13 for name, grades in grade_book.items():
14     total = sum(grades)
15     print(f'Average for {name} is {total/len(grades):.2f}')
16     all_grades_total += total
17     all_grades_count += len(grades)
18
19 print(f"Class's average is: {all_grades_total / all_grades_count:.2f}")

```

```

Average for Susan is 92.33
Average for Eduardo is 85.67
Average for Azizi is 87.33
Average for Pantipa is 93.33
Class's average is: 89.67

```

Fig. 6.1 | Instructor’s gradebook dictionary.

6.2.7 Example: Word Counts²

The script in Fig. 6.2 builds a dictionary to count the number of occurrences of each word in a string. Lines 4–5 create a string `text` that we'll break into words—a process known as **tokenizing a string**. Python automatically concatenates strings separated by whitespace in parentheses. Line 7 creates an empty dictionary. The dictionary's keys will be the unique words, and its values will be integer counts of how many times each word appears in `text`.

```

1  # fig06_02.py
2  """Tokenizing a string and counting unique words."""
3
4  text = ('this is sample text with several words '
5         'this is more sample text with some different words')
6
7  word_counts = {}
8
9  # count occurrences of each unique word
10 for word in text.split():
11     if word in word_counts:
12         word_counts[word] += 1 # update existing key-value pair
13     else:
14         word_counts[word] = 1 # insert new key-value pair
15
16 print(f'{"WORD":<12}{COUNT}')
17
18 for word, count in sorted(word_counts.items()):
19     print(f'{word:<12}{count}')
20
21 print('\nNumber of unique words:', len(word_counts))

```

WORD	COUNT
different	1
is	2
more	1
sample	2
several	1
some	1
text	2
this	2
with	2
words	2
Number of unique words: 10	

Fig. 6.2 | Tokenizing a string and producing word counts.

Line 10 tokenizes `text` by calling string method `split`, which separates the words using the method's delimiter string argument. If you do not provide an argument, `split` uses a space. The method returns a list of tokens (that is, the words in `text`). Lines 10–14

- Techniques like word frequency counting are often used to analyze published works. For example, some people believe that the works of William Shakespeare actually might have been written by Sir Francis Bacon, Christopher Marlowe or others. Comparing the word frequencies of their works with those of Shakespeare can reveal writing-style similarities. We'll look at other document-analysis techniques in the "Natural Language Processing (NLP)" chapter.

iterate through the list of words. For each word, line 11 determines whether that word (the key) is already in the dictionary. If so, line 12 increments that word's count; otherwise, line 14 inserts a new key–value pair for that word with an initial count of 1.

Lines 16–21 summarize the results in a two-column table containing each word and its corresponding count. The for statement in lines 18 and 19 iterates through the dictionary's key–value pairs. It unpacks each key and value into the variables word and count, then displays them in two columns. Line 21 displays the number of unique words.

Python Standard Library Module `collections`

The Python Standard Library already contains the counting functionality that we implemented using the dictionary and the loop in lines 10–14. The module `collections` contains the type `Counter`, which receives an iterable and summarizes its elements. Let's reimplement the preceding script in fewer lines of code with Counter:

```
In [1]: from collections import Counter

In [2]: text = ('this is sample text with several words '
...:             'this is more sample text with some different words')
...:

In [3]: counter = Counter(text.split())

In [4]: for word, count in sorted(counter.items()):
...:     print(f'{word:<12}{count}')
...:
different    1
is           2
more         1
sample       2
several      1
some         1
text          2
this          2
with          2
words         2

In [5]: print('Number of unique keys:', len(counter.keys()))
Number of unique keys: 10
```

Snippet [3] creates the Counter, which summarizes the list of strings returned by `text.split()`. In snippet [4], Counter method `items` returns each string and its associated count as a tuple. We use built-in function `sorted` to get a list of these tuples in ascending order. By default `sorted` orders the tuples by their first elements. If those are identical, then it looks at the second element, and so on. The for statement iterates over the resulting sorted list, displaying each word and count in two columns.



Self Check

- 1 (*Fill-In*) String method _____ tokenizes a string using the delimiter provided in the method's string argument.

Answer: `split`.

- 2 (*IPython Session*) Use a comprehension to create a list of 50 random integers in the range 1–5. Summarize them with a Counter. Display the results in two-column format.

Answer:

```
In [1]: import random

In [2]: numbers = [random.randrange(1, 6) for i in range(50)]

In [3]: from collections import Counter

In [4]: counter = Counter(numbers)

In [5]: for value, count in sorted(counter.items()):
    ...:     print(f'{value:<4}{count}')
    ...:
1   9
2   6
3   13
4   10
5   12
```

6.2.8 Dictionary Method update

You may insert and update key–value pairs using dictionary method **update**. First, let's create an empty `country_codes` dictionary:

```
In [1]: country_codes = {}
```

The following `update` call receives a dictionary of key–value pairs to insert or update:

```
In [2]: country_codes.update({'South Africa': 'za'})
```

```
In [3]: country_codes
Out[3]: {'South Africa': 'za'}
```

Method `update` can convert keyword arguments into key–value pairs to insert. The following call automatically converts the parameter name `Australia` into the string key '`Australia`' and associates the value '`ar`' with that key:

```
In [4]: country_codes.update(Australia='ar')
```

```
In [5]: country_codes
Out[5]: {'South Africa': 'za', 'Australia': 'ar'}
```

Snippet [4] provided an incorrect country code for `Australia`. Let's correct this by using another keyword argument to update the value associated with '`Australia`':

```
In [6]: country_codes.update(Australia='au')
```

```
In [7]: country_codes
Out[7]: {'South Africa': 'za', 'Australia': 'au'}
```

Method `update` also can receive an iterable object containing key–value pairs, such as a list of two-element tuples.

6.2.9 Dictionary Comprehensions

Dictionary comprehensions provide a convenient notation for quickly generating dictionaries, often by mapping one dictionary to another. For example, in a dictionary with *unique* values, you can reverse the key–value pairs:

```
In [1]: months = {'January': 1, 'February': 2, 'March': 3}
In [2]: months2 = {number: name for name, number in months.items()}
In [3]: months2
Out[3]: {1: 'January', 2: 'February', 3: 'March'}
```

Curly braces delimit a *dictionary comprehension*, and the expression to the left of the `for` clause specifies a key–value pair of the form `key: value`. The comprehension iterates through `months.items()`, unpacking each key–value pair tuple into the variables `name` and `number`. The expression `number: name` reverses the key and value, so the new dictionary maps the month numbers to the month names.

What if `months` contained *duplicate* values? As these become the keys in `months2`, attempting to insert a *duplicate* key simply updates the existing key’s value. So if ‘February’ and ‘March’ both mapped to 2 originally, the preceding code would have produced

```
{1: 'January', 2: 'March'}
```

A dictionary comprehension also can map a dictionary’s values to new values. The following comprehension converts a dictionary of names and lists of grades into a dictionary of names and grade-point averages. The variables `k` and `v` commonly mean *key* and *value*:

```
In [4]: grades = {'Sue': [98, 87, 94], 'Bob': [84, 95, 91]}
In [5]: grades2 = {k: sum(v) / len(v) for k, v in grades.items()}
In [6]: grades2
Out[6]: {'Sue': 93.0, 'Bob': 90.0}
```

The comprehension unpacks each tuple returned by `grades.items()` into `k` (the name) and `v` (the list of grades). Then, the comprehension creates a new key–value pair with the key `k` and the value of `sum(v) / len(v)`, which averages the list’s elements.



Self Check

I (*IPython Session*) Use a dictionary comprehension to create a dictionary of the numbers 1–5 mapped to their cubes:

Answer:

```
In [1]: {number: number ** 3 for number in range(1, 6)}
Out[1]: {1: 1, 2: 8, 3: 27, 4: 64, 5: 125}
```

6.3 Sets

A set is an unordered collection of *unique* values. Sets may contain only immutable objects, like strings, ints, floats and tuples that contain only immutable elements. Though sets are iterable, they are not sequences and do not support indexing and slicing with square brackets, `[]`. Dictionaries also do not support slicing.

Creating a Set with Curly Braces

The following code creates a set of strings named `colors`:

```
In [1]: colors = {'red', 'orange', 'yellow', 'green', 'red', 'blue'}
In [2]: colors
Out[2]: {'blue', 'green', 'orange', 'red', 'yellow'}
```

Notice that the duplicate string 'red' was ignored (without causing an error). An important use of sets is **duplicate elimination**, which is automatic when creating a set. Also, the resulting set's values are *not* displayed in the same order as they were listed in snippet [1]. Though the color names are displayed in sorted order, sets are *unordered*. You should not write code that depends on the order of their elements.

Determining a Set's Length

You can determine the number of items in a set with the built-in `len` function:

```
In [3]: len(colors)
Out[3]: 5
```

Checking Whether a Value Is in a Set

You can check whether a set contains a particular value using the `in` and `not in` operators:

```
In [4]: 'red' in colors
Out[4]: True

In [5]: 'purple' in colors
Out[5]: False

In [6]: 'purple' not in colors
Out[6]: True
```

Iterating Through a Set

Sets are iterable, so you can process each set element with a `for` loop:

```
In [7]: for color in colors:
    ...:     print(color.upper(), end=' ')
    ...:
RED GREEN YELLOW BLUE ORANGE
```

Sets are *unordered*, so there's no significance to the iteration order.

Creating a Set with the Built-In `set` Function

You can create a set from another collection of values by using the built-in `set` function—here we create a list that contains several duplicate integer values and use that list as `set`'s argument:

```
In [8]: numbers = list(range(10)) + list(range(5))

In [9]: numbers
Out[9]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4]

In [10]: set(numbers)
Out[10]: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

If you need to create an empty set, you must use the `set` function with empty parentheses, rather than empty braces, {}, which represent an empty dictionary:

```
In [11]: set()
Out[11]: set()
```

Python displays an empty set as `set()` to avoid confusion with Python's string representation of an empty dictionary ({}).

Frozenset: An Immutable Set Type

Sets are *mutable*—you can add and remove elements, but set *elements* must be *immutable*. Therefore, a set cannot have other sets as elements. A **frozenset** is an *immutable* set—it cannot be modified after you create it, so a set *can* contain frozensets as elements. The built-in function **frozenset** creates a frozenset from any iterable.



Self Check

- 1** (*True/False*) Sets are collections of unique mutable and immutable objects.

Answer: False. Sets are collections of unique *immutable* objects.

- 2** (*Fill-In*) You can create a set from another collection of values by using the built-in _____ function.

Answer: set.

- 3** (*IPython Session*) Assign the following string to variable `text`, then split it into tokens with string method `split` and create a set from the results. Display the unique words in sorted order.

```
'to be or not to be that is the question'
```

Answer:

```
In [1]: text = 'to be or not to be that is the question'

In [2]: unique_words = set(text.split())

In [3]: for word in sorted(unique_words):
...:     print(word, end=' ')
...:
be is not or question that the to
```

6.3.1 Comparing Sets

Various operators and methods can be used to compare sets. The following sets contain the same values, so `==` returns `True` and `!=` returns `False`.

```
In [1]: {1, 3, 5} == {3, 5, 1}
Out[1]: True
```

```
In [2]: {1, 3, 5} != {3, 5, 1}
Out[2]: False
```

The `<` operator tests whether the set to its left is a **proper subset** of the one to its right—that is, all the elements in the left operand are in the right operand, and the sets are not equal:

```
In [3]: {1, 3, 5} < {3, 5, 1}
Out[3]: False
```

```
In [4]: {1, 3, 5} < {7, 3, 5, 1}
Out[4]: True
```

The `<=` operator tests whether the set to its left is an **improper subset** of the one to its right—that is, all the elements in the left operand are in the right operand, and the sets might be equal:

```
In [5]: {1, 3, 5} <= {3, 5, 1}
Out[5]: True
```

```
In [6]: {1, 3} <= {3, 5, 1}
Out[6]: True
```

You may also check for an improper subset with the set method `issubset`:

```
In [7]: {1, 3, 5}.issubset({3, 5, 1})
Out[7]: True
```

```
In [8]: {1, 2}.issubset({3, 5, 1})
Out[8]: False
```

The `>` operator tests whether the set to its left is a **proper superset** of the one to its right—that is, all the elements in the right operand are in the left operand, and the left operand has more elements:

```
In [9]: {1, 3, 5} > {3, 5, 1}
Out[9]: False
```

```
In [10]: {1, 3, 5, 7} > {3, 5, 1}
Out[10]: True
```

The `>=` operator tests whether the set to its left is an **improper superset** of the one to its right—that is, all the elements in the right operand are in the left operand, and the sets might be equal:

```
In [11]: {1, 3, 5} >= {3, 5, 1}
Out[11]: True
```

```
In [12]: {1, 3, 5} >= {3, 1}
Out[12]: True
```

```
In [13]: {1, 3} >= {3, 1, 7}
Out[13]: False
```

You may also check for an improper superset with the set method `issuperset`:

```
In [14]: {1, 3, 5}.issuperset({3, 5, 1})
Out[14]: True
```

```
In [15]: {1, 3, 5}.issuperset({3, 2})
Out[15]: False
```

The argument to `issubset` or `issuperset` can be *any* iterable. When either of these methods receives a non-set iterable argument, it first converts the iterable to a set, then performs the operation.



Self Check

- 1** (*True/False*) Sets may be compared with only the `==` and `!=` comparison operators.
Answer: False. All the comparison operators may be used to compare sets.

- 2** (*Fill-In*) A subset is a(n) _____ subset of another set if all the subset's elements are in the other set and the other set has more elements.

Answer: proper.

3 (IPython Session) Use sets and `issuperset` to determine whether the characters of the string 'abc def ghi jkl mno' are a superset of the characters in the string 'hi mom'.

Answer:

```
In [1]: set('abc def ghi jkl mno').issuperset('hi mom')
Out[1]: True
```

6.3.2 Mathematical Set Operations

This section presents the set type's mathematical operators `|`, `&`, `-` and `^` and the corresponding methods.

Union

The **union** of two sets is a set consisting of all the unique elements from both sets. You can calculate the union with the `| operator` or with the set type's **union** method:

```
In [1]: {1, 3, 5} | {2, 3, 4}
Out[1]: {1, 2, 3, 4, 5}
```

```
In [2]: {1, 3, 5}.union([20, 20, 3, 40, 40])
Out[2]: {1, 3, 5, 20, 40}
```

The operands of the binary set operators, like `|`, must both be sets. The corresponding set methods may receive any iterable object as an argument—we passed a list. When a mathematical set method receives a non-set iterable argument, it first converts the iterable to a set, then applies the mathematical operation. Again, though the new sets' string representations show the values in ascending order, you should not write code that depends on this.

Intersection

The **intersection** of two sets is a set consisting of all the unique elements that the two sets have in common. You can calculate the intersection with the `& operator` or with the set type's **intersection** method:

```
In [3]: {1, 3, 5} & {2, 3, 4}
Out[3]: {3}
```

```
In [4]: {1, 3, 5}.intersection([1, 2, 2, 3, 3, 4, 4])
Out[4]: {1, 3}
```

Difference

The **difference** between two sets is a set consisting of the elements in the left operand that are not in the right operand. You can calculate the difference with the `- operator` or with the set type's **difference** method:

```
In [5]: {1, 3, 5} - {2, 3, 4}
Out[5]: {1, 5}
```

```
In [6]: {1, 3, 5, 7}.difference([2, 2, 3, 3, 4, 4])
Out[6]: {1, 5, 7}
```

Symmetric Difference

The **symmetric difference** between two sets is a set consisting of the elements of both sets that are not in common with one another. You can calculate the symmetric difference with the `^ operator` or with the set type's **symmetric_difference** method:

```
In [7]: {1, 3, 5} ^ {2, 3, 4}
Out[7]: {1, 2, 4, 5}

In [8]: {1, 3, 5, 7}.symmetric_difference([2, 2, 3, 3, 4, 4])
Out[8]: {1, 2, 4, 5, 7}
```

Disjoint

Two sets are **disjoint** if they do not have any common elements. You can determine this with the set type's **isdisjoint** method:

```
In [9]: {1, 3, 5}.isdisjoint({2, 4, 6})
Out[9]: True

In [10]: {1, 3, 5}.isdisjoint({4, 6, 1})
Out[10]: False
```



Self Check

1 (Fill-In) Two sets are _____ if the sets do not have any common elements.

Answer: disjoint.

2 (IPython Session) Given the sets $\{10, 20, 30\}$ and $\{5, 10, 15, 20\}$, use the mathematical set operators to produce the following sets:

- $\{30\}$
- $\{5, 15, 30\}$
- $\{5, 10, 15, 20, 30\}$
- $\{10, 20\}$

Answer:

```
In [1]: {10, 20, 30} - {5, 10, 15, 20}
Out[1]: {30}

In [2]: {10, 20, 30} ^ {5, 10, 15, 20}
Out[2]: {5, 15, 30}

In [3]: {10, 20, 30} | {5, 10, 15, 20}
Out[3]: {5, 10, 15, 20, 30}

In [4]: {10, 20, 30} & {5, 10, 15, 20}
Out[4]: {10, 20}
```

6.3.3 Mutable Set Operators and Methods

The operators and methods presented in the preceding section each result in a *new* set. Here we discuss operators and methods that modify an *existing* set.

Mutable Mathematical Set Operations

Like operator `|`, **union augmented assignment** `|=` performs a set union operation, but `|=` modifies its left operand:

```
In [1]: numbers = {1, 3, 5}
In [2]: numbers |= {2, 3, 4}
In [3]: numbers
Out[3]: {1, 2, 3, 4, 5}
```

Similarly, the set type's **update** method performs a union operation modifying the set on which it's called—the argument can be any iterable:

```
In [4]: numbers.update(range(10))

In [5]: numbers
Out[5]: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

The other mutable set methods are:

- intersection augmented assignment &=
- difference augmented assignment -=
- symmetric difference augmented assignment ^=

and their corresponding methods with iterable arguments are:

- intersection_update
- difference_update
- symmetric_difference_update

Methods for Adding and Removing Elements

Set method **add** inserts its argument if the argument is *not* already in the set; otherwise, the set remains unchanged:

```
In [6]: numbers.add(17)

In [7]: numbers.add(3)

In [8]: numbers
Out[8]: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 17}
```

Set method **remove** removes its argument from the set—a **KeyError** occurs if the value is not in the set:

```
In [9]: numbers.remove(3)

In [10]: numbers
Out[10]: {0, 1, 2, 4, 5, 6, 7, 8, 9, 17}
```

Method **discard** also removes its argument from the set but does not cause an exception if the value is not in the set.

You also can remove an *arbitrary* set element and return it with **pop**, but sets are unordered, so you do not know which element will be returned:

```
In [11]: numbers.pop()
Out[11]: 0

In [12]: numbers
Out[12]: {1, 2, 4, 5, 6, 7, 8, 9, 17}
```

A **KeyError** occurs if the set is empty when you call **pop**.

Finally, method **clear** empties the set on which it's called:

```
In [13]: numbers.clear()

In [14]: numbers
Out[14]: set()
```



Self Check

1 (*True/False*) Set method `pop` returns the first element added to the set.

Answer: False. Set method `pop` returns an *arbitrary* set element.

2 (*Fill-In*) Set method _____ performs a union operation, modifying the set on which it's called.

Answer: `update`.

6.3.4 Set Comprehensions

Like dictionary comprehensions, you define set comprehensions in curly braces. Let's create a new set containing only the unique even values in the list `numbers`:

```
In [1]: numbers = [1, 2, 2, 3, 4, 5, 6, 6, 7, 8, 9, 10, 10]
```

```
In [2]: evens = {item for item in numbers if item % 2 == 0}
```

```
In [3]: evens
```

```
Out[3]: {2, 4, 6, 8, 10}
```

6.4 Intro to Data Science: Dynamic Visualizations

The preceding chapter's Intro to Data Science section introduced visualization. We simulated rolling a six-sided die and used the Seaborn and Matplotlib visualization libraries to create a publication-quality *static* bar plot showing the frequencies and percentages of each roll value. In this section, we make things "come alive" with *dynamic visualizations*.

The Law of Large Numbers

When we introduced random-number generation, we mentioned that if the `random` module's `randrange` function indeed produces integers at random, then every number in the specified range has an equal probability (or likelihood) of being chosen each time the function is called. For a six-sided die, each value 1 through 6 should occur one-sixth of the time, so the probability of any one of these values occurring is 1/6th or about 16.667%.

In the next section, we create and execute a *dynamic* (that is, *animated*) die-rolling simulation script. In general, you'll see that the more rolls we attempt, the closer each die value's percentage of the total rolls gets to 16.667% and the heights of the bars gradually become about the same. This is a manifestation of the *law of large numbers*.



Self Check

1 (*Fill-In*) As we toss a coin an increasing number of times, we expect the percentages of heads and tails to become closer to 50% each. This is a manifestation of _____.

Answer: the law of large numbers.

6.4.1 How Dynamic Visualization Works

The plots produced with Seaborn and Matplotlib in the previous chapter's Intro to Data Science section help you analyze the results for a fixed number of die rolls *after* the simulation completes. This section's enhances that code with the Matplotlib `animation` module's `FuncAnimation` function, which updates the bar plot *dynamically*. You'll see the bars, die frequencies and percentages "come alive," updating *continuously* as the rolls occur.

Animation Frames

`FuncAnimation` drives a **frame-by-frame animation**. Each **animation frame** specifies everything that should change during one plot update. Stringing together many of these updates over time creates the animation effect. You decide what each frame displays with a function you define and pass to `FuncAnimation`.

Each animation frame will:

- roll the dice a specified number of times (from 1 to as many as you'd like), updating die frequencies with each roll,
- clear the current plot,
- create a new set of bars representing the updated frequencies, and
- create new frequency and percentage text for each bar.

Generally, displaying more frames-per-second yields smoother animation. For example, video games with fast-moving elements try to display *at least* 30 frames-per-second and often more. Though you'll specify the number of milliseconds between animation frames, the actual number of frames-per-second can be affected by the amount of work you perform in each frame and the speed of your computer's processor. This example displays an animation frame every 33 milliseconds—yielding approximately 30 ($1000 / 33$) frames-per-second. Try larger and smaller values to see how they affect the animation. Experimentation is important in developing the best visualizations.

Running `RollDieDynamic.py`

In the previous chapter's Intro to Data Science section, we developed the static visualization *interactively* so you could see how the code updates the bar plot as you execute each statement. The actual bar plot with the final frequencies and percentages was drawn only once.

For this dynamic visualization, the screen results update frequently so that you can see the animation. Many things change continuously—the lengths of the bars, the frequencies and percentages above the bars, the spacing and labels on the axes and the total number of die rolls shown in the plot's title. For this reason, we present this visualization as a script, rather than interactively developing it.

The script takes two command-line arguments:

- `number_of_frames`—The number of animation frames to display. This value determines the total number of times that `FuncAnimation` updates the graph. For each animation frame, `FuncAnimation` calls a function that you define (in this example, `update`) to specify how to change the plot.
- `rolls_per_frame`—The number of times to roll the die in each animation frame. We'll use a loop to roll the die this number of times, summarize the results, then update the graph with bars and text representing the new frequencies.

To understand how we use these two values, consider the following command:

```
ipython RollDieDynamic.py 6000 1
```

In this case, `FuncAnimation` calls our `update` function 6000 times, rolling one die per frame for a total of 6000 rolls. This enables you to see the bars, frequencies and percentages

update one roll at a time. On our system, this animation took about 3.33 minutes (6000 frames / 30 frames-per-second / 60 seconds-per-minute) to show you only 6000 die rolls.

Displaying animation frames to the screen is a relatively slow *input–output-bound* operation compared to the die rolls, which occur at the computer’s super fast CPU speeds. If we roll only one die per animation frame, we won’t be able to run a large number of rolls in a reasonable amount of time. Also, for small numbers of rolls, you’re unlikely to see the die percentages converge on their expected 16.667% of the total rolls.

To see the law of large numbers in action, you can increase the execution speed by rolling the die more times per animation frame. Consider the following command:

```
ipython RollDieDynamic.py 10000 600
```

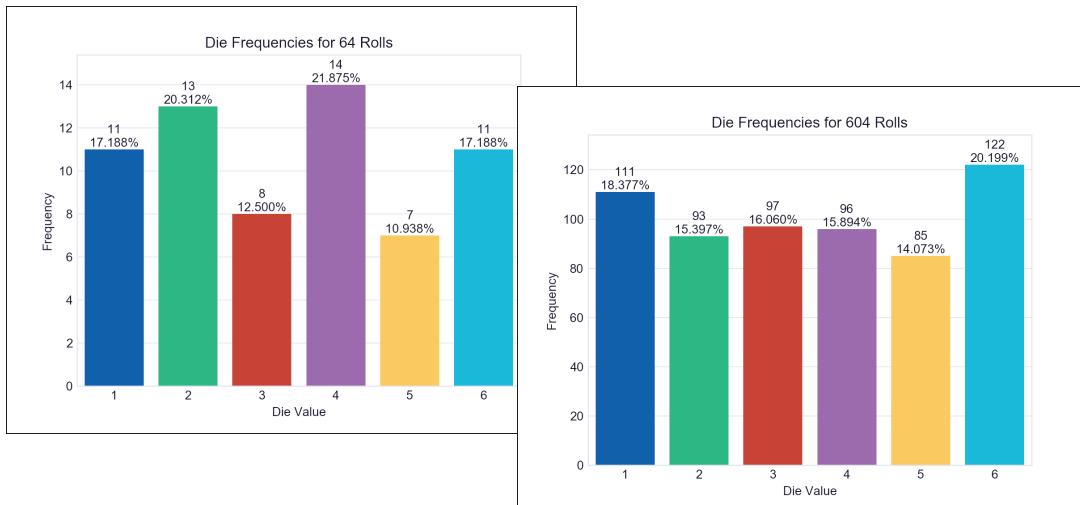
In this case, `FuncAnimation` will call our `update` function 10,000 times, performing 600 rolls-per-frame for a total of 6,000,000 rolls. On our system, this took about 5.55 minutes (10,000 frames / 30 frames-per-second / 60 seconds-per-minute), but displayed approximately 18,000 rolls-per-second (30 frames-per-second * 600 rolls-per-frame), so we could quickly see the frequencies and percentages converge on their expected values of about 1,000,000 rolls per face and 16.667% per face.

Experiment with the numbers of rolls and frames until you feel that the program is helping you visualize the results most effectively. It’s fun and informative to watch it run and to tweak it until you’re satisfied with the animation quality.

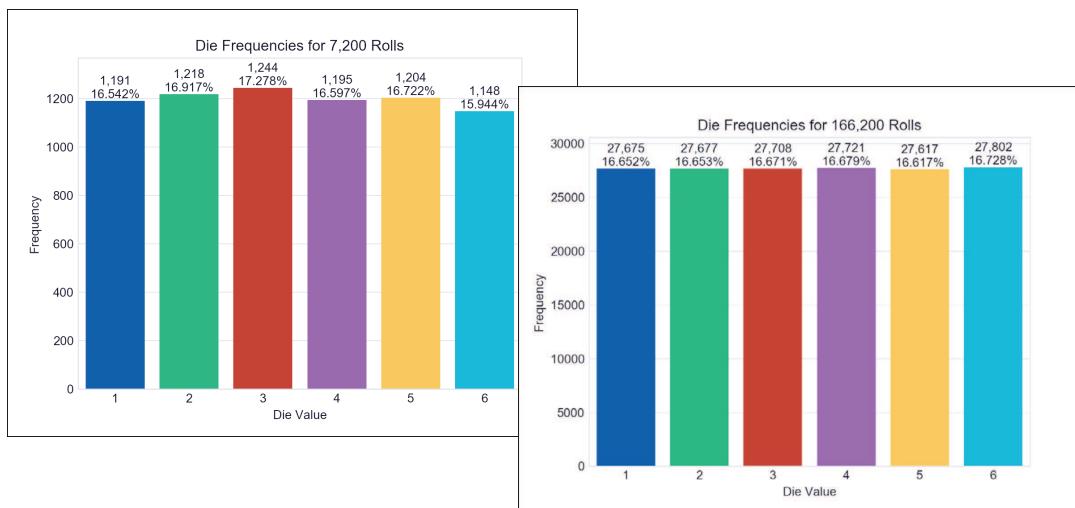
Sample Executions

We took the following four screen captures during each of two sample executions. In the first, the screens show the graph after just 64 die rolls, then again after 604 of the 6000 total die rolls. Run this script live to see over time how the bars update dynamically. In the second execution, the screen captures show the graph after 7200 die rolls and again after 166,200 out of the 6,000,000 rolls. With more rolls, you can see the percentages closing in on their expected values of 16.667% as predicted by the law of large numbers.

Execute 6000 animation frames rolling the die once per frame:
`ipython RollDieDynamic.py 6000 1`



Execute 10,000 animation frames rolling the die 600 times per frame:
`ipython RollDieDynamic.py 10000 600`



Self Check

- 1 (Fill-In) A(n) _____ specifies everything that should change during one plot update. Stringing together many of these over time creates the animation effect.

Answer: animation frame.

- 2 (True/False) Generally, displaying fewer frames-per-second yields smoother animation.

Answer: False. Generally, displaying *more* frames-per-second yields smoother animation.

- 3 (True/False) The actual number of frames-per-second is affected only by the millisecond interval between animation frames.

Answer: False. The actual number of frames-per-second also can be affected by the amount of work performed in each frame and the speed of your computer's processor.

6.4.2 Implementing a Dynamic Visualization

The script we present in this section uses the same Seaborn and Matplotlib features shown in the previous chapter's Intro to Data Science section. We reorganized the code for use with Matplotlib's *animation* capabilities.

Importing the Matplotlib animation Module

We focus primarily on the new features used in this example. Line 3 imports the Matplotlib animation module.

```

1 # RollDieDynamic.py
2 """Dynamically graphing frequencies of die rolls."""
3 from matplotlib import animation
4 import matplotlib.pyplot as plt
5 import random
6 import seaborn as sns
7 import sys
8

```

Function update

Lines 9–27 define the update function that FuncAnimation calls once per animation frame. This function must provide at least one argument. Lines 9–10 show the beginning of the function definition. The parameters are:

- `frame_number`—The next value from FuncAnimation’s `frames` argument, which we’ll discuss momentarily. Though FuncAnimation requires the update function to have this parameter, we do not use it in this update function.
- `rolls`—The number of die rolls per animation frame.
- `faces`—The die face values used as labels along the graph’s *x*-axis.
- `frequencies`—The list in which we summarize the die frequencies.

We discuss the rest of the function’s body in the next several subsections.

```
9  def update(frame_number, rolls, faces, frequencies):
10     """Configures bar plot contents for each animation frame."""
```

Function update: Rolling the Die and Updating the frequencies List

Lines 12–13 roll the die `rolls` times and increment the appropriate `frequencies` element for each roll. Note that we subtract 1 from the die value (1 through 6) before incrementing the corresponding `frequencies` element—as you’ll see, `frequencies` is a six-element list (defined in line 36), so its indices are 0 through 5.

```
11     # roll die and update frequencies
12     for i in range(rolls):
13         frequencies[random.randrange(1, 7) - 1] += 1
14
```

Function update: Configuring the Bar Plot and Text

Line 16 in function `update` calls the `matplotlib.pyplot` module’s `cla` (clear axes) function to remove the existing bar plot elements before drawing new ones for the current animation frame. We discussed the code in lines 17–27 in the previous chapter’s Intro to Data Science section. Lines 17–20 create the bars, set the bar plot’s title, set the *x*- and *y*-axis labels and scale the plot to make room for the frequency and percentage text above each bar. Lines 23–27 display the frequency and percentage text.

```
15     # reconfigure plot for updated die frequencies
16     plt.cla() # clear old contents contents of current Figure
17     axes = sns.barplot(faces, frequencies, palette='bright') # new bars
18     axes.set_title(f'Die Frequencies for {sum(frequencies):,} Rolls')
19     axes.set(xlabel='Die Value', ylabel='Frequency')
20     axes.set_ylim(top=max(frequencies) * 1.10) # scale y-axis by 10%
21
22     # display frequency & percentage above each patch (bar)
23     for bar, frequency in zip(axes.patches, frequencies):
24         text_x = bar.get_x() + bar.get_width() / 2.0
25         text_y = bar.get_height()
26         text = f'{frequency:,}\n{frequency / sum(frequencies):.3%}'
27         axes.text(text_x, text_y, text, ha='center', va='bottom')
28
```

Variables Used to Configure the Graph and Maintain State

Lines 30 and 31 use the `sys` module's `argv` list to get the script's command-line arguments. Line 33 specifies the Seaborn 'whitegrid' style. Line 34 calls the `matplotlib.pyplot` module's `Figure` function to get the `Figure` object in which `FuncAnimation` displays the animation. The function's argument is the window's title. As you'll soon see, this is one of `FuncAnimation`'s required arguments. Line 35 creates a list containing the die face values 1–6 to display on the plot's *x*-axis. Line 36 creates the six-element `frequencies` list with each element initialized to 0—we update this list's counts with each die roll.

```
29 # read command-line arguments for number of frames and rolls per frame
30 number_of_frames = int(sys.argv[1])
31 rolls_per_frame = int(sys.argv[2])
32
33 sns.set_style('whitegrid') # white background with gray grid lines
34 figure = plt.figure('Rolling a Six-Sided Die') # Figure for animation
35 values = list(range(1, 7)) # die faces for display on x-axis
36 frequencies = [0] * 6 # six-element list of die frequencies
37
```

Calling the animation Module's FuncAnimation Function

Lines 39–41 call the Matplotlib animation module's `FuncAnimation` function to update the bar chart dynamically. The function returns an object representing the animation. Though this is not used explicitly, you *must* store the reference to the animation; otherwise, Python immediately terminates the animation and returns its memory to the system.

```
38 # configure and start animation that calls function update
39 die_animation = animation.FuncAnimation(
40     figure, update, repeat=False, frames=number_of_frames, interval=33,
41     fargs=(rolls_per_frame, values, frequencies))
42
43 plt.show() # display window
```

`FuncAnimation` has two required arguments:

- `figure`—the `Figure` object in which to display the animation, and
- `update`—the function to call once per animation frame.

In this case, we also pass the following optional keyword arguments:

- `repeat`—`False` terminates the animation after the specified number of frames. If `True` (the default), when the animation completes it restarts from the beginning.
- `frames`—The total number of animation frames, which controls how many times `FuncAnimation` calls `update`. Passing an integer is equivalent to passing a range—for example, 600 means `range(600)`. `FuncAnimation` passes one value from this range as the first argument in each call to `update`.
- `interval`—The number of milliseconds (33, in this case) between animation frames (the default is 200). After each call to `update`, `FuncAnimation` waits 33 milliseconds before making the next call.
- `fargs` (short for “function arguments”)—A tuple of other arguments to pass to the function you specified in `FuncAnimation`'s second argument. The arguments you specify in the `fargs` tuple correspond to `update`'s parameters `rolls`, `faces` and `frequencies` (line 9).

For a list of FuncAnimation's other optional arguments, see

[https://matplotlib.org/api/_as_gen/
matplotlib.animation.FuncAnimation.html](https://matplotlib.org/api/_as_gen/matplotlib.animation.FuncAnimation.html)

Finally, line 43 displays the window.



Self Check

1 (*Fill-In*) The Matplotlib _____ module's _____ function dynamically updates a visualization.

Answer: animation, FuncAnimation.

2 (*Fill-In*) FuncAnimation's _____ keyword argument enables you to pass custom arguments to the function that's called once per animation frame.

Answer: fargs.

6.5 Wrap-Up

In this chapter, we discussed Python's dictionary and set collections. We said what a dictionary is and presented several examples. We showed the syntax of key–value pairs and showed how to use them to create dictionaries with comma-separated lists of key–value pairs in curly braces, {}. You also created dictionaries with dictionary comprehensions.

You used square brackets, [], to retrieve the value corresponding to a key, and to insert and update key–value pairs. You also used the dictionary method `update` to change a key's associated value. You iterated through a dictionary's keys, values and items.

You created sets of unique immutable values. You compared sets with the comparison operators, combined sets with set operators and methods, changed sets' values with the mutable set operations and created sets with set comprehensions. You saw that sets are mutable. Froszensets are immutable, so they can be used as set and frozenset elements.

In the Intro to Data Science section, we continued our visualization introduction by presenting the die-rolling simulation with a *dynamic* bar plot to make the law of large numbers "come alive." In addition, to the Seaborn and Matplotlib features shown in the previous chapter's Intro to Data Science section, we used Matplotlib's FuncAnimation function to control a frame-by-frame animation. FuncAnimation called a function we defined that specified what to display in each animation frame.

In the next chapter, we discuss array-oriented programming with the popular NumPy library. As you'll see, NumPy's `ndarray` collection can be up to two orders of magnitude faster than performing many of the same operations with Python's built-in lists. This power will come in handy for today's big data applications.

Exercises

Unless specified otherwise, use IPython sessions for each exercise.

6.1 (*Discussion: Dictionary Methods*) Briefly explain the operation of each of the following dictionary methods:

- add
- keys
- values
- items

6.2 (*What's Wrong with This Code?*) The following code should display the unique words in the string `text` and the number of occurrences of each word.

```
from collections import Counter
text = ('to be or not to be that is the question')
counter = Counter(text.split())
for word, count in sorted(counter):
    print(f'{word:<12}{count}')
```

6.3 (*What Does This Code Do?*) The dictionary `temperatures` contains three Fahrenheit temperature samples for each of four days. What does the `for` statement do?

```
temperatures = {
    'Monday': [66, 70, 74],
    'Tuesday': [50, 56, 64],
    'Wednesday': [75, 80, 83],
    'Thursday': [67, 74, 81]
}

for k, v in temperatures.items():
    print(f'{k}: {sum(v)/len(v):.2f}')
```

6.4 (*Fill in the Missing Code*) In each of the following expressions, replace the `***`s with a set operator that produces the result shown in the comment. The last operation should check whether the left operand is an improper subset of the right operand. For each of the first four expressions, specify the name of the set operation that produces the specified result.

- a) `{1, 2, 4, 8, 16} *** {1, 4, 16, 64, 256}` # `{1,2,4,8,16,64,256}`
- b) `{1, 2, 4, 8, 16} *** {1, 4, 16, 64, 256}` # `{1,4,16}`
- c) `{1, 2, 4, 8, 16} *** {1, 4, 16, 64, 256}` # `{2,8}`
- d) `{1, 2, 4, 8, 16} *** {1, 4, 16, 64, 256}` # `{2,8,64,256}`
- e) `{1, 2, 4, 8, 16} *** {1, 4, 16, 64, 256}` # `False`

6.5 (*Counting Duplicate Words*) Write a script that uses a dictionary to determine and print the number of *duplicate* words in a sentence. Treat uppercase and lowercase letters the same and assume there is no punctuation in the sentence. Use the techniques you learned in Section 6.2.7. Words with counts larger than 1 have duplicates.

6.6 (*Duplicate Word Removal*) Write a function that receives a list of words, then determines and displays in alphabetical order only the unique words. Treat uppercase and lowercase letters the same. The function should use a set to get the unique words in the list. Test your function with several sentences.

6.7 (*Character Counts*) Recall that strings are sequences of characters. Use techniques similar to Fig. 6.2 to write a script that inputs a sentence from the user, then uses a dictionary to summarize the number of occurrences of each letter. Ignore case, ignore blanks and assume the user does not enter any punctuation. Display a two-column table of the letters and their counts with the letters in sorted order. Challenge: Use a set operation to determine which letters of the alphabet were not in the original string.

6.8 (*Challenge: Writing the Word Equivalent of a Check Amount*) In check-writing systems, it's crucial to prevent alteration of check amounts. One common security method

requires that the amount be written in numbers and spelled out in words as well. Even if someone can alter the numerical amount of the check, it's tough to change the amount in words. Create a dictionary that maps numbers to their corresponding word equivalents. Write a script that inputs a numeric check amount that's less than 1000 and uses the dictionary to write the word equivalent of the amount. For example, the amount 112.43 should be written as

ONE HUNDRED TWELVE AND 43/100

- 6.9** (*Dictionary Manipulations*) Using the following dictionary, which maps country names to Internet top-level domains (TLDs):

```
tlds = {'Canada': 'ca', 'United States': 'us', 'Mexico': 'mx'}
```

perform the following tasks and display the results:

- Check whether the dictionary contains the key 'Canada'.
- Check whether the dictionary contains the key 'France'.
- Iterate through the key–value pairs and display them in two-column format.
- Add the key–value pair 'Sweden' and 'sw' (which is incorrect).
- Update the value for the key 'Sweden' to 'se'.
- Use a dictionary comprehension to reverse the keys and values.
- With the result of part (f), use a dictionary comprehension to convert the country names to all uppercase letters.

- 6.10** (*Set Manipulations*) Using the following sets:

```
{'red', 'green', 'blue'}
{'cyan', 'green', 'blue', 'magenta', 'red'}
```

display the results of:

- comparing the sets using each of the comparison operators.
- combining the sets using each of the mathematical set operators.

- 6.11** (*Analyzing the Game of Craps*) Modify the script of Fig. 4.2 to play 1,000,000 games of craps. Use a `wins` dictionary to keep track of the number of games won for a particular number of rolls. Similarly, use a `losses` dictionary to keep track of the number of games lost for a particular number of rolls. As the simulation proceeds, keep updating the dictionaries.

A typical key–value pair in the `wins` dictionary might be

4: 50217

indicating that 50217 games were won on the 4th roll. Display a summary of the results including:

- the percentage of the total games played that were won.
- the percentage of the total games played that were lost.
- the percentages of the total games played that were won or lost on a given roll (column 2 of the sample output).
- the *cumulative* percentage of the total games played that were won or lost up to and including a given number of rolls (column 3 of the sample output).

Your output should be similar to the one below.

Percentage of wins: 50.2%		
Percentage of losses: 49.8%		
Percentage of wins/losses based on total number of rolls		
Rolls	% Resolved on this roll	Cumulative % of games resolved
1	30.10%	30.10%
2	20.80%	50.90%
3	14.10%	65.00%
4	9.90%	74.90%
5	7.40%	82.30%
6	4.60%	86.90%
7	3.70%	90.60%
8	2.40%	93.00%
9	1.90%	94.90%
10	1.10%	96.00%
11	0.90%	96.90%
12	0.80%	97.70%
13	0.80%	98.50%
14	0.30%	98.80%
15	0.30%	99.10%
16	0.30%	99.40%
17	0.50%	99.90%
25	0.10%	100.00%

6.12 (Translation Dictionary) Use an online translation tool such as Bing Microsoft Translator or Google Translate to translate English words to another language. Create a translations dictionary that maps the English words to their translations. Display a two-column table of translations.

6.13 (Synonyms Dictionary) Use an online thesaurus to look up synonyms for five words, then create a synonyms dictionary that maps those words to lists containing three synonyms for each word. Display the dictionary’s contents as a key with an indented list of synonyms below it.

6.14 (Intro to Data Science: Dynamic Visualization of Coin Tossing) Modify your coin-tossing simulation from Exercise 5.31 to update the bar plot dynamically as you flip the coin. Use the techniques you learned in Section 6.4.2.

6.15 (Intro to Data Science: Dynamic Visualization of Rolling Two Dice) Modify your simulation of rolling two dice from Exercise 5.32 to update the bar plot dynamically as you roll the dice. Use the techniques you learned in Section 6.4.2.

6.16 (Intro to Data Science: Dynamic Visualization of the Dice Game of Craps) Reimplement your solution to Exercise 5.33, using the techniques you learned in Section 6.4.2 to create a dynamic bar chart showing the wins and losses on the first roll, second roll, third roll, etc.

6.17 (Project: Cooking with Healthier Ingredients) In the “Strings: A Deeper Look” chapter’s exercises, you’ll write a script that enables its user to enter ingredients from a cooking recipe, then recommends healthier replacements.³ In preparation for that exercise, create a dictionary that maps ingredients to lists of potential replacements. Some ingredient replacements are shown below:

3. Always consult a healthcare professional before making significant changes to your diet.

Ingredient	Substitution
1 cup sour cream	1 cup yogurt
1 cup milk	1/2 cup evaporated milk and 1/2 cup water
1 teaspoon lemon juice	1/2 teaspoon vinegar
1 cup sugar	1/2 cup honey, 1 cup molasses or 1/4 cup agave nectar
1 cup butter	1 cup margarine or yogurt
1 cup flour	1 cup rye or rice flour
1 cup mayonnaise	1 cup cottage cheese or 1/8 cup mayonnaise and 7/8 cup yogurt
1 egg	2 tablespoons cornstarch, arrowroot flour or potato starch or 2 egg whites or 1/2 of a large banana (mashed)
1 cup milk	1 cup soy milk
1 cup oil	1 cup applesauce

Your dictionary should take into consideration that replacements are not always one-for-one. For example, if a cake recipe calls for three eggs, it might reasonably use six egg whites instead. Research conversion data for measurements and ingredient substitutes online. Your dictionary should map the ingredients to lists of potential substitutes.

Array-Oriented Programming with NumPy

7



Objectives

In this chapter you'll:

- Learn what arrays are and how they differ from lists.
- Use the `numpy` module's high-performance `ndarrays`.
- Compare list and `ndarray` performance with the IPython `%timeit` magic.
- Use `ndarrays` to store and retrieve data efficiently.
- Create and initialize `ndarrays`.
- Refer to individual `ndarray` elements.
- Iterate through `ndarrays`.
- Create and manipulate multidimensional `ndarrays`.
- Perform common `ndarray` manipulations.
- Create and manipulate pandas one-dimensional `Series` and two-dimensional `DataFrames`.
- Customize `Series` and `DataFrame` indices.
- Calculate basic descriptive statistics for data in a `Series` and a `DataFrame`.
- Customize floating-point number precision in pandas output formatting.

Outline

7.1	Introduction	7.10	Indexing and Slicing
7.2	Creating arrays from Existing Data	7.11	Views: Shallow Copies
7.3	array Attributes	7.12	Deep Copies
7.4	Filling arrays with Specific Values	7.13	Reshaping and Transposing
7.5	Creating arrays from Ranges	7.14	Intro to Data Science: pandas Series and DataFrames
7.6	List vs. array Performance: Introducing %timeit	7.14.1	pandas Series
7.7	array Operators	7.14.2	DataFrames
7.8	NumPy Calculation Methods	7.15	Wrap-Up
7.9	Universal Functions	Exercises	

7.1 Introduction

The **NumPy (Numerical Python)** library first appeared in 2006 and is the preferred Python array implementation. It offers a high-performance, richly functional *n*-dimensional array type called **ndarray**, which from this point forward we'll refer to by its synonym, **array**. NumPy is one of the many open-source libraries that the Anaconda Python distribution installs. Operations on arrays are up to two orders of magnitude faster than those on lists. In a big-data world in which applications may do massive amounts of processing on vast amounts of array-based data, this performance advantage can be critical. According to `libraries.io`, over 450 Python libraries depend on NumPy. Many popular data science libraries such as Pandas, SciPy (Scientific Python) and Keras (for deep learning) are built on or depend on NumPy.

In this chapter, we explore `array`'s basic capabilities. Lists can have multiple dimensions. You generally process multi-dimensional lists with nested loops or list comprehensions with multiple `for` clauses. A strength of NumPy is “array-oriented programming,” which uses functional-style programming with *internal* iteration to make array manipulations concise and straightforward, eliminating the kinds of bugs that can occur with the *external* iteration of explicitly programmed loops.

In this chapter's Intro to Data Science section, we begin our multi-section introduction to the `pandas` library that you'll use in many of the data science case study chapters. Big data applications often need more flexible collections than NumPy's arrays—collections that support mixed data types, custom indexing, missing data, data that's not structured consistently and data that needs to be manipulated into forms appropriate for the databases and data analysis packages you use. We'll introduce `pandas` array-like one-dimensional `Series` and two-dimensional `DataFrames` and begin demonstrating their powerful capabilities. After reading this chapter, you'll be familiar with four array-like collections—lists, `arrays`, `Series` and `DataFrames`. We'll add a fifth—`tensors`—in the “Deep Learning” chapter.



Self Check

- I (Fill-In) The NumPy library provides the _____ data structure, which is typically much faster than lists.

Answer: `ndarray`.

7.2 Creating arrays from Existing Data

The NumPy documentation recommends importing the **numpy module** as `np` so that you can access its members with "`np.`":

```
In [1]: import numpy as np
```

The `numpy` module provides various functions for creating arrays. Here we use the `array` function, which receives as an argument an array or other collection of elements and returns a new array containing the argument's elements. Let's pass a list:

```
In [2]: numbers = np.array([2, 3, 5, 7, 11])
```

The `array` function copies its argument's contents into the array. Let's look at the type of object that function `array` returns and display its contents:

```
In [3]: type(numbers)
Out[3]: numpy.ndarray
```

```
In [4]: numbers
Out[4]: array([ 2,  3,  5,  7, 11])
```

Note that the `type` is `numpy.ndarray`, but all arrays are output as "array." When outputting an array, NumPy separates each value from the next with a comma and a space and *right-aligns* all the values using the same field width. It determines the field width based on the value that occupies the *largest* number of character positions. In this case, the value 11 occupies the two character positions, so all the values are formatted in two-character fields. That's why there's a leading space between the [and 2.

Multidimensional Arguments

The `array` function copies its argument's dimensions. Let's create an array from a two-row-by-three-column list:

```
In [5]: np.array([[1, 2, 3], [4, 5, 6]])
Out[5]:
array([[1, 2, 3],
       [4, 5, 6]])
```

NumPy auto-formats arrays, based on their number of dimensions, aligning the columns within each row.



Self Check

1 (*Fill-In*) Function `array` creates an array from _____.

Answer: an array or other collection of elements.

2 (*IPython Session*) Create a one-dimensional array from a list comprehension that produces the even integers from 2 through 20.

Answer:

```
In [1]: import numpy as np
```

```
In [2]: np.array([x for x in range(2, 21, 2)])
Out[2]: array([ 2,  4,  6,  8, 10, 12, 14, 16, 18, 20])
```

3 (*IPython Session*) Create a 2-by-5 array containing the even integers from 2 through 10 in the first row and the odd integers from 1 through 9 in the second row.

Answer:

```
In [3]: np.array([[2, 4, 6, 8, 10], [1, 3, 5, 7, 9]])
Out[3]:
array([[ 2,  4,  6,  8, 10],
       [ 1,  3,  5,  7,  9]])
```

7.3 array Attributes

An array object provides **attributes** that enable you to discover information about its structure and contents. In this section we'll use the following arrays:

```
In [1]: import numpy as np
In [2]: integers = np.array([[1, 2, 3], [4, 5, 6]])
In [3]: integers
Out[3]:
array([[1, 2, 3],
       [4, 5, 6]])
In [4]: floats = np.array([0.0, 0.1, 0.2, 0.3, 0.4])
In [5]: floats
Out[5]: array([ 0.,  0.1,  0.2,  0.3,  0.4])
```

NumPy does not display trailing 0s to the right of the decimal point in floating-point values.

Determining an array's Element Type

The `array` function determines an array's element type from its argument's elements. You can check the element type with an array's **`dtype`** attribute:

```
In [6]: integers.dtype
Out[6]: dtype('int64') # int32 on some platforms
In [7]: floats.dtype
Out[7]: dtype('float64')
```

As you'll see in the next section, various array-creation functions receive a `dtype` keyword argument so you can specify an array's element type.

For performance reasons, NumPy is written in the C programming language and uses C's data types. By default, NumPy stores integers as the NumPy type `int64` values—which correspond to 64-bit (8-byte) integers in C—and stores floating-point numbers as the NumPy type `float64` values—which correspond to 64-bit (8-byte) floating-point values in C. In our examples, most commonly you'll see the types `int64`, `float64`, `bool` (for Boolean) and `object` for non-numeric data (such as strings). The complete list of supported types is at <https://docs.scipy.org/doc/numpy/user/basics.types.html>.

Determining an array's Dimensions

The attribute **`ndim`** contains an array's number of dimensions and the attribute **`shape`** contains a *tuple* specifying an array's dimensions:

```
In [8]: integers.ndim
Out[8]: 2
In [9]: floats.ndim
Out[9]: 1
```

```
In [10]: integers.shape
Out[10]: (2, 3)

In [11]: floats.shape
Out[11]: (5,)
```

Here, `integers` has 2 rows and 3 columns (6 elements) and `floats` is one-dimensional, so snippet [11] shows a one-element tuple (indicated by the comma) containing `floats`' number of elements (5).

Determining an array's Number of Elements and Element Size

You can view an array's total number of elements with the attribute `size` and the number of bytes required to store each element with `itemsize`:

```
In [12]: integers.size
Out[12]: 6

In [13]: integers.itemsize # 4 if C compiler uses 32-bit ints
Out[13]: 8

In [14]: floats.size
Out[14]: 5

In [15]: floats.itemsize
Out[15]: 8
```

Note that `integers`' size is the product of the `shape` tuple's values—two rows of three elements each for a total of six elements. In each case, `itemsize` is 8 because `integers` contains `int64` values and `floats` contains `float64` values, which each occupy 8 bytes.

Iterating Through a Multidimensional array's Elements

You'll generally manipulate arrays using concise functional-style programming techniques. However, because arrays are *iterable*, you can use external iteration if you'd like:

```
In [16]: for row in integers:
    ...:     for column in row:
    ...:         print(column, end=' ')
    ...:     print()
    ...:
1 2 3
4 5 6
```

You can iterate through a multidimensional array as if it were one-dimensional by using its `flat` attribute:

```
In [17]: for i in integers.flat:
    ...:     print(i, end=' ')
    ...:
1 2 3 4 5 6
```



Self Check

I (*True/False*) By default, NumPy displays trailing 0s to the right of the decimal point in a floating-point value.

Answer: False. By default, NumPy does *not* display trailing 0s in the fractional part of a floating-point value

2 (IPython Session) For the two-dimensional array in the previous section’s Self Check, display the number of dimensions and shape of the array.

Answer:

```
In [1]: import numpy as np
In [2]: a = np.array([[2, 4, 6, 8, 10], [1, 3, 5, 7, 9]])
In [3]: a.ndim
Out[3]: 2
In [4]: a.shape
Out[4]: (2, 5)
```

7.4 Filling arrays with Specific Values

NumPy provides functions `zeros`, `ones` and `full` for creating arrays containing 0s, 1s or a specified value, respectively. By default, `zeros` and `ones` create arrays containing `float64` values. We’ll show how to customize the element type momentarily. The first argument to these functions must be an integer or a tuple of integers specifying the desired dimensions. For an integer, each function returns a one-dimensional array with the specified number of elements:

```
In [1]: import numpy as np
In [2]: np.zeros(5)
Out[2]: array([ 0.,  0.,  0.,  0.,  0.])
```

For a tuple of integers, these functions return a multidimensional array with the specified dimensions. You can specify the array’s element type with the `zeros` and `ones` function’s `dtype` keyword argument:

```
In [3]: np.ones((2, 4), dtype=int)
Out[3]:
array([[1, 1, 1, 1],
       [1, 1, 1, 1]])
```

The array returned by `full` contains elements with the second argument’s value and type:

```
In [4]: np.full((3, 5), 13)
Out[4]:
array([[13, 13, 13, 13, 13],
       [13, 13, 13, 13, 13],
       [13, 13, 13, 13, 13]])
```

7.5 Creating arrays from Ranges

NumPy provides optimized functions for creating arrays from ranges. We focus on simple evenly spaced integer and floating-point ranges, but NumPy also supports nonlinear ranges.¹

Creating Integer Ranges with `arange`

Let’s use NumPy’s `arange` function to create integer ranges—similar to using built-in function `range`. In each case, `arange` first determines the resulting array’s number of elements, allocates the memory, then stores the specified range of values in the array:

1. <https://docs.scipy.org/doc/numpy/reference/routines.array-creation.html>.

```
In [1]: import numpy as np  
  
In [2]: np.arange(5)  
Out[2]: array([0, 1, 2, 3, 4])  
  
In [3]: np.arange(5, 10)  
Out[3]: array([5, 6, 7, 8, 9])  
  
In [4]: np.arange(10, 1, -2)  
Out[4]: array([10, 8, 6, 4, 2])
```

Though you can create arrays by passing ranges as arguments, always use `arange` as it's optimized for arrays. Soon we'll show how to determine the execution time of various operations so you can compare their performance.

Creating Floating-Point Ranges with `linspace`

You can produce evenly spaced floating-point ranges with NumPy's `linspace` function. The function's first two arguments specify the starting and ending values in the range, and the ending value is *included* in the array. The optional keyword argument `num` specifies the number of evenly spaced values to produce—this argument's default value is 50:

```
In [5]: np.linspace(0.0, 1.0, num=5)  
Out[5]: array([ 0. ,  0.25,  0.5 ,  0.75,  1. ])
```

Reshaping an array

You also can create an array from a range of elements, then use array method `reshape` to transform the one-dimensional array into a multidimensional array. Let's create an array containing the values from 1 through 20, then reshape it into four rows by five columns:

```
In [6]: np.arange(1, 21).reshape(4, 5)  
Out[6]:  
array([[ 1,  2,  3,  4,  5],  
       [ 6,  7,  8,  9, 10],  
       [11, 12, 13, 14, 15],  
       [16, 17, 18, 19, 20]])
```

Note the *chained method calls* in the preceding snippet. First, `arange` produces an array containing the values 1–20. Then we call `reshape` on that array to get the 4-by-5 array that was displayed.

You can reshape any array, provided that the new shape has the *same* number of elements as the original. So a six-element one-dimensional array can become a 3-by-2 or 2-by-3 array, and vice versa, but attempting to reshape a 15-element array into a 4-by-4 array (16 elements) causes a `ValueError`.

Displaying Large arrays

When displaying an array, if there are 1000 items or more, NumPy drops the middle rows, columns or both from the output. The following snippets generate 100,000 elements. The first case shows all four rows but only the first and last three of the 25,000 columns. The notation `...` represents the missing data. The second case shows the first and last three of the 100 rows, and the first and last three of the 1000 columns:

```
In [7]: np.arange(1, 100001).reshape(4, 25000)
Out[7]:
array([[    1,     2,     3, ..., 24998, 24999, 25000],
       [25001, 25002, 25003, ..., 49998, 49999, 50000],
       [50001, 50002, 50003, ..., 74998, 74999, 75000],
       [75001, 75002, 75003, ..., 99998, 99999, 100000]])

In [8]: np.arange(1, 100001).reshape(100, 1000)
Out[8]:
array([[    1,     2,     3, ...,    998,    999,   1000],
       [ 1001, 1002, 1003, ..., 1998, 1999, 2000],
       [ 2001, 2002, 2003, ..., 2998, 2999, 3000],
       ...,
       [ 97001, 97002, 97003, ..., 97998, 97999, 98000],
       [ 98001, 98002, 98003, ..., 98998, 98999, 99000],
       [ 99001, 99002, 99003, ..., 99998, 99999, 100000]])
```



Self Check

- 1 (*Fill-In*) NumPy function _____ returns an ndarray containing evenly spaced floating-point values.

Answer: linspace.

- 2 (*IPython Session*) Use NumPy function arange to create an array of 20 even integers from 2 through 40, then reshape the result into a 4-by-5 array.

Answer:

```
In [1]: import numpy as np

In [2]: np.arange(2, 41, 2).reshape(4, 5)
Out[2]:
array([[ 2,  4,  6,  8, 10],
       [12, 14, 16, 18, 20],
       [22, 24, 26, 28, 30],
       [32, 34, 36, 38, 40]])
```

7.6 List vs. array Performance: Introducing %timeit

Most array operations execute *significantly* faster than corresponding list operations. To demonstrate, we'll use the IPython **%timeit** magic command, which times the *average* duration of operations. Note that the times displayed on your system may vary from what we show here.

Timing the Creation of a List Containing Results of 6,000,000 Die Rolls

We've demonstrated rolling a six-sided die 6,000,000 times. Here, let's use the random module's randrange function with a list comprehension to create a list of six million die rolls and time the operation using %timeit. Note that we used the line-continuation character (\) to split the statement in snippet [2] over two lines:

```
In [1]: import random

In [2]: %timeit rolls_list = \
...:     [random.randrange(1, 7) for i in range(0, 6_000_000)]
6.29 s ± 119 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

By default, `%timeit` executes a statement in a loop, and it runs the loop *seven* times. If you do not indicate the number of loops, `%timeit` chooses an appropriate value. In our testing, operations that on average took more than 500 milliseconds iterated only once, and operations that took fewer than 500 milliseconds iterated 10 times or more.

After executing the statement, `%timeit` displays the statement's *average* execution time, as well as the standard deviation of all the executions. On average, `%timeit` indicates that it took 6.29 seconds (s) to create the list with a standard deviation of 119 milliseconds (ms). In total, the preceding snippet took about 44 seconds to run the snippet seven times.

Timing the Creation of an array Containing Results of 6,000,000 Die Rolls

Now, let's use the `randint` function from the `numpy.random` module to create an array of 6,000,000 die rolls

```
In [3]: import numpy as np
```

```
In [4]: %timeit rolls_array = np.random.randint(1, 7, 6_000_000)
72.4 ms ± 635 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

On average, `%timeit` indicates that it took only 72.4 *milliseconds* with a standard deviation of 635 microseconds (μ s) to create the array. In total, the preceding snippet took just under half a second to execute on our computer—about 1/100th of the time snippet [2] took to execute. The operation is *two orders of magnitude faster* with array!

60,000,000 and 600,000,000 Die Rolls

Now, let's create an array of 60,000,000 die rolls:

```
In [5]: %timeit rolls_array = np.random.randint(1, 7, 60_000_000)
873 ms ± 29.4 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

On average, it took only 873 milliseconds to create the array.

Finally, let's do 600,000,000 million die rolls:

```
In [6]: %timeit rolls_array = np.random.randint(1, 7, 600_000_000)
10.1 s ± 232 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

It took about 10 seconds to create 600,000,000 elements with NumPy vs. about 6 seconds to create only 6,000,000 elements with a list comprehension.

Based on these timing studies, you can see clearly why arrays are preferred over lists for compute-intensive operations. In the data science case studies, we'll enter the performance-intensive worlds of big data and AI. We'll see how clever hardware, software, communications and algorithm designs combine to meet the often enormous computing challenges of today's applications.

Customizing the `%timeit` Iterations

The number of iterations within each `%timeit` loop and the number of loops are customizable with the `-n` and `-r` options. The following executes snippet [4]'s statement three times per loop and runs the loop twice:²

```
In [7]: %timeit -n3 -r2 rolls_array = np.random.randint(1, 7, 6_000_000)
85.5 ms ± 5.32 ms per loop (mean ± std. dev. of 2 runs, 3 loops each)
```

2. For most readers, using `%timeit`'s default settings should be fine.

Other IPython Magics

IPython provides dozens of magics for a variety of tasks—for a complete list, see the IPython magics documentation.³ Here are a few helpful ones:

- **%load** to read code into IPython from a local file or URL.
- **%save** to save snippets to a file.
- **%run** to execute a .py file from IPython.
- **%precision** to change the default floating-point precision for IPython outputs.
- **%cd** to change directories without having to exit IPython first.
- **%edit** to launch an external editor—handy if you need to modify more complex snippets.
- **%history** to view a list of all snippets and commands you've executed in the current IPython session.



Self Check

I (IPython Session) Use `%timeit` to compare the execution time of the following two statements. The first uses a list comprehension to create a list of the integers from 0 to 9,999,999, then totals them with the built-in `sum` function. The second statement does the same thing using an array and its `sum` method.

```
sum([x for x in range(10_000_000)])
np.arange(10_000_000).sum()
```

Answer:

```
In [1]: import numpy as np

In [2]: %timeit sum([x for x in range(10_000_000)])
708 ms ± 28.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

In [3]: %timeit np.arange(10_000_000).sum()
27.2 ms ± 676 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

The statement with the list comprehension took 26 times longer to execute than the one with the array.

7.7 array Operators

NumPy provides many operators which enable you to write simple expressions that perform operations on entire arrays. Here, we demonstrate arithmetic between arrays and numeric values and between arrays of the same shape.

Arithmetic Operations with arrays and Individual Numeric Values

First, let's perform *element-wise arithmetic* with arrays and numeric values by using arithmetic operators and augmented assignments. Element-wise operations are applied to every element, so snippet [4] multiplies every element by 2 and snippet [5] cubes every element. Each returns a *new array* containing the result:

3. <http://ipython.readthedocs.io/en/stable/interactive/magics.html>.

```
In [1]: import numpy as np

In [2]: numbers = np.arange(1, 6)

In [3]: numbers
Out[3]: array([1, 2, 3, 4, 5])

In [4]: numbers * 2
Out[4]: array([ 2,  4,  6,  8, 10])

In [5]: numbers ** 3
Out[5]: array([ 1,   8,  27,  64, 125])

In [6]: numbers # numbers is unchanged by the arithmetic operators
Out[6]: array([1, 2, 3, 4, 5])
```

Snippet [6] shows that the arithmetic operators did not modify `numbers`. Operators `+` and `*` are *commutative*, so snippet [4] could also be written as `2 * numbers`.

Augmented assignments *modify* every element in the left operand.

```
In [7]: numbers += 10

In [8]: numbers
Out[8]: array([11, 12, 13, 14, 15])
```

Broadcasting

Normally, the arithmetic operations require as operands two arrays of the *same size and shape*. When one operand is a single value, called a **scalar**, NumPy performs the element-wise calculations as if the scalar were an array of the same shape as the other operand, but with the scalar value in all its elements. This is called **broadcasting**. Snippets [4], [5] and [7] each use this capability. For example, snippet [4] is equivalent to:

```
numbers * [2, 2, 2, 2, 2]
```

Broadcasting also can be applied between arrays of different sizes and shapes, enabling some concise and powerful manipulations. We'll show more examples of broadcasting later in the chapter when we introduce NumPy's universal functions.

Arithmetic Operations Between arrays

You may perform arithmetic operations and augmented assignments between arrays of the *same shape*. Let's multiply the one-dimensional arrays `numbers` and `numbers2` (created below) that each contain five elements:

```
In [9]: numbers2 = np.linspace(1.1, 5.5, 5)

In [10]: numbers2
Out[10]: array([ 1.1,  2.2,  3.3,  4.4,  5.5])

In [11]: numbers * numbers2
Out[11]: array([ 12.1,  26.4,  42.9,  61.6,  82.5])
```

The result is a new array formed by multiplying the arrays *element-wise* in each operand— $11 * 1.1, 12 * 2.2, 13 * 3.3$, etc. Arithmetic between arrays of integers and floating-point numbers results in an array of floating-point numbers.

Comparing arrays

You can compare arrays with individual values and with other arrays. Comparisons are performed *element-wise*. Such comparisons produce arrays of Boolean values in which each element's True or False value indicates the comparison result:

```
In [12]: numbers
Out[12]: array([11, 12, 13, 14, 15])

In [13]: numbers >= 13
Out[13]: array([False, False, True, True, True])

In [14]: numbers2
Out[14]: array([ 1.1,  2.2,  3.3,  4.4,  5.5])

In [15]: numbers2 < numbers
Out[15]: array([ True,  True,  True,  True,  True])

In [16]: numbers == numbers2
Out[16]: array([False, False, False, False, False])

In [17]: numbers == numbers
Out[17]: array([ True,  True,  True,  True,  True])
```

Snippet [13] uses broadcasting to determine whether each element of `numbers` is greater than or equal to 13. The remaining snippets compare the corresponding elements of each array operand.



Self Check

- 1** (*True/False*) When one of the operands of an array operator is a scalar, NumPy uses broadcasting to perform the calculation as if the scalar were an array of the same shape as the other operand, but containing the scalar value in all its elements.

Answer: True.

- 2** (*IPython Session*) Create an array of the values from 1 through 5, then use broadcasting to square each value.

Answer:

```
In [1]: import numpy as np

In [2]: np.arange(1, 6) ** 2
Out[2]: array([ 1,  4,  9, 16, 25])
```

7.8 NumPy Calculation Methods

An array has various methods that perform calculations using its contents. By default, these methods ignore the array's shape and use *all* the elements in the calculations. For example, calculating the mean of an array totals all of its elements regardless of its shape, then divides by the total number of elements. You can perform these calculations on each dimension as well. For example, in a two-dimensional array, you can calculate each row's mean and each column's mean.

Consider an array representing four students' grades on three exams:

```
In [1]: import numpy as np

In [2]: grades = np.array([[87, 96, 70], [100, 87, 90],
...:                      [94, 77, 90], [100, 81, 82]])
...:

In [3]: grades
Out[3]:
array([[ 87,  96,  70],
       [100,  87,  90],
       [ 94,  77,  90],
       [100,  81,  82]])
```

We can use methods to calculate `sum`, `min`, `max`, `mean`, `std` (standard deviation) and `var` (variance)—each is a functional-style programming *reduction*:

```
In [4]: grades.sum()
Out[4]: 1054

In [5]: grades.min()
Out[5]: 70

In [6]: grades.max()
Out[6]: 100

In [7]: grades.mean()
Out[7]: 87.83333333333333

In [8]: grades.std()
Out[8]: 8.792357792739987

In [9]: grades.var()
Out[9]: 77.30555555555555
```

Calculations by Row or Column

Many calculation methods can be performed on specific array dimensions, known as the array's *axes*. These methods receive an `axis` keyword argument that specifies which dimension to use in the calculation, giving you a quick way to perform calculations by row or column in a two-dimensional array.

Assume that you want to calculate the average grade on each *exam*, represented by the columns of `grades`. Specifying `axis=0` performs the calculation on all the *row* values within each column:

```
In [10]: grades.mean(axis=0)
Out[10]: array([95.25, 85.25, 83. ])
```

So 95.25 above is the average of the first column's grades (87, 100, 94 and 100), 85.25 is the average of the second column's grades (96, 87, 77 and 81) and 83 is the average of the third column's grades (70, 90, 90 and 82). Again, NumPy does *not* display trailing 0s to the right of the decimal point in '83.'. Also note that it *does* display all element values in the same field width, which is why '83.' is followed by two spaces.

Similarly, specifying `axis=1` performs the calculation on all the *column* values within each individual row. To calculate each student's average grade for all exams, we can use:

```
In [11]: grades.mean(axis=1)
Out[11]: array([84.33333333, 92.33333333, 87.           , 87.66666667])
```

This produces four averages—one each for the values in each row. So 84.3333333 is the average of row 0's grades (87, 96 and 70), and the other averages are for the remaining rows.

NumPy arrays have many more calculation methods. For the complete list, see

<https://docs.scipy.org/doc/numpy/reference/arrays.ndarray.html>



Self Check

- 1 (*Fill-In*) NumPy functions _____ and _____ calculate variance and standard deviation, respectively.

Answer: var, std.

- 2 (*IPython Session*) Use NumPy random-number generation to create an array of twelve random grades in the range 60 through 100, then reshape the result into a 3-by-4 array. Calculate the average of all the grades, the averages of the grades in each column and the averages of the grades in each row.

Answer:

```
In [1]: import numpy as np

In [2]: grades = np.random.randint(60, 101, 12).reshape(3, 4)

In [3]: grades
Out[3]:
array([[94, 72, 76, 91],
       [65, 78, 66, 70],
       [65, 60, 63, 72]])

In [4]: grades.mean()
Out[4]: 72.66666666666667

In [5]: grades.mean(axis=0)
Out[5]: array([74.66666667, 70.           , 68.33333333, 77.66666667])

In [6]: grades.mean(axis=1)
Out[6]: array([83.25, 69.75, 65.   ])
```

7.9 Universal Functions

NumPy offers dozens of standalone **universal functions** (or **ufuncs**) that perform various element-wise operations. Each performs its task using one or two array or array-like (such as lists) arguments. Some of these functions are called when you use operators like + and * on arrays. Each returns a new array containing the results.

Let's create an array and calculate the square root of its values, using the **sqrt universal function**:

```
In [1]: import numpy as np

In [2]: numbers = np.array([1, 4, 9, 16, 25, 36])

In [3]: np.sqrt(numbers)
Out[3]: array([1., 2., 3., 4., 5., 6.])
```

Let's add two arrays with the same shape, using the **add universal function**:

```
In [4]: numbers2 = np.arange(1, 7) * 10
```

```
In [5]: numbers2
```

```
Out[5]: array([10, 20, 30, 40, 50, 60])
```

```
In [6]: np.add(numbers, numbers2)
```

```
Out[6]: array([11, 24, 39, 56, 75, 96])
```

The expression `np.add(numbers, numbers2)` is equivalent to:

```
numbers + numbers2
```

Broadcasting with Universal Functions

Let's use the **multiply universal function** to multiply every element of `numbers2` by the scalar value 5:

```
In [7]: np.multiply(numbers2, 5)
```

```
Out[7]: array([ 50, 100, 150, 200, 250, 300])
```

The expression `np.multiply(numbers2, 5)` is equivalent to:

```
numbers2 * 5
```

Let's reshape `numbers2` into a 2-by-3 array, then multiply its values by a one-dimensional array of three elements:

```
In [8]: numbers3 = numbers2.reshape(2, 3)
```

```
In [9]: numbers3
```

```
Out[9]:
```

```
array([[10, 20, 30],
       [40, 50, 60]])
```

```
In [10]: numbers4 = np.array([2, 4, 6])
```

```
In [11]: np.multiply(numbers3, numbers4)
```

```
Out[11]:
```

```
array([[ 20,  80, 180],
       [ 80, 200, 360]])
```

This works because `numbers4` has the same length as each row of `numbers3`, so NumPy can apply the `multiply` operation by treating `numbers4` as if it were the following array:

```
array([[2, 4, 6],
       [2, 4, 6]])
```

If a universal function receives two arrays of different shapes that do not support broadcasting, a `ValueError` occurs. You can view the broadcasting rules at:

<https://docs.scipy.org/doc/numpy/user/basics.broadcasting.html>

Other Universal Functions

The NumPy documentation lists universal functions in five categories—math, trigonometry, bit manipulation, comparison and floating point. The following table lists some functions from each category. You can view the complete list, their descriptions and more information about universal functions at:

<https://docs.scipy.org/doc/numpy/reference/ufuncs.html>

NumPy universal functions

Math—add, subtract, multiply, divide, remainder, exp, log, sqrt, power, and more.

Trigonometry—sin, cos, tan, hypot, arcsin, arccos, arctan, and more.

Bit manipulation—bitwise_and, bitwise_or, bitwise_xor, invert, left_shift and right_shift.

Comparison—greater, greater_equal, less, less_equal, equal, not_equal, logical_and, logical_or, logical_xor, logical_not, minimum, maximum, and more.

Floating point—floor, ceil, isnan, fabs, trunc, and more.



Self Check

- 1 (*Fill-In*) NumPy offers dozens of standalone functions, which it calls _____.
Answer: universal functions (or ufuncs).

- 2 (*IPython Session*) Create an array of the values from 1 through 5, then use the power universal function and broadcasting to cube each value.

Answer:

```
In [1]: import numpy as np
In [2]: numbers = np.arange(1, 6)
In [3]: np.power(numbers, 3)
Out[3]: array([ 1,   8,  27,  64, 125])
```

7.10 Indexing and Slicing

One-dimensional arrays can be indexed and sliced using the same syntax and techniques we demonstrated in the “Sequences: Lists and Tuples” chapter. Here, we focus on array-specific indexing and slicing capabilities.

Indexing with Two-Dimensional arrays

To select an element in a two-dimensional array, specify a tuple containing the element’s row and column indices in square brackets (as in snippet [4]):

```
In [1]: import numpy as np
In [2]: grades = np.array([[87, 96, 70], [100, 87, 90],
...:                      [94, 77, 90], [100, 81, 82]])
...:
In [3]: grades
Out[3]:
array([[ 87,  96,  70],
       [100,  87,  90],
       [ 94,  77,  90],
       [100,  81,  82]])
In [4]: grades[0, 1] # row 0, column 1
Out[4]: 96
```

Selecting a Subset of a Two-Dimensional array's Rows

To select a single row, specify only one index in square brackets:

```
In [5]: grades[1]
Out[5]: array([100, 87, 90])
```

To select multiple sequential rows, use slice notation:

```
In [6]: grades[0:2]
Out[6]:
array([[ 87, 96, 70],
       [100, 87, 90]])
```

To select multiple non-sequential rows, use a list of row indices:

```
In [7]: grades[[1, 3]]
Out[7]:
array([[100, 87, 90],
       [100, 81, 82]])
```

Selecting a Subset of a Two-Dimensional array's Columns

You can select subsets of the columns by providing a tuple specifying the row(s) and column(s) to select. Each can be a specific index, a slice or a list. Let's select only the elements in the first column:

```
In [8]: grades[:, 0]
Out[8]: array([ 87, 100, 94, 100])
```

The 0 after the comma indicates that we're selecting only column 0. The : before the comma indicates which rows within that column to select. In this case, : is a *slice* representing *all* rows. This also could be a specific row number, a slice representing a subset of the rows or a list of specific row indices to select, as in snippets [5]–[7].

You can select consecutive columns using a slice:

```
In [9]: grades[:, 1:3]
Out[9]:
array([[96, 70],
       [87, 90],
       [77, 90],
       [81, 82]])
```

or specific columns using a *list* of column indices:

```
In [10]: grades[:, [0, 2]]
Out[10]:
array([[ 87, 70],
       [100, 90],
       [ 94, 90],
       [100, 82]])
```



Self Check

- I (*IPython Session*) Given the following array:

```
array([[ 1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10],
       [11, 12, 13, 14, 15]])
```

write statements to perform the following tasks:

- a) Select the second row.
- b) Select the first and third rows.
- c) Select the middle three columns.

Answer:

```
In [1]: import numpy as np
In [2]: a = np.arange(1, 16).reshape(3, 5)
In [3]: a
Out[3]:
array([[ 1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10],
       [11, 12, 13, 14, 15]])
In [4]: a[1]
Out[4]: array([ 6,  7,  8,  9, 10])
In [5]: a[[0, 2]]
Out[5]:
array([[ 1,  2,  3,  4,  5],
       [11, 12, 13, 14, 15]])
In [6]: a[:, 1:4]
Out[6]:
array([[ 2,  3,  4],
       [ 7,  8,  9],
       [12, 13, 14]])
```

7.11 Views: Shallow Copies

The previous chapter introduced *view objects*—that is, objects that “see” the data in other objects, rather than having their own copies of the data. Views are also known as **shallow copies**. Various array methods and slicing operations produce views of an array’s data.

The array method `view` returns a *new* array object with a *view* of the original array object’s data. First, let’s create an array and a view of that array:

```
In [1]: import numpy as np
In [2]: numbers = np.arange(1, 6)
In [3]: numbers
Out[3]: array([1, 2, 3, 4, 5])
In [4]: numbers2 = numbers.view()
In [5]: numbers2
Out[5]: array([1, 2, 3, 4, 5])
```

We can use the built-in `id` function to see that `numbers` and `numbers2` are *different* objects:

```
In [6]: id(numbers)
Out[6]: 4462958592
In [7]: id(numbers2)
Out[7]: 4590846240
```

To prove that `numbers2` views the *same* data as `numbers`, let's modify an element in `numbers`, then display both arrays:

```
In [8]: numbers[1] *= 10
In [9]: numbers2
Out[9]: array([ 1, 20,  3,  4,  5])
In [10]: numbers
Out[10]: array([ 1, 20,  3,  4,  5])
```

Similarly, changing a value in the view also changes that value in the original array:

```
In [11]: numbers2[1] /= 10
In [12]: numbers
Out[12]: array([1, 2, 3, 4, 5])
In [13]: numbers2
Out[13]: array([1, 2, 3, 4, 5])
```

Slice Views

Slices also create views. Let's make `numbers2` a slice that views only the first three elements of `numbers`:

```
In [14]: numbers2 = numbers[0:3]
In [15]: numbers2
Out[15]: array([1, 2, 3])
```

Again, we can confirm that `numbers` and `numbers2` are different objects with `id`:

```
In [16]: id(numbers)
Out[16]: 4462958592
In [17]: id(numbers2)
Out[17]: 4590848000
```

We can confirm that `numbers2` is a view of *only* the first *three* `numbers` elements by attempting to access `numbers2[3]`, which produces an `IndexError`:

```
In [18]: numbers2[3]
-----
IndexError                                 Traceback (most recent call last)
<ipython-input-16-582053f52daa> in <module>()
      1 numbers2[3]
-----
IndexError: index 3 is out of bounds for axis 0 with size 3
```

Now, let's modify an element both arrays share, then display them. Again, we see that `numbers2` is a view of `numbers`:

```
In [19]: numbers[1] *= 20
In [20]: numbers
Out[20]: array([1, 2, 3, 4, 5])
In [21]: numbers2
Out[21]: array([ 1, 40,  3])
```

✓ Self Check

1 (Fill-In) A view is also known as a(n) _____.

Answer: shallow copy.

7.12 Deep Copies

Though views are *separate* array objects, they save memory by sharing element data from other arrays. However, when sharing *mutable* values, sometimes it's necessary to create a **deep copy** with *independent* copies of the original data. This is especially important in multi-core programming, where separate parts of your program could attempt to modify your data at the same time, possibly corrupting it.

The **array method copy** returns a new array object with a *deep copy* of the original array object's data. First, let's create an array and a deep copy of that array:

```
In [1]: import numpy as np
In [2]: numbers = np.arange(1, 6)
In [3]: numbers
Out[3]: array([1, 2, 3, 4, 5])
In [4]: numbers2 = numbers.copy()
In [5]: numbers2
Out[5]: array([1, 2, 3, 4, 5])
```

To prove that `numbers2` has a separate copy of the data in `numbers`, let's modify an element in `numbers`, then display both arrays:

```
In [6]: numbers[1] *= 10
In [7]: numbers
Out[7]: array([ 1, 20,  3,  4,  5])
In [8]: numbers2
Out[8]: array([ 1,  2,  3,  4,  5])
```

As you can see, the change appears only in `numbers`.

Module copy—Shallow vs. Deep Copies for Other Types of Python Objects

In previous chapters, we covered *shallow copying*. In this chapter, we've covered how to *deep copy* array objects using their `copy` method. If you need deep copies of other types of Python objects, pass them to the `copy` module's **deepcopy** function.

✓ Self Check

1 (True/False) The `array method copy` returns a new array that is a view (shallow copy) of the original array.

Answer: False. The `array method copy` produces a *deep copy* of the original array.

2 (True/False) Module `copy` provides function `deep_copy`, which returns a deep copy of its argument.

Answer: False. The name of the function is `deepcopy`.

7.13 Reshaping and Transposing

We've used array method `reshape` to produce two-dimensional arrays from one-dimensional ranges. NumPy provides various other ways to reshape arrays.

`reshape` vs. `resize`

The array methods `reshape` and `resize` both enable you to change an array's dimensions. Method `reshape` returns a *view* (shallow copy) of the original array with the new dimensions. It does *not* modify the original array:

```
In [1]: import numpy as np
In [2]: grades = np.array([[87, 96, 70], [100, 87, 90]])
In [3]: grades
Out[3]:
array([[ 87,  96,  70],
       [100,  87,  90]])
In [4]: grades.reshape(1, 6)
Out[4]: array([[ 87,  96,  70, 100,  87,  90]])
In [5]: grades
Out[5]:
array([[ 87,  96,  70],
       [100,  87,  90]])
```

Method `resize` modifies the original array's shape:

```
In [6]: grades.resize(1, 6)
In [7]: grades
Out[7]: array([[ 87,  96,  70, 100,  87,  90]])
```

`flatten` vs. `ravel`

You can take a multidimensional array and flatten it into a single dimension with the methods `flatten` and `ravel`. Method `flatten` *deep copies* the original array's data:

```
In [8]: grades = np.array([[87, 96, 70], [100, 87, 90]])
In [9]: grades
Out[9]:
array([[ 87,  96,  70],
       [100,  87,  90]])
In [10]: flattened = grades.flatten()
In [11]: flattened
Out[11]: array([ 87,  96,  70, 100,  87,  90])
In [12]: grades
Out[12]:
array([[ 87,  96,  70],
       [100,  87,  90]])
```

To confirm that `grades` and `flattened` do *not* share the data, let's modify an element of `flattened`, then display both arrays:

```
In [13]: flattened[0] = 100
In [14]: flattened
Out[14]: array([100,  96,  70, 100,  87,  90])
In [15]: grades
Out[15]:
array([[ 87,  96,  70],
       [100,  87,  90]])
```

Method `ravel` produces a *view* of the original array, which *shares* the `grades` array's data:

```
In [16]: raveled = grades.ravel()
In [17]: raveled
Out[17]: array([ 87,  96,  70, 100,  87,  90])
In [18]: grades
Out[18]:
array([[ 87,  96,  70],
       [100,  87,  90]])
```

To confirm that `grades` and `raveled` *share* the same data, let's modify an element of `raveled`, then display both arrays:

```
In [19]: raveled[0] = 100
In [20]: raveled
Out[20]: array([100,  96,  70, 100,  87,  90])
In [21]: grades
Out[21]:
array([[100,  96,  70],
       [100,  87,  90]])
```

Transposing Rows and Columns

You can quickly **transpose** an array's rows and columns—that is “flip” the array, so the rows become the columns and the columns become the rows. The **T attribute** returns a transposed *view* (shallow copy) of the array. The original `grades` array represents two students' grades (the rows) on three exams (the columns). Let's transpose the rows and columns to view the data as the grades on three exams (the rows) for two students (the columns):

```
In [22]: grades.T
Out[22]:
array([[100, 100],
       [ 96,  87],
       [ 70,  90]])
```

Transposing does *not* modify the original array:

```
In [23]: grades
Out[23]:
array([[100,  96,  70],
       [100,  87,  90]])
```

Horizontal and Vertical Stacking

You can combine arrays by adding more columns or more rows—known as *horizontal stacking* and *vertical stacking*. Let’s create another 2-by-3 array of grades:

```
In [24]: grades2 = np.array([[94, 77, 90], [100, 81, 82]])
```

Let’s assume `grades2` represents three additional exam grades for the two students in the `grades` array. We can combine `grades` and `grades2` with NumPy’s **`hstack` (horizontal stack) function** by passing a tuple containing the arrays to combine. The extra parentheses are required because `hstack` expects one argument:

```
In [25]: np.hstack((grades, grades2))
Out[25]:
array([[100, 96, 70, 94, 77, 90],
       [100, 87, 90, 100, 81, 82]])
```

Next, let’s assume that `grades2` represents two more students’ grades on three exams. In this case, we can combine `grades` and `grades2` with NumPy’s **`vstack` (vertical stack) function**:

```
In [26]: np.vstack((grades, grades2))
Out[26]:
array([[100, 96, 70],
       [100, 87, 90],
       [94, 77, 90],
       [100, 81, 82]])
```



Self Check

I (*IPython Session*) Given a 2-by-3 array:

```
array([[1, 2, 3],
       [4, 5, 6]])
```

use `hstack` and `vstack` to produce the following array:

```
array([[1, 2, 3, 1, 2, 3],
       [4, 5, 6, 4, 5, 6],
       [1, 2, 3, 1, 2, 3],
       [4, 5, 6, 4, 5, 6]])
```

Answer:

```
In [1]: import numpy as np
In [2]: a = np.arange(1, 7).reshape(2, 3)
In [3]: a = np.hstack((a, a))
In [4]: a = np.vstack((a, a))
In [5]: a
Out[5]:
array([[1, 2, 3, 1, 2, 3],
       [4, 5, 6, 4, 5, 6],
       [1, 2, 3, 1, 2, 3],
       [4, 5, 6, 4, 5, 6]])
```

7.14 Intro to Data Science: pandas Series and DataFrames

NumPy's array is optimized for homogeneous numeric data that's accessed via integer indices. Data science presents unique demands for which more customized data structures are required. Big data applications must support mixed data types, customized indexing, missing data, data that's not structured consistently and data that needs to be manipulated into forms appropriate for the databases and data analysis packages you use.

Pandas is the most popular library for dealing with such data. It provides two key collections that you'll use in several of our Intro to Data Science sections and throughout the data science case studies—**Series** for one-dimensional collections and **DataFrames** for two-dimensional collections. You can use pandas' **MultiIndex** to manipulate multi-dimensional data in the context of Series and DataFrames.

Wes McKinney created pandas in 2008 while working in industry. The name pandas is derived from the term “panel data,” which is data for measurements over time, such as stock prices or historical temperature readings. McKinney needed a library in which the same data structures could handle both time- and non-time-based data with support for data alignment, missing data, common database-style data manipulations, and more.⁴

NumPy and pandas are intimately related. Series and DataFrames use arrays “under the hood.” Series and DataFrames are valid arguments to many NumPy operations. Similarly, arrays are valid arguments to many Series and DataFrame operations.

Pandas is a massive topic—the PDF of its documentation⁵ is over 2000 pages. In this and the next chapters' Intro to Data Science sections, we present an introduction to pandas. We discuss its Series and DataFrames collections, and use them in support of data preparation. You'll see that Series and DataFrames make it easy for you to perform common tasks like selecting elements a variety of ways, filter/map/reduce operations (central to functional-style programming and big data), mathematical operations, visualization and more.

7.14.1 pandas Series

A **Series** is an enhanced one-dimensional array. Whereas arrays use only zero-based integer indices, Series support custom indexing, including even non-integer indices like strings. Series also offer additional capabilities that make them more convenient for many data-science oriented tasks. For example, Series may have missing data, and many Series operations ignore missing data by default.

Creating a Series with Default Indices

By default, a Series has integer indices numbered sequentially from 0. The following creates a Series of student grades from a list of integers:

```
In [1]: import pandas as pd
```

```
In [2]: grades = pd.Series([87, 100, 94])
```

4. McKinney, Wes. *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython*, pp. 123–165. Sebastopol, CA: O'Reilly Media, 2018.

5. For the latest pandas documentation, see <http://pandas.pydata.org/pandas-docs/stable/>.

The initializer also may be a tuple, a dictionary, an array, another Series or a single value. We'll show a single value momentarily.

Displaying a Series

Pandas displays a Series in two-column format with the indices *left aligned* in the left column and the values *right aligned* in the right column. After listing the Series elements, pandas shows the data type (`dtype`) of the underlying array's elements:

```
In [3]: grades  
Out[3]:  
0    87  
1    100  
2    94  
dtype: int64
```

Note how easy it is to display a Series in this format, compared to the corresponding code for displaying a list in the same two-column format.

Creating a Series with All Elements Having the Same Value

You can create a series of elements that all have the same value:

```
In [4]: pd.Series(98.6, range(3))  
Out[4]:  
0    98.6  
1    98.6  
2    98.6  
dtype: float64
```

The second argument is a one-dimensional iterable object (such as a list, an array or a range) containing the Series' indices. The number of indices determines the number of elements.

Accessing a Series' Elements

You can access a Series's elements by via square brackets containing an index:

```
In [5]: grades[0]  
Out[5]: 87
```

Producing Descriptive Statistics for a Series

Series provides many methods for common tasks including producing various descriptive statistics. Here we show `count`, `mean`, `min`, `max` and `std` (standard deviation):

```
In [6]: grades.count()  
Out[6]: 3  
  
In [7]: grades.mean()  
Out[7]: 93.66666666666667  
  
In [8]: grades.min()  
Out[8]: 87  
  
In [9]: grades.max()  
Out[9]: 100  
  
In [10]: grades.std()  
Out[10]: 6.506407098647712
```

Each of these is a functional-style reduction. Calling Series method `describe` produces all these stats and more:

```
In [11]: grades.describe()
Out[11]:
count      3.000000
mean      93.666667
std       6.506407
min       87.000000
25%      90.500000
50%      94.000000
75%      97.000000
max      100.000000
dtype: float64
```

The 25%, 50% and 75% are **quartiles**:

- 50% represents the median of the sorted values.
- 25% represents the median of the first half of the sorted values.
- 75% represents the median of the second half of the sorted values.

For the quartiles, if there are two middle elements, then their average is that quartile's median. We have only three values in our Series, so the 25% quartile is the average of 87 and 94, and the 75% quartile is the average of 94 and 100. Together, the **interquartile range** is the 75% quartile minus the 25% quartile, which is another measure of dispersion, like standard deviation and variance. Of course, quartiles and interquartile range are more useful in larger datasets.

Creating a Series with Custom Indices

You can specify *custom* indices with the `index` keyword argument:

```
In [12]: grades = pd.Series([87, 100, 94], index=['Wally', 'Eva', 'Sam'])

In [13]: grades
Out[13]:
Wally    87
Eva     100
Sam     94
dtype: int64
```

In this case, we used string indices, but you can use other immutable types, including integers not beginning at 0 and nonconsecutive integers. Again, notice how nicely and concisely pandas formats a Series for display.

Dictionary Initializers

If you initialize a Series with a dictionary, its keys become the Series' indices, and its values become the Series' element values:

```
In [14]: grades = pd.Series({'Wally': 87, 'Eva': 100, 'Sam': 94})

In [15]: grades
Out[15]:
Wally    87
Eva     100
Sam     94
dtype: int64
```

Accessing Elements of a Series Via Custom Indices

In a Series with custom indices, you can access individual elements via square brackets containing a custom index value:

```
In [16]: grades['Eva']
Out[16]: 100
```

If the custom indices are strings that could represent valid Python identifiers, pandas automatically adds them to the Series as attributes that you can access via a dot (.), as in:

```
In [17]: grades.Wally
Out[17]: 87
```

Series also has *built-in* attributes. For example, the **dtype attribute** returns the underlying array's element type:

```
In [18]: grades.dtype
Out[18]: dtype('int64')
```

and the **values attribute** returns the underlying array:

```
In [19]: grades.values
Out[19]: array([ 87, 100,  94])
```

Creating a Series of Strings

If a Series contains strings, you can use its **str attribute** to call string methods on the elements. First, let's create a Series of hardware-related strings:

```
In [20]: hardware = pd.Series(['Hammer', 'Saw', 'Wrench'])

In [21]: hardware
Out[21]:
0    Hammer
1      Saw
2    Wrench
dtype: object
```

Note that pandas also *right-aligns* string element values and that the **dtype** for strings is **object**.

Let's call string method **contains** on each element to determine whether the value of each element contains a lowercase 'a':

```
In [22]: hardware.str.contains('a')
Out[22]:
0    True
1    True
2   False
dtype: bool
```

Pandas returns a Series containing **bool** values indicating the **contains** method's result for each element—the element at index 2 ('Wrench') does not contain an 'a', so its element in the resulting Series is **False**. Note that pandas handles the iteration internally for you—another example of functional-style programming. The **str attribute** provides many string-processing methods that are similar to those in Python's string type. For a list, see: <https://pandas.pydata.org/pandas-docs/stable/api.html#string-handling>.

The following uses string method **upper** to produce a *new Series* containing the uppercase versions of each element in **hardware**:

```
In [23]: hardware.str.upper()
Out[23]:
0    HAMMER
1      SAW
2   WRENCH
dtype: object
```



Self Check

I (*IPython Session*) Use the NumPy's random-number generation to create an array of five random integers that represent summertime temperatures in the range 60–100, then perform the following tasks:

- Convert the array into the Series named `temperatures` and display it.
- Determine the lowest, highest and average temperatures.
- Produce descriptive statistics for the Series.

Answer:

```
In [1]: import numpy as np

In [2]: import pandas as pd

In [3]: temps = np.random.randint(60, 101, 6)

In [4]: temperatures = pd.Series(temps)

In [5]: temperatures
Out[5]:
0    98
1    62
2    63
3    70
4    69
dtype: int64

In [6]: temperatures.min()
Out[6]: 62

In [7]: temperatures.max()
Out[7]: 98

In [8]: temperatures.mean()
Out[8]: 72.4

In [9]: temperatures.describe()
Out[9]:
count      5.000000
mean      72.000000
std       14.741099
min      62.000000
25%      63.000000
50%      69.000000
75%      70.000000
max      98.000000
dtype: float64
```

7.14.2 DataFrames

A **DataFrame** is an enhanced two-dimensional array. Like **Series**, **DataFrames** can have custom row and column indices, and offer additional operations and capabilities that make them more convenient for many data-science oriented tasks. **DataFrames** also support missing data. Each column in a **DataFrame** is a **Series**. The **Series** representing each column may contain different element types, as you'll soon see when we discuss loading datasets into **DataFrames**.

Creating a DataFrame from a Dictionary

Let's create a **DataFrame** from a dictionary that represents student grades on three exams:

```
In [1]: import pandas as pd

In [2]: grades_dict = {'Wally': [87, 96, 70], 'Eva': [100, 87, 90],
...:             'Sam': [94, 77, 90], 'Katie': [100, 81, 82],
...:             'Bob': [83, 65, 85]}
...:

In [3]: grades = pd.DataFrame(grades_dict)

In [4]: grades
Out[4]:
   Wally  Eva  Sam  Katie  Bob
0      87  100   94    100   83
1      96   87   77     81   65
2      70   90   90     82   85
```

Pandas displays **DataFrames** in tabular format with the indices *left aligned* in the index column and the remaining columns' values *right aligned*. The dictionary's *keys* become the column names and the *values* associated with each key become the element values in the corresponding column. Shortly, we'll show how to "flip" the rows and columns. By default, the row indices are auto-generated integers starting from 0.

Customizing a DataFrame's Indices with the index Attribute

We could have specified custom indices with the `index` keyword argument when we created the **DataFrame**, as in:

```
pd.DataFrame(grades_dict, index=['Test1', 'Test2', 'Test3'])
```

Let's use the **index attribute** to change the **DataFrame**'s indices from sequential integers to labels:

```
In [5]: grades.index = ['Test1', 'Test2', 'Test3']

In [6]: grades
Out[6]:
   Wally  Eva  Sam  Katie  Bob
Test1    87  100   94    100   83
Test2    96   87   77     81   65
Test3    70   90   90     82   85
```

When specifying the indices, you must provide a one-dimensional collection that has the same number of elements as there are *rows* in the **DataFrame**; otherwise, a **ValueError** occurs. **Series** also provides an **index attribute** for changing an existing **Series**' indices.

Accessing a DataFrame's Columns

One benefit of pandas is that you can quickly and conveniently look at your data in many different ways, including selecting portions of the data. Let's start by getting Eva's grades by name, which displays her column as a `Series`:

```
In [7]: grades['Eva']
Out[7]:
Test1    100
Test2     87
Test3     90
Name: Eva, dtype: int64
```

If a `DataFrame`'s column-name strings are valid Python identifiers, you can use them as attributes. Let's get Sam's grades with the `Sam` *attribute*:

```
In [8]: grades.Sam
Out[8]:
Test1    94
Test2     77
Test3     90
Name: Sam, dtype: int64
```

Selecting Rows via the `loc` and `iloc` Attributes

Though `DataFrames` support indexing capabilities with `[]`, the pandas documentation recommends using the attributes `loc`, `iloc`, `at` and `iat`, which are optimized to access `DataFrames` and also provide additional capabilities beyond what you can do only with `[]`. Also, the documentation states that indexing with `[]` *often* produces a copy of the data, which is a logic error if you attempt to assign new values to the `DataFrame` by assigning to the result of the `[]` operation.

You can access a row by its label via the `DataFrame`'s **`loc` attribute**. The following lists all the grades in the row 'Test1':

```
In [9]: grades.loc['Test1']
Out[9]:
Wally     87
Eva      100
Sam       94
Katie     100
Bob       83
Name: Test1, dtype: int64
```

You also can access rows by integer zero-based indices using the **`i`l`oc` attribute** (the `i` in `i`l`oc` means that it's used with integer indices). The following lists all the grades in the second row:

```
In [10]: grades.iloc[1]
Out[10]:
Wally     96
Eva      87
Sam       77
Katie     81
Bob       65
Name: Test2, dtype: int64
```

Selecting Rows via Slices and Lists with the loc and iloc Attributes

The index can be a *slice*. When using slices containing labels with `loc`, the range specified *includes* the high index ('Test3'):

```
In [11]: grades.loc['Test1':'Test3']
Out[11]:
      Wally  Eva  Sam  Katie  Bob
Test1    87  100   94   100   83
Test2    96   87   77   81   65
Test3    70   90   90   82   85
```

When using slices containing integer indices with `iloc`, the range you specify *excludes* the high index (2):

```
In [12]: grades.iloc[0:2]
Out[12]:
      Wally  Eva  Sam  Katie  Bob
Test1    87  100   94   100   83
Test2    96   87   77   81   65
```

To select *specific rows*, use a *list* rather than slice notation with `loc` or `iloc`:

```
In [13]: grades.loc[['Test1', 'Test3']]
Out[13]:
      Wally  Eva  Sam  Katie  Bob
Test1    87  100   94   100   83
Test3    70   90   90   82   85

In [14]: grades.iloc[[0, 2]]
Out[14]:
      Wally  Eva  Sam  Katie  Bob
Test1    87  100   94   100   83
Test3    70   90   90   82   85
```

Selecting Subsets of the Rows and Columns

So far, we've selected only *entire* rows. You can focus on small subsets of a DataFrame by selecting rows *and* columns using two slices, two lists or a combination of slices and lists.

Suppose you want to view only Eva's and Katie's grades on Test1 and Test2. We can do that by using `loc` with a slice for the two consecutive rows and a list for the two non-consecutive columns:

```
In [15]: grades.loc['Test1':'Test2', ['Eva', 'Katie']]
Out[15]:
      Eva  Katie
Test1  100   100
Test2   87    81
```

The slice '`Test1':'Test2'` selects the rows for `Test1` and `Test2`. The list `['Eva', 'Katie']` selects only the corresponding grades from those two columns.

Let's use `iloc` with a list and a slice to select the first and third tests and the first three columns for those tests:

```
In [16]: grades.iloc[[0, 2], 0:3]
Out[16]:
      Wally  Eva  Sam
Test1    87  100   94
Test3    70   90   90
```

Boolean Indexing

One of pandas' more powerful selection capabilities is **Boolean indexing**. For example, let's select all the A grades—that is, those that are greater than or equal to 90:

```
In [17]: grades[grades >= 90]
Out[17]:
      Wally    Eva    Sam   Katie   Bob
Test1     NaN  100.0  94.0  100.0   NaN
Test2    96.0    NaN    NaN    NaN   NaN
Test3     NaN  90.0  90.0    NaN   NaN
```

Pandas checks every grade to determine whether its value is greater than or equal to 90 and, if so, includes it in the new DataFrame. Grades for which the condition is `False` are represented as **`NaN` (not a number)** in the new DataFrame. `NaN` is pandas' notation for missing values.

Let's select all the B grades in the range 80–89:

```
In [18]: grades[(grades >= 80) & (grades < 90)]
Out[18]:
      Wally    Eva    Sam   Katie   Bob
Test1    87.0    NaN    NaN    NaN  83.0
Test2     NaN  87.0    NaN   81.0   NaN
Test3     NaN    NaN    NaN   82.0  85.0
```

Pandas Boolean indices combine multiple conditions with the Python operator `&` (bitwise AND), *not* the `and` Boolean operator. For `or` conditions, use `|` (bitwise OR). NumPy also supports Boolean indexing for arrays, but always returns a one-dimensional array containing only the values that satisfy the condition.

Accessing a Specific DataFrame Cell by Row and Column

You can use a DataFrame's `at` and `iat` attributes to get a single value from a DataFrame. Like `loc` and `iloc`, `at` uses labels and `iat` uses integer indices. In each case, the row and column indices must be separated by a comma. Let's select Eva's `Test2` grade (87) and Wally's `Test3` grade (70)

```
In [19]: grades.at['Test2', 'Eva']
Out[19]: 87

In [20]: grades.iat[2, 0]
Out[20]: 70
```

You also can assign new values to specific elements. Let's change Eva's `Test2` grade to 100 using `at`, then change it back to 87 using `iat`:

```
In [21]: grades.at['Test2', 'Eva'] = 100

In [22]: grades.at['Test2', 'Eva']
Out[22]: 100

In [23]: grades.iat[1, 2] = 87

In [24]: grades.iat[1, 2]
Out[24]: 87.0
```

Descriptive Statistics

Both Series and DataFrames have a **describe** method that calculates basic descriptive statistics for the data and returns them as a DataFrame. In a DataFrame, the statistics are calculated by column (again, soon you'll see how to flip rows and columns):

```
In [25]: grades.describe()
```

```
Out[25]:
```

	Wally	Eva	Sam	Katie	Bob
count	3.000000	3.000000	3.000000	3.000000	3.000000
mean	84.333333	92.333333	87.000000	87.666667	77.666667
std	13.203535	6.806859	8.888194	10.692677	11.015141
min	70.000000	87.000000	77.000000	81.000000	65.000000
25%	78.500000	88.500000	83.500000	81.500000	74.000000
50%	87.000000	90.000000	90.000000	82.000000	83.000000
75%	91.500000	95.000000	92.000000	91.000000	84.000000
max	96.000000	100.000000	94.000000	100.000000	85.000000

As you can see, `describe` gives you a quick way to summarize your data. It nicely demonstrates the power of array-oriented programming with a clean, concise functional-style call. Pandas handles internally all the details of calculating these statistics for each column. You might be interested in seeing similar statistics on test-by-test basis so you can see how all the students performs on Tests 1, 2 and 3—we'll show how to do that shortly.

By default, pandas calculates the descriptive statistics with floating-point values and displays them with six digits of precision. You can control the precision and other default settings with pandas' `set_option` function:

```
In [26]: pd.set_option('precision', 2)
```

```
In [27]: grades.describe()
```

```
Out[27]:
```

	Wally	Eva	Sam	Katie	Bob
count	3.00	3.00	3.00	3.00	3.00
mean	84.33	92.33	87.00	87.67	77.67
std	13.20	6.81	8.89	10.69	11.02
min	70.00	87.00	77.00	81.00	65.00
25%	78.50	88.50	83.50	81.50	74.00
50%	87.00	90.00	90.00	82.00	83.00
75%	91.50	95.00	92.00	91.00	84.00
max	96.00	100.00	94.00	100.00	85.00

For student grades, the most important of these statistics is probably the mean. You can calculate that for each student simply by calling `mean` on the DataFrame:

```
In [28]: grades.mean()
```

```
Out[28]:
```

Wally	84.33
Eva	92.33
Sam	87.00
Katie	87.67
Bob	77.67
	dtype: float64

In a moment, we'll show how to get the average of all the students' grades on each test in one line of additional code.

Transposing the DataFrame with the T Attribute

You can quickly `transpose` the rows and columns—so the rows become the columns, and the columns become the rows—by using the `T attribute`:

```
In [29]: grades.T
Out[29]:
   Test1  Test2  Test3
Wally    87    96    70
Eva     100    87    90
Sam      94    77    90
Katie    100    81    82
Bob      83    65    85
```

`T` returns a transposed *view* (not a copy) of the `DataFrame`.

Let's assume that rather than getting the summary statistics by student, you want to get them by test. Simply call `describe` on `grades.T`, as in:

```
In [30]: grades.T.describe()
Out[30]:
   Test1  Test2  Test3
count    5.00    5.00    5.00
mean    92.80   81.20   83.40
std      7.66   11.54    8.23
min     83.00   65.00   70.00
25%     87.00   77.00   82.00
50%     94.00   81.00   85.00
75%    100.00   87.00   90.00
max    100.00   96.00   90.00
```

To see the average of all the students' grades on each test, just call `mean` on the `T attribute`:

```
In [31]: grades.T.mean()
Out[31]:
Test1    92.8
Test2    81.2
Test3    83.4
dtype: float64
```

Sorting by Rows by Their Indices

You'll often sort data for easier readability. You can sort a `DataFrame` by its rows or columns, based on their indices or values. Let's sort the rows by their *indices* in *descending* order using `sort_index` and its keyword argument `ascending=False` (the default is to sort in *ascending* order). This returns a new `DataFrame` containing the sorted data:

```
In [32]: grades.sort_index(ascending=False)
Out[32]:
   Wally  Eva  Sam  Katie  Bob
Test3    70   90   90    82   85
Test2    96   87   77    81   65
Test1    87  100   94   100   83
```

Sorting by Column Indices

Now let's sort the columns into ascending order (left-to-right) by their column names. Passing the `axis=1 keyword argument` indicates that we wish to sort the *column* indices, rather than the row indices—`axis=0` (the default) sorts the *row* indices:

```
In [33]: grades.sort_index(axis=1)
Out[33]:
   Bob  Eva  Katie  Sam  Wally
Test1    83  100    100   94    87
Test2    65   87     81   77    96
Test3    85   90     82   90    70
```

Sorting by Column Values

Let's assume we want to see Test1's grades in descending order so we can see the students' names in highest-to-lowest grade order. We can call the method `sort_values` as follows:

```
In [34]: grades.sort_values(by='Test1', axis=1, ascending=False)
Out[34]:
      Eva  Katie  Sam  Wally  Bob
Test1  100    100   94    87   83
Test2   87     81   77    96   65
Test3   90     82   90    70   85
```

The `by` and `axis` keyword arguments work together to determine which values will be sorted. In this case, we sort based on the column values (`axis=1`) for Test1.

Of course, it might be easier to read the grades and names if they were in a column, so we can sort the transposed DataFrame instead. Here, we did not need to specify the `axis` keyword argument, because `sort_values` sorts data in a specified column by default:

```
In [35]: grades.T.sort_values(by='Test1', ascending=False)
Out[35]:
      Test1  Test2  Test3
Eva      100     87     90
Katie     100     81     82
Sam       94      77     90
Wally     87      96     70
Bob       83      65     85
```

Finally, since you're sorting only Test1's grades, you might not want to see the other tests at all. So, let's combine selection with sorting:

```
In [36]: grades.loc['Test1'].sort_values(ascending=False)
Out[36]:
Katie    100
Eva     100
Sam      94
Wally    87
Bob      83
Name: Test1, dtype: int64
```

Copy vs. In-Place Sorting

By default the `sort_index` and `sort_values` return a *copy* of the original DataFrame, which could require substantial memory in a big data application. You can sort the DataFrame *in place*, rather than *copying* the data. To do so, pass the keyword argument `inplace=True` to either `sort_index` or `sort_values`.

We've shown many pandas Series and DataFrame features. In the next chapter's Intro to Data Science section, we'll use Series and DataFrames for *data munging*—cleaning and preparing data for use in your database or analytics software.

✓ Self Check

- I (*IPython Session*) Given the following dictionary:

```
temps = {'Mon': [68, 89], 'Tue': [71, 93], 'Wed': [66, 82],
         'Thu': [75, 97], 'Fri': [62, 79]}
```

perform the following tasks:

- Convert the dictionary into the DataFrame named `temperatures` with 'Low' and 'High' as the indices, then display the DataFrame.
- Use the column names to select only the columns for 'Mon' through 'Wed'.
- Use the row index 'Low' to select only the low temperatures for each day.
- Set the floating-point precision to 2, then calculate the average temperature for each day.
- Calculate the average low and high temperatures.

Answer:

```
In [1]: import pandas as pd

In [2]: temps = {'Mon': [68, 89], 'Tue': [71, 93], 'Wed': [66, 82],
   ...:           'Thu': [75, 97], 'Fri': [62, 79]}
   ...:

In [3]: temperatures = pd.DataFrame(temps, index=['Low', 'High']) # (a)

In [4]: temperatures # (a)
Out[4]:
   Mon   Tue   Wed   Thu   Fri
Low    68    71    66    75    62
High   89    93    82    97    79

In [5]: temperatures.loc[:, 'Mon':'Wed'] # (b)
Out[5]:
   Mon   Tue   Wed
Low    68    71    66
High   89    93    82

In [6]: temperatures.loc['Low'] # (c)
Out[6]:
Mon    68
Tue    71
Wed    66
Thu    75
Fri    62
Name: Low, dtype: int64

In [7]: pd.set_option('precision', 2) # (d)

In [8]: temperatures.mean() # (d)
Out[8]:
Mon    78.5
Tue    82.0
Wed    74.0
Thu    86.0
Fri    70.5
dtype: float64
```

```
In [9]: temperatures.mean(axis=1) # (e)
Out[9]:
Low      68.4
High     88.0
dtype: float64
```

7.15 Wrap-Up

This chapter explored the use of NumPy’s high-performance `ndarrays` for storing and retrieving data, and for performing common data manipulations concisely and with reduced chance of errors with functional-style programming. We refer to `ndarrays` simply by their synonym, `arrays`.

The chapter examples demonstrated how to create, initialize and refer to individual elements of one- and two-dimensional arrays. We used attributes to determine an array’s size, shape and element type. We showed functions that create arrays of 0s, 1s, specific values or ranges values. We compared list and array performance with the IPython `%timeit` magic and saw that arrays are up to two orders of magnitude faster.

We used `array` operators and NumPy universal functions to perform element-wise calculations on every element of arrays that have the same shape. You also saw that NumPy uses broadcasting to perform element-wise operations between arrays and scalar values, and between arrays of different shapes. We introduced various built-in `array` methods for performing calculations using all elements of an array, and we showed how to perform those calculations row-by-row or column-by-column. We demonstrated various `array` slicing and indexing capabilities that are more powerful than those provided by Python’s built-in collections. We demonstrated various ways to reshape arrays. We discussed how to shallow copy and deep copy arrays and other Python objects.

In the Intro to Data Science section, we began our multisection introduction to the popular `pandas` library that you’ll use in many of the data science case study chapters. You learned that many big data applications need more flexible collections than NumPy’s `arrays`, collections that support mixed data types, custom indexing, missing data, data that’s not structured consistently and data that needs to be manipulated into forms appropriate for the databases and data analysis packages you use.

We showed how to create and manipulate `pandas` array-like one-dimensional `Series` and two-dimensional `DataFrames`. We customized `Series` and `DataFrame` indices. You saw `pandas`’ nicely formatted outputs and customized the precision of floating-point values. We showed various ways to access and select data in `Series` and `DataFrames`. We used method `describe` to calculate basic descriptive statistics for `Series` and `DataFrames`. We showed how to transpose `DataFrame` rows and columns via the `T` attribute. You saw several ways to sort `DataFrames` using their index values, their column names, the data in their rows and the data in their columns. You’re now familiar with four powerful array-like collections—lists, arrays, `Series` and `DataFrames`—and the contexts in which to use them. We’ll add a fifth—`tensors`—in the “Deep Learning” chapter.

In the next chapter, we take a deeper look at strings, string formatting and string methods. We also introduce regular expressions, which we’ll use to match patterns in text. The capabilities you’ll learn will help you prepare for the “Natural Language Processing (NLP)” chapter and other key data science chapters. In the next chapter’s Intro to Data Science section, we’ll introduce `pandas` *data munging*—preparing data for use in your

database or analytics software. In subsequent chapters, we'll use pandas for basic time-series analysis and introduce pandas visualization capabilities.

Exercises

Use IPython sessions for each exercise where practical. Each time you create or modify an array, Series or DataFrame, display the result.

7.1 (Filling arrays) Fill a 2-by-3 array with ones, a 3-by-3 array with zeros and a 2-by-5 array with 7s.

7.2 (Broadcasting) Use `arange` to create a 2-by-2 array containing the numbers 0–3.

Use broadcasting to perform each of the following operations on the original array:

- Cube every element of the array.
- Add 7 to every element of the array.
- Multiply every element of the array by 2.

7.3 (Element-Wise array Multiplication) Create a 3-by-3 array containing the even integers from 2 through 18. Create a second 3-by-3 array containing the integers from 9 down to 1, then multiply the first array by the second.

7.4 (array from List of Lists) Create a 2-by-5 array from an argument which is a list of the two five-element lists [2, 3, 5, 7, 11] and [13, 17, 19, 23, 29].

7.5 (Flattening arrays with `flatten` vs. `ravel`) Create a 2-by-3 array containing the first six powers of 2 beginning with 2^0 . Flatten the array first with method `flatten`, then with `ravel`. In each case, display the result then display the original array to show that it was unmodified.

7.6 (Research: array Method `astype`) Research in the NumPy documentation the array method `astype`, which converts an array's elements to another type. Use `linspace` and `reshape` to create a 2-by-3 array with the values 1.1, 2.2, ..., 6.6. Then use `astype` to convert the array to an array of integers.

7.7 (Challenge Project: Reimplement NumPy array Output) You saw that NumPy outputs two-dimensional arrays in a nice column-based format that right-aligns every element in a field width. The field width's size is determined by the array element value that requires the most character positions to display. To understand how powerful it is to have this formatting simply built-in, write a function that reimplements NumPy's array formatting for two-dimensional arrays using loops. Assume the array contains only positive integer values.

7.8 (Challenge Project: Reimplement DataFrame Output) You saw that pandas displays DataFrames in an attractive column-based format with row and column labels. The values within each column are right aligned in the same field width, which is determined by that column's widest value. To understand how powerful it is to have this formatting built-in, write a function that reimplements DataFrame formatting using loops. Assume the DataFrame contains only positive integer values and that both the row and column labels are each integer values beginning at 0.

7.9 (Indexing and Slicing arrays) Create an array containing the values 1–15, reshape it into a 3-by-5 array, then use indexing and slicing techniques to perform each of the following operations:

- Select row 2.

- b) Select column 5.
- c) Select rows 0 and 1.
- d) Select columns 2–4.
- e) Select the element that is in row 1 and column 4.
- f) Select all elements from rows 1 and 2 that are in columns 0, 2 and 4.

7.10 (*Project: Two-Player, Two-Dimensional Tic-Tac-Toe*) Write a script to play two-dimensional Tic-Tac-Toe between two human players who alternate entering their moves on the same computer. Use a 3-by-3 two-dimensional array. Each player indicates their moves by entering a pair of numbers representing the row and column indices of the square in which they want to place their mark, either an 'X' or an 'O'. When the first player moves, place an 'X' in the specified square. When the second player moves, place an 'O' in the specified square. Each move must be to an empty square. After each move, determine whether the game has been won and whether it's a draw.

7.11 (*Challenge Project: Tic-Tac-Toe with Player Against the Computer*) Modify your script from the previous exercise so that the computer makes the moves for one of the players. Also, allow the player to specify whether he or she wants to go first or second.

7.12 (*Super Challenge Project: 3D Tic-Tac-Toe with Player Against the Computer*) Develop a script that plays three-dimensional Tic-Tac-Toe on a 4-by-4-by-4 board. [Note: This is an extremely challenging project! In the “Deep Learning” chapter, you’ll learn techniques that will help you develop and AI-based approach to solving this problem.]

7.13 (*Research and Use Other Broadcasting Capabilities*) Research the NumPy broadcasting rules, then create your own arrays to test the rules.

7.14 (*Horizontal and Vertical Stacking*) Create the two-dimensional arrays

```
array1 = np.array([[0, 1], [2, 3]])
array2 = np.array([[4, 5], [6, 7]])
```

- a) Use vertical stacking to create the 4-by-2 array named array3 with array1 stacked on top of array2.
- b) Use horizontal stacking to create the 2-by-4 array named array4 with array2 to the right of array1.
- c) Use vertical stacking with two copies of array4 to create a 4-by-4 array5.
- d) Use horizontal stacking with two copies of array3 to create a 4-by-4 array6.

7.15 (*Research and Use NumPy’s `concatenate` Function*) Research NumPy function concatenate, then use it to reimplement the previous exercise.

7.16 (*Research: NumPy `tile` Function*) Research and use NumPy’s tile function to create a checkerboard pattern of dashes and asterisks.

7.17 (*Research: NumPy `bincount` Functions*) Research and use the NumPy bincount function to count the number of occurrences of each non-negative integer in a 5-by-5 array of random integers in the range 0–99.

7.18 (*Median and Mode of an array*) NumPy arrays offer a mean method, but not median or mode. Write functions median and mode that use existing NumPy capabilities to determine the median (middle) and mode (most frequent) of the values in an array. Your functions should determine the median and mode regardless of the array’s shape. Test your function on three arrays of different shapes.

7.19 (*Enhanced Median and Mode of an array*) Modify your functions from the previous exercise to allow the user to provide an `axis` keyword argument so the calculations can be performed row-by-row or column-by-column on a two-dimensional array.

7.20 (*Performance Analysis*) In this chapter, we used `%timeit` to compare the average execution times of generating a list of 6,000,000 random die rolls vs. generating an array of 6,000,000 random die rolls. Though we saw approximately two orders of magnitude performance improvement with `array`, we generated the list and the `array` using two *different* random-number generators and different techniques for building each collection. If you use the same techniques we showed to generate a one-element list and a one-element array, creating the list is slightly faster. Repeat the `%timeit` operations for one-element collections. Then do it again for 10, 100, 1000, 10,000, 100,000, and 1,000,000 elements and compare the results on your system. The table below shows the results on our system, with measurements in nanoseconds (ns), microseconds (μ s), milliseconds (ms) and seconds (s).

Number of values	List average execution time	array average execution time
1	1.56 μ s \pm 25.2 ns	1.89 μ s \pm 24.4 ns
10	11.6 μ s \pm 59.6 ns	1.96 μ s \pm 27.6 ns
100	109 μ s \pm 1.61 μ s	3 μ s \pm 147 ns
1000	1.09 ms \pm 8.59 μ s	12.3 μ s \pm 419 ns
10,000	11.1 ms \pm 210 μ s	102 μ s \pm 669 ns
100,000	111 ms \pm 1.77 ms	1.02 ms \pm 32.9 μ s
1,000,000	1.1 s \pm 8.47 ms	10.1 ms \pm 250 μ s

This analysis shows why `%timeit` is convenient for quick performance studies. However, you also need to develop performance-analysis wisdom. Many factors can affect performance—the underlying hardware, the operating system, the interpreter or compiler you’re using, the other applications running on your computer at the same time, and many more. The way we thought about performance over the years is changing rapidly now with big data, data analytics and artificial intelligence. As we head into the AI portion of the book, you’ll place enormous performance demands on your system, so it’s always good to be thinking about performance issues.

7.21 (*Shallow vs. Deep Copy*) In this chapter, we discussed shallow vs. deep copies of arrays. Python’s built-in list and dictionary types have `copy` methods that perform *shallow* copies. Using the following dictionary

```
dictionary = {'Sophia': [97, 88]}
```

demonstrate that a dictionary’s `copy` method indeed performs a shallow copy. To do so, call `copy` to make the shallow copy, modify the list stored in the original dictionary, then display both dictionaries to see that they have the same contents.

Next, use the `copy` module’s `deepcopy` function to create a *deep* copy of the dictionary. Modify the list stored in the original dictionary, then display both dictionaries to prove that each has its own data.

7.22 (*Pandas: Series*) Perform the following tasks with pandas `Series`:

- a) Create a `Series` from the list [7, 11, 13, 17].

- b) Create a Series with five elements that are all 100.0.
- c) Create a Series with 20 elements that are all random numbers in the range 0 to 100. Use method describe to produce the Series' basic descriptive statistics.
- d) Create a Series called temperatures of the floating-point values 98.6, 98.9, 100.2 and 97.9. Using the index keyword argument, specify the custom indices 'Julie', 'Charlie', 'Sam' and 'Andrea'.
- e) Form a dictionary from the names and values in Part (d), then use it to initialize a Series.

7.23

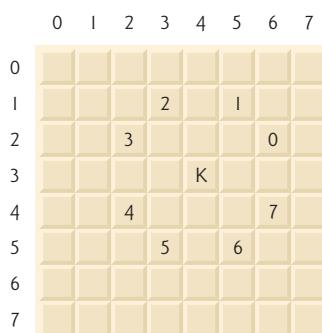
(*Pandas: DataFrames*) Perform the following tasks with pandas DataFrames:

- a) Create a DataFrame named temperatures from a dictionary of three temperature readings each for 'Maxine', 'James' and 'Amanda'.
- b) Recreate the DataFrame temperatures in Part (a) with custom indices using the index keyword argument and a list containing 'Morning', 'Afternoon' and 'Evening'.
- c) Select from temperatures the column of temperature readings for 'Maxine'.
- d) Select from temperatures the row of 'Morning' temperature readings.
- e) Select from temperatures the rows for 'Morning' and 'Evening' temperature readings.
- f) Select from temperatures the columns of temperature readings for 'Amanda' and 'Maxine'.
- g) Select from temperatures the elements for 'Amanda' and 'Maxine' in the 'Morning' and 'Afternoon'.
- h) Use the describe method to produce temperatures' descriptive statistics.
- i) Transpose temperatures.
- j) Sort temperatures so that its column names are in alphabetical order.

7.24

(*AI Project: Introducing Heuristic Programming with the Knight's Tour*) An interesting puzzler for chess buffs is the Knight's Tour problem, originally proposed by the mathematician Euler. Can the knight piece move around an empty chessboard and touch each of the 64 squares once and only once? We study this intriguing problem in depth here.

The knight makes only L-shaped moves (two spaces in one direction and one space in a perpendicular direction). Thus, as shown in the figure below, from a square near the middle of an empty chessboard, the knight (labeled K) can make eight different moves (numbered 0 through 7).



- a) Draw an eight-by-eight chessboard on a sheet of paper, and attempt a Knight's Tour by hand. Put a 1 in the starting square, a 2 in the second square, a 3 in the third, and so on. Before starting the tour, estimate how far you think you'll get, remembering that a full tour consists of 64 moves. How far did you get? Was this close to your estimate?
- b) Now let's develop a script that will move the knight around a chessboard represented by an eight-by-eight two-dimensional array named `board`. Initialize each square to zero. We describe each of the eight possible moves in terms of its horizontal and vertical components. For example, a move of type 0, as shown in the preceding figure, consists of moving two squares horizontally to the right and one square vertically upward. A move of type 2 consists of moving one square horizontally to the left and two squares vertically upward. Horizontal moves to the left and vertical moves upward are indicated with negative numbers. The eight moves may be described by two one-dimensional arrays, `horizontal` and `vertical`, as follows:

<code>horizontal[0] = 2</code>	<code>vertical[0] = -1</code>
<code>horizontal[1] = 1</code>	<code>vertical[1] = -2</code>
<code>horizontal[2] = -1</code>	<code>vertical[2] = -2</code>
<code>horizontal[3] = -2</code>	<code>vertical[3] = -1</code>
<code>horizontal[4] = -2</code>	<code>vertical[4] = 1</code>
<code>horizontal[5] = -1</code>	<code>vertical[5] = 2</code>
<code>horizontal[6] = 1</code>	<code>vertical[6] = 2</code>
<code>horizontal[7] = 2</code>	<code>vertical[7] = 1</code>

Let the variables `current_row` and `current_column` indicate the row and column, respectively, of the knight's current position. To make a move of type `move_number` (a value 0–7), your script should use the statements

```
current_row += vertical[move_number]
current_column += horizontal[move_number]
```

Write a script to move the knight around the chessboard. Keep a counter that varies from 1 to 64. Record the latest count in each square the knight moves to. Test each potential move to see if the knight has already visited that square. Test every potential move to ensure that the knight does not land off the chessboard. Run the application. How many moves did the knight make?

- c) After attempting to write and run a Knight's Tour script, you've probably developed some valuable insights. We'll use these insights to develop a *heuristic* (i.e., a common-sense rule) for moving the knight. Heuristics do not guarantee success, but a carefully developed heuristic greatly improves the chance of success. You may have observed that the outer squares are more troublesome than the squares nearer the center of the board. In fact, the most troublesome or inaccessible squares are the four corners.

Intuition may suggest that you should attempt to move the knight to the most troublesome squares first and leave open those that are easiest to get to so that when the board gets congested near the end of the tour, there will be a greater chance of success.

We could develop an “accessibility heuristic” by classifying each of the squares according to how accessible it is and always moving the knight (using

the knight's L-shaped moves) to the most inaccessible square. We fill two-dimensional array `accessibility` with numbers indicating from how many squares each particular square is accessible. On a blank chessboard, each of the 16 squares nearest the center is rated as 8, each corner square is rated as 2, and the other squares have accessibility numbers of 3, 4 or 6 as follows:

2	3	4	4	4	4	3	2
3	4	6	6	6	6	4	3
4	6	8	8	8	8	6	4
4	6	8	8	8	8	6	4
4	6	8	8	8	8	6	4
4	6	8	8	8	8	6	4
3	4	6	6	6	6	4	3
2	3	4	4	4	4	3	2

Write a new version of the Knight's Tour, using the accessibility heuristic. The knight should always move to the square with the lowest accessibility number. In case of a tie, the knight may move to any of the tied squares. Therefore, the tour may begin in any of the four corners. [Note: As the knight moves around the chessboard, your application should reduce the accessibility numbers as more squares become occupied. In this way, at any given time during the tour, each available square's accessibility number will remain equal to precisely the number of squares from which that square may be reached.] Run this version of your script. Did you get a full tour? Modify the script to run 64 tours, one starting from each square of the chessboard. How many full tours did you get?

7.25 (Knight's Tour Project: Brute-Force Approaches) In Part (c) of the previous exercise, we developed a solution to the Knight's Tour problem. The approach used, called the “accessibility heuristic,” generates many solutions and executes efficiently.

As computers continue to increase in power, we'll be able to solve more problems with sheer computer power and relatively unsophisticated algorithms. Let's call this approach “brute-force” problem solving.

- Use random-number generation to enable the knight to walk around the chessboard (in its legitimate L-shaped moves) at random. Your script should run one tour and display the final chessboard. How far did the knight get?
- Most likely, the script in Part (a) produced a relatively short tour. Now modify your script to attempt 1,000,000 tours. Use a one-dimensional array to keep track of the number of tours of each length. When your script finishes attempting the 1,000,000 tours, it should display this information in a neat tabular format. What was the best result?
- Most likely, the script in Part (b) gave you some “respectable” tours, but no full tours. Now let your script run until it produces a full tour. [Caution: This version of the script could run for hours on a powerful computer.] Once again, keep a table of the number of tours of each length, and display this table when the first full tour is found. How many tours did your script attempt before producing a full tour? How much time did it take?
- Compare the brute-force version of the Knight's Tour with the accessibility-heuristic version. Which required a more careful study of the problem? Which

algorithm was more challenging to develop? Which required more computer power? Could we be certain (in advance) of obtaining a full tour with the accessibility-heuristic approach? Could we be certain (in advance) of obtaining a full tour with the brute-force approach? Argue the pros and cons of brute-force problem-solving in general.

7.26 (Knight's Tour Project: Closed-Tour Test) In the Knight's Tour, a full tour occurs when the knight makes 64 moves, touching each square of the chessboard once and only once. A closed tour occurs when the 64th move is one move away from the square in which the knight started the tour. Modify the script you wrote in Exercise 7.24 to test for a closed tour if a full tour has occurred.

8

Strings: A Deeper Look



Objectives

In this chapter, you'll:

- Understand text processing.
- Use string methods.
- Format string content.
- Concatenate and repeat strings.
- Strip whitespace from the ends of strings.
- Change characters from lowercase to uppercase and vice versa.
- Compare strings with the comparison operators.
- Search strings for substrings and replace substrings.
- Split strings into tokens.
- Concatenate strings into a single string with a specified separator between items.
- Create and use regular expressions to match patterns in strings, replace substrings and validate data.
- Use regular expression metacharacters, quantifiers, character classes and grouping.
- Understand how critical string manipulations are to natural language processing.
- Understand the data science terms data munging, data wrangling and data cleaning, and use regular expressions to munge data into preferred formats.

Outline

8.1 Introduction	8.10 Characters and Character-Testing Methods
8.2 Formatting Strings	8.11 Raw Strings
8.2.1 Presentation Types	8.12 Introduction to Regular Expressions
8.2.2 Field Widths and Alignment	8.12.1 <code>re</code> Module and Function <code>fullmatch</code>
8.2.3 Numeric Formatting	8.12.2 Replacing Substrings and Splitting Strings
8.2.4 String's <code>format</code> Method	8.12.3 Other Search Functions; Accessing Matches
8.3 Concatenating and Repeating Strings	8.13 Intro to Data Science: Pandas, Regular Expressions and Data Munging
8.4 Stripping Whitespace from Strings	8.14 Wrap-Up
8.5 Changing Character Case	Exercises
8.6 Comparison Operators for Strings	
8.7 Searching for Substrings	
8.8 Replacing Substrings	
8.9 Splitting and Joining Strings	

8.1 Introduction

We've introduced strings, basic string formatting and several string operators and methods. You saw that strings support many of the same sequence operations as lists and tuples, and that strings, like tuples, are immutable. Now, we take a deeper look at strings and introduce regular expressions and the `re` module, which we'll use to match patterns¹ in text. Regular expressions are particularly important in today's data rich applications. The capabilities you learn here will help you prepare for the "Natural Language Processing (NLP)" chapter and other key data science chapters. In the NLP chapter, we'll look at other ways to have computers manipulate and even "understand" text. The table below shows many string-processing and NLP-related applications. In the Intro to Data Science section, we briefly introduce data cleaning/munging/wrangling with Pandas `Series` and `DataFrames`.

String and NLP applications

Anagrams	Inter-language translation	Spam classification
Automated grading of written homework	Legal document preparation	Speech-to-text engines
Automated teaching systems	Monitoring social media posts	Spell checkers
Categorizing articles	Natural language understanding	Steganography
Chatbots	Opinion analysis	Text editors
Compilers and interpreters	Page-composition software	Text-to-speech engines
Creative writing	Palindromes	Web scraping
Cryptography	Parts-of-speech tagging	Who authored Shakespeare's works?
Document classification	Project Gutenberg free books	Word clouds
Document similarity	Reading books, articles, documentation and absorbing knowledge	Word games
Document summarization	Search engines	Writing medical diagnoses from x-rays, scans, blood tests
Electronic book readers	Sentiment analysis	and many more...
Fraud detection		
Grammar checkers		

1. We'll see in the data science case study chapters that searching for patterns in text is a crucial part of machine learning.

8.2 Formatting Strings

Proper text formatting makes data easier to read and understand. Here, we present many text-formatting capabilities.

8.2.1 Presentation Types

You've seen basic string formatting with f-strings. When you specify a placeholder for a value in an f-string, Python assumes the value should be displayed as a string unless you specify another type. In some cases, the type is required. For example, let's format the float value 17.489 rounded to the hundredths position:

```
In [1]: f'{17.489:.2f}'  
Out[1]: '17.49'
```

Python supports precision *only* for floating-point and Decimal values. Formatting is *type dependent*—if you try to use .2f to format a string like 'hello', a ValueError occurs. So the **presentation type** f in the *format specifier* .2f is required. It indicates what type is being formatted so Python can determine whether the other formatting information is allowed for that type. Here, we show some common presentation types. You can view the complete list at

<https://docs.python.org/3/library/string.html#formatspec>

Integers

The **d presentation type** formats integer values as strings:

```
In [2]: f'{10:d}'  
Out[2]: '10'
```

There also are integer presentation types (b, o and x or X) that format integers using the binary, octal or hexadecimal number systems.²

Characters

The **c presentation type** formats an integer character code as the corresponding character:

```
In [3]: f'{65:c} {97:c}'  
Out[3]: 'A a'
```

Strings

The **s presentation type** is the default. If you specify s explicitly, the value to format must be a variable that references a string, an expression that produces a string or a string literal, as in the first placeholder below. If you do not specify a presentation type, as in the second placeholder below, non-string values like the integer 7 are converted to strings:

```
In [4]: f'{"hello":s} {7}'  
Out[4]: 'hello 7'
```

In this snippet, "hello" is enclosed in double quotes. Recall that you cannot place single quotes inside a single-quoted string.

2. See the online appendix “Number Systems” for information about the binary, octal and hexadecimal number systems.

Floating-Point and Decimal Values

You've used the `f` presentation type to format floating-point and `Decimal` values. For extremely large and small values of these types, **Exponential (scientific) notation** can be used to format the values more compactly. Let's show the difference between `f` and `e` for a large value, each with three digits of precision to the right of the decimal point:

```
In [5]: from decimal import Decimal

In [6]: f'{Decimal("10000000000000000000000000000000.0"):.3f}'
Out[6]: '10000000000000000000000000000000.000'

In [7]: f'{Decimal("10000000000000000000000000000000.0"):.3e}'
Out[7]: '1.000e+25'
```

For the **e presentation type** in snippet [5], the formatted value `1.000e+25` is equivalent to

$$1.000 \times 10^{25}$$

If you prefer a capital E for the exponent, use the **E presentation type** rather than e.



Self Check

- 1 (Fill-In)** Presentation types _____ and _____ format floating-point and `Decimal` values in scientific notation.

Answer: e, E.

- 2 (Fill-In)** Presentation type _____ formats a character code as its corresponding character.

Answer: c.

- 3 (IPython Session)** Use the type specifier c to display the characters that correspond to the character codes 58, 45 and 41.

Answer:

```
In [1]: print(f'{58:c}{45:c}{41:c}')
:-)
```

8.2.2 Field Widths and Alignment

Previously you used *field widths* to format text in a specified number of character positions. By default, Python *right-aligns* numbers and *left-aligns* other values such as strings—we enclose the results below in brackets ([]) so you can see how the values align in the field:

```
In [1]: f'[{27:10d}]'
Out[1]: '['          27]'

In [2]: f'[{3.5:10f}]'
Out[2]: '[ 3.500000]'

In [3]: f'[{ "hello":10}]'
Out[3]: '[hello      ]'
```

Snippet [2] shows that Python formats `float` values with six digits of precision to the right of the decimal point by default. For values that have fewer characters than the field width, the remaining character positions are filled with spaces. Values with more characters than the field width use as many character positions as they need.

Explicitly Specifying Left and Right Alignment in a Field

Recall that you can specify left and right alignment with < and >:

```
In [4]: f'{27:<15d}'  
Out[4]: '[27           ]'  
  
In [5]: f'{3.5:<15f}'  
Out[5]: '[3.500000    ]'  
  
In [6]: f'{"hello":>15}'  
Out[6]: '[        hello]'
```

Centering a Value in a Field

In addition, you can *center* values:

```
In [7]: f'{27:^7d}'  
Out[7]: '[ 27  ]'  
  
In [8]: f'{3.5:^7.1f}'  
Out[8]: '[ 3.5 ]'  
  
In [9]: f'{"hello":^7}'  
Out[9]: '[ hello ]'
```

Centering attempts to spread the remaining unoccupied character positions equally to the left and right of the formatted value. Python places the extra space to the right if an odd number of character positions remain.



Self Check

- 1** (*True/False*) If you do not specify the alignment, all values displayed in a field are right aligned by default.

Answer: False. Only numeric values are right aligned by default.

- 2** (*IPython Session*) Display on separate lines the name 'Amanda' right-, center- and left-aligned in a field of 10 characters. Enclose each result in brackets so you can see the alignment results more clearly.

Answer:

```
In [1]: print(f'{"Amanda":>10}\n{"Amanda":^10}\n{"Amanda":<10}')  
[     Amanda]  
[ Amanda   ]  
[Amanda     ]
```

8.2.3 Numeric Formatting

There are a variety of numeric formatting capabilities.

Formatting Positive Numbers with Signs

Sometimes it's desirable to force the sign on a positive number:

```
In [1]: f'{27:+10d}'  
Out[1]: '[      +27]'
```

The + before the field width specifies that a positive number should be preceded by a +. A negative number always starts with a -. To fill the remaining characters of the field with 0s rather than spaces, place a 0 before the field width (and *after* the + if there is one):

```
In [2]: f'{27:+010d}'
Out[2]: '+000000027'
```

Using a Space Where a + Sign Would Appear in a Positive Value

A space indicates that positive numbers should show a space character in the sign position. This is useful for aligning positive and negative values for display purposes:

```
In [3]: print(f'{27:d}\n{27: d}\n{-27: d}')
27
27
-27
```

Note that the two numbers with a space in their format specifiers align. If a field width is specified, the space should appear *before* the field width.

Grouping Digits

You can format numbers with **thousands separators** by using a **comma (,)**, as follows:

```
In [4]: f'{12345678:,d}'
Out[4]: '12,345,678'

In [5]: f'{123456.78:,.2f}'
Out[5]: '123,456.78'
```



Self Check

- 1** (*Fill-In*) To display all numeric values with their sign, use a(n) _____ in the format specifier; to display a space rather than a sign for positive values use a(n) _____ instead.
Answer: +, space character.

- 2** (*IPython Session*) Print the values 10240.473 and -3210.9521, each preceded by its sign, in 10-character fields with thousands separators, their decimal points aligned vertically and two digits of precision.

Answer:

```
In [1]: print(f'{10240.473:+10,.2f}\n{-3210.9521:+10,.2f}')
+10,240.47
-3,210.95
```

8.2.4 String's format Method

Python's f-strings were added to the language in version 3.6. Before that, formatting was performed with the string method **format**. In fact, f-string formatting is based on the **format** method's capabilities. We show you the **format** method here because you'll encounter it in code written prior to Python 3.6. You'll often see the **format** method in the Python documentation and in the many Python books and articles written before f-strings were introduced. However, we recommend using the newer f-string formatting that we've presented to this point.

You call method **format** on a *format string* containing curly brace **{}** *placeholders*, possibly with format specifiers. You pass to the method the values to be formatted. Let's format the float value 17.489 rounded to the hundredths position:

```
In [1]: '{:.2f}'.format(17.489)
Out[1]: '17.49'
```

In a placeholder, if there's a format specifier, you precede it by a colon (:), as in f-strings. The result of the format call is a new string containing the formatted results.

Multiple Placeholders

A format string may contain multiple placeholders, in which case the `format` method's arguments correspond to the placeholders from left to right:

```
In [2]: '{} {}'.format('Amanda', 'Cyan')
Out[2]: 'Amanda Cyan'
```

Referencing Arguments By Position Number

The format string can reference specific arguments by their position in the `format` method's argument list, starting with position 0:

```
In [3]: '{0} {0} {1}'.format('Happy', 'Birthday')
Out[3]: 'Happy Happy Birthday'
```

Note that we used the position number 0 ('Happy') twice—you can reference each argument as often as you like and in any order.

Referencing Keyword Arguments

You can reference keyword arguments by their keys in the placeholders:

```
In [4]: '{first} {last}'.format(first='Amanda', last='Gray')
Out[4]: 'Amanda Gray'

In [5]: '{last} {first}'.format(first='Amanda', last='Gray')
Out[5]: 'Gray Amanda'
```



Self Check

- (IPython Session)* Use string method `format` to reimplement the IPython sessions in the Self Check exercises from Sections 8.2.1–8.2.3.

Answer:

```
In [1]: print('{:c}{:c}{:c}'.format(58, 45, 41))
:-)

In [2]: print('{0:>10}\n{0:^10}\n{0:<10}'.format('Amanda'))
[     Amanda]
[   Amanda  ]
[Amanda      ]

In [3]: print('{:+10,.2f}\n{:+10,.2f}'.format(10240.473, -3210.9521))
+10,240.47
-3,210.95
```

Note that snippet [2] references `format`'s argument three times via its position number (0) in the argument list.

8.3 Concatenating and Repeating Strings

In earlier chapters, we used the `+` operator to concatenate strings and the `*` operator to repeat strings. You also can perform these operations with augmented assignments. Strings are immutable, so each operation assigns a new string object to the variable:

```
In [1]: s1 = 'happy'
In [2]: s2 = 'birthday'
In [3]: s1 += ' ' + s2
In [4]: s1
Out[4]: 'happy birthday'
In [5]: symbol = '>'
In [6]: symbol *= 5
In [7]: symbol
Out[7]: '>>>>'
```



Self Check

- I (*IPython Session*) Use the `+=` operator to concatenate your first and last name. Then use the `*=` operator to create a bar of asterisks with the same number of characters as your full name and display the bar above and below your name.

Answer:

```
In [1]: name = 'Pam'
In [2]: name += ' Black'
In [3]: bar = '*'
In [4]: bar *= len(name)
In [5]: print(f'{bar}\n{name}\n{bar}')
*****
Pam Black
*****
```

8.4 Stripping Whitespace from Strings

There are several string methods for removing whitespace from the ends of a string. Each returns a new string leaving the original unmodified. Strings are immutable, so each method that appears to modify a string returns a new one.

Removing Leading and Trailing Whitespace

Let's use string method `strip` to remove the leading and trailing whitespace from a string:

```
In [1]: sentence = '\t \n This is a test string. \t\t \n'
In [2]: sentence.strip()
Out[2]: 'This is a test string.'
```

Removing Leading Whitespace

Method `lstrip` removes only leading whitespace:

```
In [3]: sentence.lstrip()
Out[3]: 'This is a test string. \t\t \n'
```

Removing Trailing Whitespace

Method `rstrip` removes only trailing whitespace:

```
In [4]: sentence.rstrip()
Out[4]: '\t \n This is a test string.'
```

As the outputs demonstrate, these methods remove all kinds of whitespace, including spaces, newlines and tabs.



Self Check

I (IPython Session) Use the methods in this section to strip the whitespace from the following string, which has five spaces at the beginning and end of the string:

```
name = '      Margo Magenta      '
```

Answer:

```
In [1]: name = '      Margo Magenta      '
In [2]: name.strip()
Out[2]: 'Margo Magenta'
In [3]: name.lstrip()
Out[3]: 'Margo Magenta      '
In [4]: name.rstrip()
Out[4]: '      Margo Magenta'
```

8.5 Changing Character Case

In earlier chapters, you used string methods `lower` and `upper` to convert strings to all lowercase or all uppercase letters. You also can change a string's capitalization with methods `capitalize` and `title`.

Capitalizing Only a String's First Character

Method `capitalize` copies the original string and returns a new string with only the first letter capitalized (this is sometimes called *sentence capitalization*):

```
In [1]: 'happy birthday'.capitalize()
Out[1]: 'Happy birthday'
```

Capitalizing the First Character of Every Word in a String

Method `title` copies the original string and returns a new string with only the first character of each word capitalized (this is sometimes called *book-title capitalization*):

```
In [2]: 'strings: a deeper look'.title()
Out[2]: 'Strings: A Deeper Look'
```



Self Check

I (IPython Session) Demonstrate the results of calling `capitalize` and `title` on the string 'happy new year'.

Answer:

```
In [1]: test_string = 'happy new year'
In [2]: test_string.capitalize()
Out[2]: 'Happy new year'
In [3]: test_string.title()
Out[3]: 'Happy New Year'
```

8.6 Comparison Operators for Strings

Strings may be compared with the comparison operators. Recall that strings are compared based on their underlying integer numeric values. So uppercase letters compare as less than lowercase letters because uppercase letters have lower integer values. For example, 'A' is 65 and 'a' is 97. You've seen that you can check character codes with `ord`:

```
In [1]: print(f'A: {ord("A")}; a: {ord("a")}')
A: 65; a: 97
```

Let's compare the strings 'Orange' and 'orange' using the comparison operators:

```
In [2]: 'Orange' == 'orange'
Out[2]: False

In [3]: 'Orange' != 'orange'
Out[3]: True

In [4]: 'Orange' < 'orange'
Out[4]: True

In [5]: 'Orange' <= 'orange'
Out[5]: True

In [6]: 'Orange' > 'orange'
Out[6]: False

In [7]: 'Orange' >= 'orange'
Out[7]: False
```

8.7 Searching for Substrings

You can search in a string for one or more adjacent characters—known as a *substring*—to count the number of occurrences, determine whether a string contains a substring, or determine the index at which a substring resides in a string. Each method shown in this section compares characters lexicographically using their underlying numeric values.

Counting Occurrences

String method `count` returns the number of times its argument occurs in the string on which the method is called:

```
In [1]: sentence = 'to be or not to be that is the question'
In [2]: sentence.count('to')
Out[2]: 2
```

If you specify as the second argument a `start_index`, `count` searches only the slice `string[start_index:]`—that is, from `start_index` through end of the string:

```
In [3]: sentence.count('to', 12)
Out[3]: 1
```

If you specify as the second and third arguments the *start_index* and *end_index*, `count` searches only the slice `string[start_index:end_index]`—that is, from *start_index* up to, but not including, *end_index*:

```
In [4]: sentence.count('that', 12, 25)
Out[4]: 1
```

Like `count`, each of the other string methods presented in this section has *start_index* and *end_index* arguments for searching only a slice of the original string.

Locating a Substring in a String

String method `index` searches for a substring within a string and returns the first index at which the substring is found; otherwise, a `ValueError` occurs:

```
In [5]: sentence.index('be')
Out[5]: 3
```

String method `rindex` performs the same operation as `index`, but searches from the end of the string and returns the *last* index at which the substring is found; otherwise, a `ValueError` occurs:

```
In [6]: sentence.rindex('be')
Out[6]: 16
```

String methods `find` and `rfind` perform the same tasks as `index` and `rindex` but, if the substring is not found, return `-1` rather than causing a `ValueError`.

Determining Whether a String Contains a Substring

If you need to know only whether a string contains a substring, use operator `in` or `not in`:

```
In [7]: 'that' in sentence
Out[7]: True
```

```
In [8]: 'THAT' in sentence
Out[8]: False
```

```
In [9]: 'THAT' not in sentence
Out[9]: True
```

Locating a Substring at the Beginning or End of a String

String methods `startswith` and `endswith` return `True` if the string starts with or ends with a specified substring:

```
In [10]: sentence.startswith('to')
Out[10]: True
```

```
In [11]: sentence.startswith('be')
Out[11]: False
```

```
In [12]: sentence.endswith('question')
Out[12]: True
```

```
In [13]: sentence.endswith('quest')
Out[13]: False
```

✓ Self Check

1 (*Fill-In*) Method _____ returns the number of times a given substring occurs in a string.

Answer: count.

2 (*True/False*) String method `find` causes a `ValueError` if it does not find the specified substring.

Answer: False. String method `find` returns -1 in this case. String method `index` causes a `ValueError`.

3 (*IPython Session*) Create a loop that locates and displays every word that starts with 't' in the string 'to be or not to be that is the question'.

Answer:

```
In [1]: for word in 'to be or not to be that is the question'.split():
...:     if word.startswith('t'):
...:         print(word, end=' ')
...:
to to that the
```

8.8 Replacing Substrings

A common text manipulation is to locate a substring and replace its value. Method `replace` takes two substrings. It searches a string for the substring in its first argument and replaces *each* occurrence with the substring in its second argument. The method returns a new string containing the results. Let's replace tab characters with commas:

```
In [1]: values = '1\t2\t3\t4\t5'
```

```
In [2]: values.replace('\t', ',')
Out[2]: '1,2,3,4,5'
```

Method `replace` can receive an optional third argument specifying the maximum number of replacements to perform.

✓ Self Check

1 (*IPython Session*) Replace the spaces in the string '1 2 3 4 5' with ' --> '.

Answer:

```
In [1]: '1 2 3 4 5'.replace(' ', ' --> ')
Out[1]: '1 --> 2 --> 3 --> 4 --> 5'
```

8.9 Splitting and Joining Strings

When you read a sentence, your brain breaks it into individual words, or **tokens**, each of which conveys meaning. Interpreters like IPython tokenize statements, breaking them into individual components such as keywords, identifiers, operators and other elements of a programming language. Tokens typically are separated by whitespace characters such as blank, tab and newline, though other characters may be used—the separators are known as **delimiters**.

Splitting Strings

We showed previously that string method `split` with *no* arguments tokenizes a string by breaking it into substrings at each whitespace character, then returns a list of tokens. To tokenize a string at a custom delimiter (such as each comma-and-space pair), specify the delimiter string (such as, `,`, `'`) that `split` uses to tokenize the string:

```
In [1]: letters = 'A, B, C, D'
```

```
In [2]: letters.split(',')
Out[2]: ['A', 'B', 'C', 'D']
```

If you provide an integer as the second argument, it specifies the maximum number of splits. The last token is the remainder of the string after the maximum number of splits:

```
In [3]: letters.split(',', 2)
Out[3]: ['A', 'B', 'C, D']
```

There is also an `rsplit` method that performs the same task as `split` but processes the maximum number of splits from the end of the string toward the beginning.

Joining Strings

String method `join` concatenates the strings in its argument, which must be an iterable containing only string values; otherwise, a `TypeError` occurs. The separator between the concatenated items is the string on which you call `join`. The following code creates strings containing comma-separated lists of values:

```
In [4]: letters_list = ['A', 'B', 'C', 'D']

In [5]: ','.join(letters_list)
Out[5]: 'A,B,C,D'
```

The next snippet joins the results of a list comprehension that creates a list of strings:

```
In [6]: ','.join([str(i) for i in range(10)])
Out[6]: '0,1,2,3,4,5,6,7,8,9'
```

In the “Files and Exceptions” chapter, you’ll see how to work with files that contain comma-separated values. These are known as **CSV files** and are a common format for storing data that can be loaded by spreadsheet applications like Microsoft Excel or Google Sheets. In the data science case study chapters, you’ll see that many key libraries, such as NumPy, Pandas and Seaborn, provide built-in capabilities for working with CSV data.

String Methods `partition` and `rpartition`

String method `partition` splits a string into a tuple of three strings based on the method’s *separator* argument. The three strings are

- the part of the original string before the separator,
- the separator itself, and
- the part of the string after the separator.

This might be useful for splitting more complex strings. Consider a string representing a student’s name and grades:

```
'Amanda: 89, 97, 92'
```

Let's split the original string into the student's name, the separator ': ' and a string representing the list of grades:

```
In [7]: 'Amanda: 89, 97, 92'.partition(': ')
Out[7]: ('Amanda', ': ', '89, 97, 92')
```

To search for the separator from the end of the string instead, use method `rpartition` to split. For example, consider the following URL string:

```
'http://www.deitel.com/books/PyCDS/table_of_contents.html'
```

Let's use `rpartition` split 'table_of_contents.html' from the rest of the URL:

```
In [8]: url = 'http://www.deitel.com/books/PyCDS/table_of_contents.html'

In [9]: rest_of_url, separator, document = url.rpartition('/')

In [10]: document
Out[10]: 'table_of_contents.html'

In [11]: rest_of_url
Out[11]: 'http://www.deitel.com/books/PyCDS'
```

String Method `splittines`

In the “Files and Exceptions” chapter, you’ll read text from a file. If you read large amounts of text into a string, you might want to split the string into a list of lines based on newline characters. Method `splittines` returns a list of new strings representing the lines of text split at each newline character in the original string. Recall that Python stores multiline strings with embedded \n characters to represent the line breaks, as shown in snippet [13]:

```
In [12]: lines = """This is line 1
...: This is line2
...: This is line3"""

In [13]: lines
Out[13]: 'This is line 1\nThis is line2\nThis is line3'

In [14]: lines.splitlines()
Out[14]: ['This is line 1', 'This is line2', 'This is line3']
```

Passing True to `splittines` keeps the newlines at the end of each string:

```
In [15]: lines.splitlines(True)
Out[15]: ['This is line 1\n', 'This is line2\n', 'This is line3']
```



Self Check

- 1 (Fill-In)** Tokens are separated from one another by _____.
Answer: delimiters.

- 2 (IPython Session)** Use `split` and `join` in one statement to reformat the string
`'Pamela White'`

into the string

```
'White, Pamela'
```

Answer:

```
In [1]: ', '.join(reversed('Pamela White'.split()))
Out[1]: 'White, Pamela'
```

- 3** (*IPython Session*) Use `partition` and `rpartition` to extract from the URL string

```
'http://www.deitel.com/books/PyCDS/table_of_contents.html'
```

the substrings 'www.deitel.com' and 'books/PyCDS'.

Answer:

```
In [2]: url = 'http://www.deitel.com/books/PyCDS/table_of_contents.html'

In [3]: protocol, separator, rest_of_url = url.partition('://')

In [4]: host, separator, document_with_path = rest_of_url.partition('/')

In [5]: host
Out[5]: 'www.deitel.com'

In [6]: path, separator, document = document_with_path.rpartition('/')

In [7]: path
Out[7]: 'books/PyCDS'
```

8.10 Characters and Character-Testing Methods

Characters (digits, letters and symbols such as \$, @, % and *) are the fundamental building blocks of programs. Every program is composed of characters that, when grouped meaningfully, represent instructions and data that the interpreter uses to perform tasks. Many programming languages have separate string and character types. In Python, a character is simply a one-character string.

Python provides string methods for testing whether a string matches certain characteristics. For example, string method `isdigit` returns True if the string on which you call the method contains only the digit characters (0–9). You might use this when validating user input that must contain only digits:

```
In [1]: '-27'.isdigit()
Out[1]: False

In [2]: '27'.isdigit()
Out[2]: True
```

and the string method `isalnum` returns True if the string on which you call the method is alphanumeric—that is, it contains only digits and letters:

```
In [3]: 'A9876'.isalnum()
Out[3]: True

In [4]: '123 Main Street'.isalnum()
Out[4]: False
```

The table below shows many of the character-testing methods. Each method returns `False` if the condition described is not satisfied:

String Method	Description
<code>isalnum()</code>	Returns <code>True</code> if the string contains only <i>alphanumeric</i> characters (i.e., digits and letters).
<code>isalpha()</code>	Returns <code>True</code> if the string contains only <i>alphabetic</i> characters (i.e., letters).
<code>isdecimal()</code>	Returns <code>True</code> if the string contains only <i>decimal integer</i> characters (that is, base 10 integers) and does not contain a + or - sign.
<code>isdigit()</code>	Returns <code>True</code> if the string contains only digits (e.g., '0', '1', '2').
<code>isidentifier()</code>	Returns <code>True</code> if the string represents a valid <i>identifier</i> .
<code>islower()</code>	Returns <code>True</code> if all alphabetic characters in the string are <i>lowercase</i> characters (e.g., 'a', 'b', 'c').
<code>isnumeric()</code>	Returns <code>True</code> if the characters in the string represent a <i>numeric value</i> without a + or - sign and without a decimal point.
<code>isspace()</code>	Returns <code>True</code> if the string contains only <i>whitespace</i> characters.
<code>istitle()</code>	Returns <code>True</code> if the first character of each word in the string is the only <i>uppercase</i> character in the word.
<code>isupper()</code>	Returns <code>True</code> if all alphabetic characters in the string are <i>uppercase</i> characters (e.g., 'A', 'B', 'C').



Self Check

1 (Fill-In) Method _____ returns `True` if a string contains only letters and numbers.
Answer: `isalnum`.

2 (Fill-In) Method _____ returns `True` if a string contains only letters.
Answer: `isalpha`.

8.11 Raw Strings

Recall that backslash characters in strings introduce *escape sequences*—like `\n` for newline and `\t` for tab. So, if you wish to include a backslash in a string, you must use two backslash characters `\\"`. This makes some strings difficult to read. For example, Microsoft Windows uses backslashes to separate folder names when specifying a file's location. To represent a file's location on Windows, you might write:

```
In [1]: file_path = 'C:\\MyFolder\\\\MySubFolder\\\\MyFile.txt'

In [2]: file_path
Out[2]: 'C:\\MyFolder\\\\MySubFolder\\\\MyFile.txt'
```

For such cases, **raw strings**—preceded by the character `r`—are more convenient. They treat each backslash as a regular character, rather than the beginning of an escape sequence:

```
In [3]: file_path = r'C:\\MyFolder\\\\MySubFolder\\\\MyFile.txt'

In [4]: file_path
Out[4]: 'C:\\MyFolder\\\\MySubFolder\\\\MyFile.txt'
```

Python converts the raw string to a regular string that still uses the two backslash characters in its internal representation, as shown in the last snippet. Raw strings can make your code more readable, particularly when using the regular expressions that we discuss in the next section. Regular expressions often contain many backslash characters.



Self Check

I (Fill-In) The raw string `r'\\Hi!\\'` represents the regular string _____.

Answer: '`\\\\\\Hi!\\\\\\'`'.

8.12 Introduction to Regular Expressions

Sometimes you'll need to recognize *patterns* in text, like phone numbers, e-mail addresses, ZIP Codes, web page addresses, Social Security numbers and more. A **regular expression** string describes a *search pattern* for *matching* characters in other strings.

Regular expressions can help you extract data from unstructured text, such as social media posts. They're also important for ensuring that data is in the correct format before you attempt to process it.³

Validating Data

Before working with text data, you'll often use regular expressions to *validate the data*. For example, you can check that:

- A U.S. ZIP Code consists of five digits (such as 02215) or five digits followed by a hyphen and four more digits (such as 02215-4775).
- A string last name contains only letters, spaces, apostrophes and hyphens.
- An e-mail address contains only the allowed characters in the allowed order.
- A U.S. Social Security number contains three digits, a hyphen, two digits, a hyphen and four digits, and adheres to other rules about the specific numbers that can be used in each group of digits.

You'll rarely need to create your own regular expressions for common items like these. Websites like

- <https://regex101.com>
- <http://www.regexlib.com>
- <https://www.regular-expressions.info>

and others offer repositories of existing regular expressions that you can copy and use. Many sites like these also provide interfaces in which you can test regular expressions to determine whether they'll meet your needs. We ask you to do this in the exercises.

3. The topic of regular expressions might feel more challenging than most other Python features you've used. After mastering this subject, you'll often write more concise code than with conventional string-processing techniques, speeding the code-development process. You'll also deal with "fringe" cases you might not ordinarily think about, possibly avoiding subtle bugs.

Other Uses of Regular Expressions

In addition to validating data, regular expressions often are used to:

- Extract data from text (sometimes known as *scraping*)—For example, locating all URLs in a web page. [You might prefer tools like BeautifulSoup, XPath and lxml.]
- Clean data—For example, removing data that's not required, removing duplicate data, handling incomplete data, fixing typos, ensuring consistent data formats, dealing with outliers and more.
- Transform data into other formats—For example, reformatting data that was collected as tab-separated or space-separated values into comma-separated values (CSV) for an application that requires data to be in CSV format.

8.12.1 re Module and Function fullmatch

To use regular expressions, import the Python Standard Library's `re` module:

```
In [1]: import re
```

One of the simplest regular expression functions is `fullmatch`, which checks whether the *entire* string in its second argument matches the pattern in its first argument.

Matching Literal Characters

Let's begin by matching *literal characters*—that is, characters that match themselves:

```
In [2]: pattern = '02215'
In [3]: 'Match' if re.fullmatch(pattern, '02215') else 'No match'
Out[3]: 'Match'

In [4]: 'Match' if re.fullmatch(pattern, '51220') else 'No match'
Out[4]: 'No match'
```

The function's first argument is the regular expression pattern to match. Any string can be a regular expression. The variable `pattern`'s value, '`02215`', contains only *literal digits* that match *themselves* in the specified order. The second argument is the string that should entirely match the pattern.

If the second argument matches the pattern in the first argument, `fullmatch` returns an object containing the matching text, which evaluates to `True`. We'll say more about this object later. In snippet [4], even though the second argument contains the *same digits* as the regular expression, they're in a *different* order. So there's no match, and `fullmatch` returns `None`, which evaluates to `False`.

Metacharacters, Character Classes and Quantifiers

Regular expressions typically contain various special symbols called `metacharacters`, which are shown in the table below:

Regular expression metacharacters

[]	{ }	{}()	\	*	+	^	\$?	.	
-----	-----	------	---	---	---	---	----	---	---	--

The `\ metacharacter` begins each of the predefined `character classes`, each matching a specific set of characters. Let's validate a five-digit ZIP Code:

```
In [5]: 'Valid' if re.fullmatch(r'\d{5}', '02215') else 'Invalid'
Out[5]: 'Valid'
```

```
In [6]: 'Valid' if re.fullmatch(r'\d{5}', '9876') else 'Invalid'
Out[6]: 'Invalid'
```

In the regular expression `\d{5}`, `\d` is a character class representing a digit (0–9). A character class is a *regular expression escape sequence* that matches *one* character. To match more than one, follow the character class with a **quantifier**. The quantifier `{5}` repeats `\d` five times, as if we had written `\d\d\d\d\d`, to match five consecutive digits. In snippet [6], `fullmatch` returns `None` because '9876' contains only four consecutive digit characters.

Other Predefined Character Classes

The table below shows some common predefined character classes and the groups of characters they match. To match any metacharacter as its *literal* value, precede it by a backslash (\). For example, `\\\` matches a backslash (\) and `\$` matches a dollar sign (\$).

Character class	Matches
<code>\d</code>	Any digit (0–9).
<code>\D</code>	Any character that is <i>not</i> a digit.
<code>\s</code>	Any whitespace character (such as spaces, tabs and newlines).
<code>\S</code>	Any character that is <i>not</i> a whitespace character.
<code>\w</code>	Any word character (also called an alphanumeric character)—that is, any uppercase or lowercase letter, any digit or an underscore
<code>\W</code>	Any character that is <i>not</i> a word character.

Custom Character Classes

Square brackets, `[]`, define a **custom character class** that matches a *single* character. For example, `[aeiou]` matches a lowercase vowel, `[A-Z]` matches an uppercase letter, `[a-z]` matches a lowercase letter and `[a-zA-Z]` matches any lowercase or uppercase letter.

Let's validate a simple first name with no spaces or punctuation. We'll ensure that it begins with an uppercase letter (A–Z) followed by any number of lowercase letters (a–z):

```
In [7]: 'Valid' if re.fullmatch('[A-Z][a-z]*', 'Wally') else 'Invalid'
Out[7]: 'Valid'
```

```
In [8]: 'Valid' if re.fullmatch('[A-Z][a-z]*', 'eva') else 'Invalid'
Out[8]: 'Invalid'
```

A first name might contain many letters. The `*` **quantifier** matches *zero or more occurrences* of the subexpression to its left (in this case, `[a-z]`). So `[A-Z][a-z]*` matches an uppercase letter followed by *zero or more* lowercase letters, such as 'Amanda', 'Bo' or even 'E'.

When a custom character class starts with a **caret** (`\^`), the class matches any character that's *not* specified. So `[\^a-z]` matches any character that's *not* a lowercase letter:

```
In [9]: 'Match' if re.fullmatch('[\^a-z]', 'A') else 'No match'
Out[9]: 'Match'
```

```
In [10]: 'Match' if re.fullmatch('[\^a-z]', 'a') else 'No match'
Out[10]: 'No match'
```

Metacharacters in a custom character class are treated as literal characters—that is, the characters themselves. So `[*+$]` matches a *single* *, + or \$ character:

```
In [11]: 'Match' if re.fullmatch('[*+$]', '*') else 'No match'
Out[11]: 'Match'
```

```
In [12]: 'Match' if re.fullmatch('[*+$]', '!') else 'No match'
Out[12]: 'No match'
```

* vs. + Quantifier

If you want to require *at least one* lowercase letter in a first name, you can replace the * quantifier in snippet [7] with +, which matches *at least one occurrence* of a subexpression:

```
In [13]: 'Valid' if re.fullmatch('[A-Z][a-z]+', 'Wally') else 'Invalid'
Out[13]: 'Valid'
```

```
In [14]: 'Valid' if re.fullmatch('[A-Z][a-z]+', 'E') else 'Invalid'
Out[14]: 'Invalid'
```

Both * and + are **greedy**—they match as many characters as possible. So the regular expression `[A-Z][a-z]+` matches 'A1', 'Eva', 'Samantha', 'Benjamin' and any other words that begin with a capital letter followed at least one lowercase letter.

Other Quantifiers

The **? quantifier** matches *zero or one occurrences* of a subexpression:

```
In [15]: 'Match' if re.fullmatch('label1?ed', 'labelled') else 'No match'
Out[15]: 'Match'
```

```
In [16]: 'Match' if re.fullmatch('label1?ed', 'labeled') else 'No match'
Out[16]: 'Match'
```

```
In [17]: 'Match' if re.fullmatch('label1?ed', 'label1led') else 'No
match'
Out[17]: 'No match'
```

The regular expression `label1?ed` matches `labelled` (the U.K. English spelling) and `labeled` (the U.S. English spelling), but not the misspelled word `label1led`. In each snippet above, the first five literal characters in the regular expression (`label`) match the first five characters of the second arguments. Then `1?` indicates that there can be *zero or one more* 1 characters before the remaining literal `ed` characters.

You can match *at least n occurrences* of a subexpression with the **{n,} quantifier**. The following regular expression matches strings containing *at least* three digits:

```
In [18]: 'Match' if re.fullmatch(r'\d{3,}', '123') else 'No match'
Out[18]: 'Match'
```

```
In [19]: 'Match' if re.fullmatch(r'\d{3,}', '1234567890') else 'No match'
Out[19]: 'Match'
```

```
In [20]: 'Match' if re.fullmatch(r'\d{3,}', '12') else 'No match'
Out[20]: 'No match'
```

You can match *between n and m (inclusive)* occurrences of a subexpression with the **{n,m} quantifier**. The following regular expression matches strings containing 3 to 6 digits:

```
In [21]: 'Match' if re.fullmatch(r'\d{3,6}', '123') else 'No match'
Out[21]: 'Match'

In [22]: 'Match' if re.fullmatch(r'\d{3,6}', '123456') else 'No match'
Out[22]: 'Match'

In [23]: 'Match' if re.fullmatch(r'\d{3,6}', '1234567') else 'No match'
Out[23]: 'No match'

In [24]: 'Match' if re.fullmatch(r'\d{3,6}', '12') else 'No match'
Out[24]: 'No match'
```



Self Check

1 (*True/False*) Any string can be a regular expression.

Answer: True.

2 (*True/False*) The ? quantifier matches *exactly one* occurrence of a subexpression.

Answer: False. The ? quantifier matches *zero or one* occurrences of a subexpression.

3 (*True/False*) The character class [^0–9] matches any digit.

Answer: False. The character class [^0–9] matches anything that is *not* a digit.

4 (*IPython Session*) Create and test a regular expression that matches a street address consisting of a number with one or more digits followed by two words of one or more characters each. The tokens should be separated by one space each, as in 123 Main Street.

Answer:

```
In [1]: import re

In [2]: street = r'\d+ [A-Z][a-z]* [A-Z][a-z]*'

In [3]: 'Match' if re.fullmatch(street, '123 Main Street') else 'No match'
Out[3]: 'Match'

In [4]: 'Match' if re.fullmatch(street, 'Main Street') else 'No match'
Out[4]: 'No match'
```

8.12.2 Replacing Substrings and Splitting Strings

The `re` module provides function `sub` for replacing patterns in a string, and function `split` for breaking a string into pieces, based on patterns.

Function `sub`—Replacing Patterns

By default, the `re` module's **sub function** replaces *all* occurrences of a pattern with the replacement text you specify. Let's convert a tab-delimited string to comma-delimited:

```
In [1]: import re

In [2]: re.sub(r'\t', ', ', '1\t2\t3\t4')
Out[2]: '1, 2, 3, 4'
```

The `sub` function receives three required arguments:

- the *pattern to match* (the tab character '\t')
- the *replacement text* (', ') and
- the *string to be searched* ('1\t2\t3\t4')

and returns a new string. The keyword argument `count` can be used to specify the maximum number of replacements:

```
In [3]: re.sub(r'\t', ' ', '1\t2\t3\t4', count=2)
Out[3]: '1, 2, 3\t4'
```

Function `split`

The `split` function tokenizes a string, using a regular expression to specify the *delimiter*, and returns a list of strings. Let's tokenize a string by splitting it at any comma that's followed by 0 or more whitespace characters—`\s*` is the whitespace character class and `*` indicates *zero or more* occurrences of the preceding subexpression:

```
In [4]: re.split(r',\s*', '1, 2, 3,4, 5,6,7,8')
Out[4]: ['1', '2', '3', '4', '5', '6', '7', '8']
```

Use the keyword argument `maxsplit` to specify the maximum number of splits:

```
In [5]: re.split(r',\s*', '1, 2, 3,4, 5,6,7,8', maxsplit=3)
Out[5]: ['1', '2', '3', '4, 5,6,7,8']
```

In this case, after the 3 splits, the fourth string contains the rest of the original string.



Self Check

- 1** (*IPython Session*) Replace each occurrence of one or more adjacent tab characters in the following string with a comma and a space:

```
'A\tB\t\tC\t\tD'
```

Answer:

```
In [1]: import re

In [2]: re.sub(r'\t+', ' ', 'A\tB\t\tC\t\tD')
Out[2]: 'A, B, C, D'
```

- 2** (*IPython Session*) Use a regular expression and the `split` function to split the following string at *one or more* adjacent \$ characters.

```
'123$Main$$Street'
```

Answer:

```
In [3]: re.split('\$+', '123$Main$$Street')
Out[3]: ['123', 'Main', 'Street']
```

8.12.3 Other Search Functions; Accessing Matches

Earlier we used the `fullmatch` function to determine whether an *entire* string matched a regular expression. There are several other searching functions. Here, we discuss the `search`, `match`, `.findall` and `finditer` functions, and show how to access the matching substrings.

Function `search`—Finding the First Match Anywhere in a String

Function `search` looks in a string for the *first* occurrence of a substring that matches a regular expression and returns a `match object` (of type `SRE_Match`) that contains the matching substring. The match object's `group` method returns that substring:

```
In [1]: import re  
In [2]: result = re.search('Python', 'Python is fun')  
In [3]: result.group() if result else 'not found'  
Out[3]: 'Python'
```

Function `search` returns `None` if the string does *not* contain the pattern:

```
In [4]: result2 = re.search('fun!', 'Python is fun')  
In [5]: result2.group() if result2 else 'not found'  
Out[5]: 'not found'
```

You can search for a match only at the *beginning* of a string with function `match`.

Ignoring Case with the Optional `flags` Keyword Argument

Many `re` module functions receive an optional `flags` keyword argument that changes how regular expressions are matched. For example, matches are *case sensitive* by default, but by using the `re` module's `IGNORECASE` constant, you can perform a *case-insensitive* search:

```
In [6]: result3 = re.search('Sam', 'SAM WHITE', flags=re.IGNORECASE)  
In [7]: result3.group() if result3 else 'not found'  
Out[7]: 'SAM'
```

Here, 'SAM' matches the pattern 'Sam' because both have the same letters, even though 'SAM' contains only uppercase letters.

Metacharacters That Restrict Matches to the Beginning or End of a String

The **`^` metacharacter** at the beginning of a regular expression (and not inside square brackets) is an anchor indicating that the expression matches only the *beginning* of a string:

```
In [8]: result = re.search('^Python', 'Python is fun')  
In [9]: result.group() if result else 'not found'  
Out[9]: 'Python'  
  
In [10]: result = re.search('^fun', 'Python is fun')  
In [11]: result.group() if result else 'not found'  
Out[11]: 'not found'
```

Similarly, the **`$` metacharacter** at the end of a regular expression is an anchor indicating that the expression matches only the *end* of a string:

```
In [12]: result = re.search('Python$', 'Python is fun')  
In [13]: result.group() if result else 'not found'  
Out[13]: 'not found'  
  
In [14]: result = re.search('fun$', 'Python is fun')  
In [15]: result.group() if result else 'not found'  
Out[15]: 'fun'
```

Function `findall` and `finditer`—Finding All Matches in a String

Function `findall` finds *every* matching substring in a string and returns a list of the matching substrings. Let's extract all the U.S. phone numbers from a string. For simplicity we'll assume that U.S. phone numbers have the form `###-###-####`:

```
In [16]: contact = 'Wally White, Home: 555-555-1234, Work: 555-555-4321'
In [17]: re.findall(r'\d{3}-\d{3}-\d{4}', contact)
Out[17]: ['555-555-1234', '555-555-4321']
```

Function `finditer` works like `findall`, but returns a lazy *iterable* of match objects. For large numbers of matches, using `finditer` can save memory because it returns one match at a time, whereas `findall` returns all the matches at once:

```
In [18]: for phone in re.finditer(r'\d{3}-\d{3}-\d{4}', contact):
    ...:     print(phone.group())
    ...:
555-555-1234
555-555-4321
```

Capturing Substrings in a Match

You can use **parentheses metacharacters**—(and)—to capture substrings in a match. For example, let's capture as separate substrings the name and e-mail address in the string `text`:

```
In [19]: text = 'Charlie Cyan, e-mail: demo1@deitel.com'
In [20]: pattern = r'([A-Z][a-z]+ [A-Z][a-z]+), e-mail: (\w+@\w+\.\w{3})'
In [21]: result = re.search(pattern, text)
```

The regular expression specifies two substrings to capture, each denoted by the metacharacters (and). These metacharacters do *not* affect whether the pattern is found in the string `text`—the `match` function returns a match object *only* if the *entire* pattern is found in the string `text`.

Let's consider the regular expression:

- '`([A-Z][a-z]+ [A-Z][a-z]+)`' matches two words separated by a space. Each word must have an initial capital letter.
- '`, e-mail:`' contains literal characters that match themselves.
- '`(\w+@\w+\.\w{3})`' matches a *simple* e-mail address consisting of one or more alphanumeric characters (`\w+`), the @ character, one or more alphanumeric characters (`\w+`), a dot (`\.`) and three alphanumeric characters (`\w{3}`). We preceded the dot with \ because a dot (.) is a regular expression metacharacter that matches one character.

The match object's `groups` method returns a tuple of the captured substrings:

```
In [22]: result.groups()
Out[22]: ('Charlie Cyan', 'demo1@deitel.com')
```

The match object's `group` method returns the *entire* match as a single string:

```
In [23]: result.group()
Out[23]: 'Charlie Cyan, e-mail: demo1@deitel.com'
```

You can access each captured substring by passing an integer to the `group` method. The captured substrings are *numbered from 1* (unlike list indices, which start at 0):

```
In [24]: result.group(1)
Out[24]: 'Charlie Cyan'

In [25]: result.group(2)
Out[25]: 'demo1@deitel.com'
```



Self Check

1 (*Fill-In*) Function _____ finds in a string the first substring that matches a regular expression.

Answer: search.

2 (*IPython Session*) Assume you have a string representing an addition problem such as
`'10 + 5'`

Use a regular expression to break the string into three groups representing the two operands and the operator, then display the groups.

Answer:

```
In [1]: import re

In [2]: result = re.search(r'(\d+) ([+-*/]) (\d+)', '10 + 5')

In [3]: result.groups()
Out[3]: ('10', '+', '5')

In [4]: result.group(1)
Out[4]: '10'

In [5]: result.group(2)
Out[5]: '+'

In [6]: result.group(3)
Out[6]: '5'
```

8.13 Intro to Data Science: Pandas, Regular Expressions and Data Munging

Data does not always come in forms ready for analysis. It could, for example, be in the wrong format, incorrect or even missing. Industry experience has shown that data scientists can spend as much as 75% of their time preparing data before they begin their studies. Preparing data for analysis is called **data munging** or **data wrangling**. These are synonyms—from this point forward, we'll say data munging.

Two of the most important steps in data munging are *data cleaning* and *transforming data* into the optimal formats for your database systems and analytics software. Some common data cleaning examples are:

- deleting observations with missing values,
- substituting reasonable values for missing values,
- deleting observations with bad values,
- substituting reasonable values for bad values,
- tossing outliers (although sometimes you'll want to keep them),
- duplicate elimination (although sometimes duplicates are valid),
- dealing with inconsistent data,
- and more.

You're probably already thinking that data cleaning is a difficult and messy process where you could easily make bad decisions that would negatively impact your results. This is correct. When you get to the data science case studies in the later chapters, you'll see that data science is more of an **empirical science**, like medicine, and less of a theoretical science, like theoretical physics. Empirical sciences base their conclusions on observations and experience. For example, many medicines that effectively solve medical problems today were developed by observing the effects that early versions of these medicines had on lab animals and eventually humans, and gradually refining ingredients and dosages. The actions data scientists take can vary per project, be based on the quality and nature of the data and be affected by evolving organization and professional standards.

Some common data transformations include:

- removing unnecessary data and *features* (we'll say more about features in the data science case studies),
- combining related features,
- sampling data to obtain a representative subset (we'll see in the data science case studies that *random sampling* is particularly effective for this and we'll say why),
- standardizing data formats,
- grouping data,
- and more.

It's always wise to hold onto your original data. We'll show simple examples of cleaning and transforming data in the context of Pandas `Series` and `DataFrames`.

Cleaning Your Data

Bad data values and missing values can significantly impact data analysis. Some data scientists advise against any attempts to insert "reasonable values." Instead, they advocate clearly marking missing data and leaving it up to the data analytics package to handle the issue. Others offer strong cautions.⁴

Let's consider a hospital that records patients' temperatures (and probably other vital signs) four times per day. Assume that the data consists of a name and four `float` values, such as

```
[ 'Brown, Sue' , 98.6 , 98.4 , 98.7 , 0.0 ]
```

The preceding patient's first three recorded temperatures are 99.7, 98.4 and 98.7. The last temperature was missing and recorded as 0.0, perhaps because the sensor malfunctioned. The average of the first three values is 98.57, which is close to normal. However, if you calculate the average temperature *including* the missing value for which 0.0 was sub-

4. This footnote was abstracted from a comment sent to us July 20, 2018 by one of the book's academic reviewers, Dr. Alison Sanchez of the University of San Diego School of Business. She commented: "Be cautious when mentioning 'substituting reasonable values' for missing or bad values.' A stern warning: 'Substituting' values that increase statistical significance or give more 'reasonable' or 'better' results is not permitted. 'Substituting' data should not turn into 'fudging' data. The first rule students should learn is not to eliminate or change values that contradict their hypotheses. 'Substituting reasonable values' does not mean students should feel free to change values to get the results they want."

stituted, the average is only 73.93, clearly a questionable result. Certainly, doctors would not want to take drastic remedial action on this patient—it's crucial to “get the data right.”

One common way to clean the data is to substitute a *reasonable* value for the missing temperature, such as the average of the patient's other readings. Had we done that above, then the patient's average temperature would remain 98.57—a much more likely average temperature, based on the other readings.

Data Validation

Let's begin by creating a *Series* of five-digit ZIP Codes from a dictionary of city-name/five-digit-ZIP-Code key–value pairs. We intentionally entered an invalid ZIP Code for Miami:

```
In [1]: import pandas as pd

In [2]: zips = pd.Series({'Boston': '02215', 'Miami': '3310'})

In [3]: zips
Out[3]:
Boston      02215
Miami      3310
dtype: object
```

Though *zips* looks like a two-dimensional array, it's actually one-dimensional. The “second column” represents the *Series*' ZIP Code *values* (from the dictionary's values), and the “first column” represents their *indices* (from the dictionary's keys).

We can use regular expressions with Pandas to validate data. The **str attribute** of a *Series* provides string-processing and various regular expression methods. Let's use the *str* attribute's **match method** to check whether each ZIP Code is valid:

```
In [4]: zips.str.match(r'\d{5}')
Out[4]:
Boston      True
Miami      False
dtype: bool
```

Method *match* applies the regular expression `\d{5}` to *each Series element*, attempting to ensure that the element is comprised of exactly five digits. You do not need to loop explicitly through all the ZIP Codes—*match* does this for you. This is another example of functional-style programming with internal rather than external iteration. The method returns a new *Series* containing `True` for each valid element. In this case, the ZIP Code for Miami did *not* match, so its element is `False`.

There are several ways to deal with invalid data. One is to catch it at its source and interact with the source to correct the value. That's not always possible. For example, the data could be coming from high-speed sensors in the Internet of Things. In that case, we would not be able to correct it at the source, so we could apply data cleaning techniques. In the case of the bad Miami ZIP Code of 3310, we might look for Miami ZIP Codes beginning with 3310. There are two—33101 and 33109—and we could pick one of those.

Sometimes, rather than matching an *entire* value to a pattern, you'll want to know whether a value contains a *substring* that matches the pattern. In this case, use method **contains** instead of *match*. Let's create a *Series* of strings, each containing a U.S. city, state and ZIP Code, then determine whether each string contains a substring matching the pattern '`[A-Z]{2}` ' (a space, followed by two uppercase letters, followed by a space):

```
In [5]: cities = pd.Series(['Boston, MA 02215', 'Miami, FL 33101'])

In [6]: cities
Out[6]:
0    Boston, MA 02215
1    Miami, FL 33101
dtype: object

In [7]: cities.str.contains(r' [A-Z]{2} ')
Out[7]:
0    True
1    True
dtype: bool

In [8]: cities.str.match(r' [A-Z]{2} ')
Out[8]:
0    False
1    False
dtype: bool
```

We did not specify the index values, so the Series uses zero-based indexes by default (snippet [6]). Snippet [7] uses contains to show that both Series elements contain substrings that match '`[A-Z]{2}`'. Snippet [8] uses match to show that neither element's value matches that pattern in its entirety, because each has other characters in its complete value.

Reformatting Your Data

We've discussed data cleaning. Now let's consider munging data into a different format. As a simple example, assume that an application requires U.S. phone numbers in the format `###-###-####`, with hyphens separating each group of digits. The phone numbers have been provided to us as 10-digit strings without hyphens. Let's create the DataFrame:

```
In [9]: contacts = [['Mike Green', 'demo1@deitel.com', '5555555555'],
...:                 ['Sue Brown', 'demo2@deitel.com', '5555551234']]
...:

In [10]: contactsdf = pd.DataFrame(contacts,
...:                               columns=['Name', 'Email', 'Phone'])
...:

In [11]: contactsdf
Out[11]:
      Name        Email     Phone
0  Mike Green  demo1@deitel.com  5555555555
1   Sue Brown  demo2@deitel.com  5555551234
```

In this DataFrame, we specified column indices via the `columns` keyword argument but did *not* specify row indices, so the rows are indexed from 0. Also, the output shows the column values right aligned by default. This differs from Python formatting in which numbers in a field are *right aligned* by default but non-numeric values are *left aligned* by default.

Now, let's munge the data with a little more functional-style programming. We can *map* the phone numbers to the proper format by calling the Series method `map` on the DataFrame's 'Phone' column. Method `map`'s argument is a *function* that receives a value and returns the *mapped* value. The function `get_formatted_phone` maps 10 consecutive digits into the format `###-###-###`:

```
In [12]: import re  
  
In [13]: def get_formatted_phone(value):  
...:     result = re.fullmatch(r'(\d{3})(\d{3})(\d{4})', value)  
...:     return '-'.join(result.groups()) if result else value  
...:  
...:
```

The regular expression in the block's first statement matches *only* 10 consecutive digits. It captures substrings containing the first three digits, the next three digits and the last four digits. The `return` statement operates as follows:

- If `result` is `None`, we simply return `value` unmodified.
- Otherwise, we call `result.groups()` to get a tuple containing the captured substrings and pass that tuple to string method `join` to concatenate the elements, separating each from the next with '-' to form the mapped phone number.

`Series` method `map` returns a new `Series` containing the results of calling its function argument for each value in the column. Snippet [15] displays the result, including the column's name and type:

```
In [14]: formatted_phone = contactsdf['Phone'].map(get_formatted_phone)  
  
In [15]: formatted_phone  
0      555-555-5555  
1      555-555-1234  
Name: Phone, dtype: object
```

Once you've confirmed that the data is in the correct format, you can update it in the original `DataFrame` by assigning the new `Series` to the 'Phone' column:

```
In [16]: contactsdf['Phone'] = formatted_phone  
  
In [17]: contactsdf  
Out[17]:  
        Name           Email       Phone  
0  Mike Green  demo1@deitel.com  555-555-5555  
1  Sue Brown  demo2@deitel.com  555-555-1234
```

We'll continue our pandas discussion in the next chapter's Intro to Data Science section, and we'll use pandas in several later chapters.



Self Check

1 (*Fill-In*) Preparing data for analysis is called _____ or _____. A subset of this process is data cleaning.

Answer: data munging, data wrangling.

2 (*IPython Session*) Let's assume that an application requires U.S. phone numbers in the format `(###) ###-####`. Modify the `get_formatted_phone` function in snippet [13] to return the phone number in this new format. Then recreate the `DataFrame` from snippets [9] and [10] and use the updated `get_formatted_phone` function to munge the data.

Answer:

```
In [1]: import pandas as pd  
  
In [2]: import re
```

```
In [3]: contacts = [['Mike Green', 'demo1@deitel.com', '5555555555'],
...:                  ['Sue Brown', 'demo2@deitel.com', '5555551234']]
...:

In [4]: contactsdf = pd.DataFrame(contacts,
...:                               columns=['Name', 'Email', 'Phone'])
...:

In [5]: def get_formatted_phone(value):
...:     result = re.fullmatch(r'(\d{3})(\d{3})(\d{4})', value)
...:     if result:
...:         part1, part2, part3 = result.groups()
...:         return '(' + part1 + ')' + part2 + '-' + part3
...:     else:
...:         return value
...:

In [6]: contactsdf['Phone'] = contactsdf['Phone'].map(get_formatted_phone)

In [7]: contactsdf
Out[7]:
      Name           Email       Phone
0  Mike Green  demo1@deitel.com  (555) 555-5555
1  Sue Brown  demo2@deitel.com  (555) 555-1234
```

8.14 Wrap-Up

In this chapter, we presented various string formatting and processing capabilities. You formatted data in f-strings and with the string method `format`. We showed the augmented assignments for concatenating and repeating strings. You used string methods to remove whitespace from the beginning and end of strings and to change their case. We discussed additional methods for splitting strings and for joining iterables of strings. We introduced various character-testing methods.

We showed raw strings that treat backslashes (\) as literal characters rather than the beginning of escape sequences. These were particularly useful for defining regular expressions, which often contain many backslashes.

Next, we introduced the powerful pattern-matching capabilities of regular expressions with functions from the `re` module. We used the `fullmatch` function to ensure that an entire string matched a pattern, which is useful for validating data. We showed how to use the `replace` function to search for and replace substrings. We used the `split` function to tokenize strings based on delimiters that match a regular expression pattern. Then we showed various ways to search for patterns in strings and to access the resulting matches.

In the Intro to Data Science section, we introduced the synonyms data munging and data wrangling and showed an sample data munging operation, namely and transforming data. We continued our discussion of Panda's `Series` and `DataFrames` by using regular expressions to validate and munge data.

In the next chapter, we'll continue using various string-processing capabilities as we introduce reading text from files and writing text to files. We'll introduce the `csv` module for manipulating comma-separated value (CSV) files. We'll also introduce exception handling so we can process exceptions as they occur, rather than displaying a traceback.

Exercises

Use IPython sessions for each exercise where practical.

8.1 (*Check Protection*) Although electronic deposit has become extremely popular, payroll and accounts payable applications often print checks. A serious problem is the intentional alteration of a check amount by someone who plans to cash a check fraudulently. To prevent a dollar amount from being altered, some computerized check-writing systems employ a technique called *check protection*. Checks designed for printing by computer typically contain a fixed number of spaces for the printed amount. Suppose a paycheck contains eight blank spaces in which the computer is supposed to print the amount of a weekly paycheck. If the amount is large, then all eight of the spaces will be filled:

```
1,230.60 (check amount)
-----
01234567 (position numbers)
```

On the other hand, if the amount is smaller, then several of the spaces would ordinarily be left blank. For example,

```
399.87
-----
01234567
```

contains two blank spaces. If a check is printed with blank spaces, it's easier for someone to alter the amount. Check-writing systems often insert *leading asterisks* to prevent alteration and protect the amount as follows:

```
**399.87
-----
01234567
```

Write a script that inputs a dollar amount, then prints the amount in check-protected format in a field of 10 characters with leading asterisks if necessary. [Hint: In a format string that explicitly specifies alignment with <, ^ or >, you can precede the alignment specifier with the fill character of your choice.]

8.2 (*Random Sentences*) Write a script that uses random-number generation to compose sentences. Use four arrays of strings called `article`, `noun`, `verb` and `preposition`. Create a sentence by selecting a word at random from each array in the following order: `article`, `noun`, `verb`, `preposition`, `article` and `noun`. As each word is picked, concatenate it to the previous words in the sentence. Spaces should separate the words. When the final sentence is output, it should start with a capital letter and end with a period. The script should generate and display 20 sentences.

8.3 (*Pig Latin*) Write a script that encodes English-language phrases into a form of coded language called pig Latin. There are many different ways to form pig Latin phrases. For simplicity, use the following algorithm:

To form a pig Latin phrase from an English-language phrase, tokenize the phrase into words with string method `split`. To translate each English word into a pig Latin word, place the first letter of the English word at the end of the word and add the letters "ay." Thus, the word "jump" becomes "umpjay," the word "the" becomes "hetay," and the word "computer" becomes "omputercay." If the word starts with a vowel, just add "ay." Blanks between words remain as blanks. Assume the following: The English phrase con-

sists of words separated by blanks, there are no punctuation marks and all words have two or more letters. Enable the user to enter a sentence, then display the sentence in pig Latin.

8.4 (Reversing a Sentence) Write a script that reads a line of text as a string, tokenizes the string with the `split` method and outputs the tokens in reverse order. Use space characters as delimiters.

8.5 (Tokenizing and Comparing Strings) Write a script that reads a line of text, tokenizes the line using space characters as delimiters and outputs only those words beginning with the letter 'b'.

8.6 (Tokenizing and Comparing Strings) Write a script that reads a line of text, tokenizes it using space characters as delimiters and outputs only those words ending with the letters 'ed'.

8.7 (Converting Integers to Characters) Use the `c` presentation type to display a table of the character codes in the range 0 to 255 and their corresponding characters.

8.8 (Converting Integers to Emojis) Modify the previous exercise to display 10 emojis beginning with the smiley face, which has the value `0x1F600`:⁵



The value `0x1F600` is a hexadecimal (base 16) integer. See the online appendix “Number Systems” for information on the hexadecimal number system. You can find emoji codes by searching online for “Unicode full emoji list.” The Unicode website precedes each character code with “U+” (representing Unicode). Replace “U+” with “0x” to properly format the code as a Python hexadecimal integer.

8.9 (Creating Three-Letter Strings from a Five-Letter Word) Write a script that reads a five-letter word from the user and produces every possible three-letter string, based on the word’s letters. For example, the three-letter words produced from the word “bathe” include “ate,” “bat,” “bet,” “tab,” “hat,” “the” and “tea.” *Challenge:* Investigate the functions from the `itertools` module, then use an appropriate function to automate this task.

8.10 (Project: Simple Sentiment Analysis) Search online for lists of positive sentiment words and negative sentiment words. Create a script that inputs text, then determines whether that text is positive or negative, based on the total number of positive words and the total number of negative words. Test your script by searching for Twitter tweets on a topic of your choosing, then entering the text for several tweets. In the data science case study chapters, we’ll take a deeper look at sentiment analysis.

8.11 (Project: Evaluate Word Problems) Write a script that enables the user to enter mathematical word problems like “two times three” and “seven minus five”, then use string processing to break apart the string into the numbers and the operation and return the result. So “two times three” would return 6 and “seven minus five” would return 2. To keep things simple, assume the user enters only the words for the numbers 0 through 9 and only the operations ‘plus’, ‘minus’, ‘times’ and ‘divided by’.

5. The look-and-feel of emojis varies across systems. The emoji shown here is from macOS. Also, depending on your system’s fonts the emoji symbols might not display correctly.

8.12 (*Project: Scrambled Text*) Use string-processing capabilities to keep the first and last letter of a word and scramble the remaining letters in between the first and last. Search online for “University of Cambridge scrambled text” for an intriguing paper on the readability of texts consisting of such scrambled words. Investigate the `random` module’s `shuffle` function to help you implement this exercise’s solution.

Regular Expression Exercises

8.13 (*Regular Expressions: Condense Spaces to a Single Space*) Check whether a sentence contains more than one space between words. If so, remove the extra spaces and display the results. For example, ‘Hello World’ should become ‘Hello World’.

8.14 (*Regular Expressions: Capturing Substrings*) Reimplement Exercises 8.5 and 8.6 using regular expressions that capture the matching substrings, then display them.

8.15 (*Regular Expressions: Counting Characters and Words*) Use regular expressions and the `findall` function to count the number of digits, non-digit characters, whitespace characters and words in a string.

8.16 (*Regular Expressions: Locating URLs*) Use a regular expression to search through a string and to locate all valid URLs. For this exercise, assume that a valid URL has the form `http://www.domain_name.extension`, where `extension` must be two or more characters.

8.17 (*Regular Expressions: Matching Numeric Values*) Write a regular expression that searches a string and matches a valid number. A number can have any number of digits, but it can have only digits and a decimal point and possibly a leading sign. The decimal point is optional, but if it appears in the number, there must be only one, and it must have digits on its left and its right. There should be whitespace or a beginning or end-of-line character on either side of a valid number.

8.18 (*Regular Expression: Password Format Validator*) Search online for secure password recommendations, then research existing regular expressions that validate secure passwords. Two examples of password requirements are:

- Passwords must contain at least five words, each separated by a hyphen, a space, a period, a comma or an underscore.
- Passwords must have a minimum of 8 characters and contain at least one each from uppercase characters, lowercase characters, digits and punctuation characters (such as characters in ‘!@#\$%^&*?’).

Write regular expressions for each of the two requirements above, then use them to test sample passwords.

8.19 (*Regular Expressions: Testing Regular Expressions Online*) Before using any regular expression in your code, you should thoroughly test it to ensure that it meets your needs. Use a regular expression website like `regex101.com` to explore and test existing regular expressions, then write your own regular expression tester.

8.20 (*Regular Expressions: Munging Dates*) Dates are stored and displayed in several common formats. Three common formats are

042555
04/25/1955
April 25, 1955

Use regular expressions to search a string containing dates, find substrings that match these formats and munge them into the other formats. The original string should have one date in each format, so there will be a total of six transformations.

8.21 (*Project: Metric Conversions*) Write a script that assists the user with some common metric-to-English conversions. Your script should allow the user to specify the names of the units as strings (i.e., centimeters, liters, grams, and so on for the metric system and inches, quarts, pounds, and so on for the English system) and should respond to simple questions, such as

```
'How many inches are in 2 meters?'
'How many liters are in 10 quarts?'
```

Your script should recognize invalid conversions. For example, the following question is not meaningful, because 'feet' is a unit of length and 'kilograms' is a unit of mass:

```
'How many feet are in 5 kilograms?'
```

Assume that all questions are in the form shown above. Use regular expressions to capture the important substrings, such as 'inches', '2' and 'meters' in the first sample question above. Recall that functions `int` and `float` can convert strings to numbers.

More Challenging String-Manipulation Exercises

The preceding exercises are keyed to the text and designed to test your understanding of fundamental string manipulation and regular expression concepts. This section includes a collection of intermediate and advanced string-manipulation exercises. You should find these problems challenging, yet entertaining. The problems vary considerably in difficulty. Some require an hour or two of coding. Others are useful for lab assignments that might require two or three weeks of study and implementation. Some are challenging term projects. In the “Natural Language Processing (NLP)” chapter, you’ll learn other text-processing techniques that will enable you to approach some of these exercises from a machine learning perspective.

8.22 (*Project: Cooking with Healthier Ingredients*) In the “Dictionaries and Sets” chapter’s exercises, you created a dictionary that mapped ingredients to lists of their possible substitutions. Use that dictionary in a script that helps users choose healthier ingredients when cooking. The script should read a recipe from the user and suggest healthier replacements for some of the ingredients. For simplicity, your script should assume the recipe has no abbreviations for measures such as teaspoons, cups, and tablespoons, and uses numerical digits for quantities (e.g., 1 egg, 2 cups) rather than spelling them out (one egg, two cups). Your program should display a warning such as, “Always consult your healthcare professional before making significant changes to your diet.” Your program should take into consideration that replacements are not always one-for-one. For example, each whole egg in a recipe can be replaced with two egg whites.

8.23 (*Project: Spam Scanner*) Spam (or junk e-mail) costs U.S. organizations billions of dollars a year in spam-prevention software, equipment, network resources, bandwidth, and lost productivity. Research online some of the most common spam e-mail messages and words, and check your junk e-mail folder. Create a list of 30 words and phrases commonly found in spam messages. Write an application in which the user enters an e-mail message. Then, scan the message for each of the 30 keywords or phrases. For each occur-

rence of one of these within the message, add a point to the message's "spam score." Next, rate the likelihood that the message is spam, based on the number of points it received. In the data science case study chapters, you'll be able to attack this problem in a more sophisticated way.

8.24 (Research: Inter-Language Translation) This exercise will help you explore one of the most challenging problems in natural language processing and artificial intelligence. The Internet brings us all together in ways that make inter-language translation particularly important. As authors, we frequently receive messages from non-English speaking readers worldwide. Not long ago, we'd write back asking them to write to us in English so we could understand.

With advances in machine learning, artificial intelligence and natural language processing, services like Google Translate (100+ languages) and Bing Microsoft Translator (60+ languages) can translate between languages instantly. In fact, the translations are so good that when non-English speakers write to us in English, we often ask them to write back in their native language, then we translate their message online.

There are many challenges in natural language translation. To get a sense of this, use online translation services to perform the following tasks:

- a) Start with a sentence in English. A popular sentence in machine translation lore is from the Bible's Matthew 26:41, "The spirit is willing, but the flesh is weak."
- b) Translate that sentence to another language, like Japanese.
- c) Translate the Japanese text back to English.

Do you get the original sentence? Often, translating from one language to another and back gives the original sentence or something close. Try chaining multiple language translations together. For instance, we took the phrase in Part (a) above and translated it from English to Chinese Traditional to Japanese to Arabic and back to English. The result was, "The soul is very happy, but the flesh is very crisp." Send us your favorite translations!

8.25 (Project: State of the Union Speeches) All U.S. Presidents' State of the Union speeches are available online. Copy and paste one into a large multiline string, then display statistics, including the total word count, the total character count, the average word length, the average sentence length, a word distribution of all words, a word distribution of words ending in 'ly' and the top 10 longest words. In the "Natural Language Processing (NLP)" chapter, you'll find lots of more sophisticated techniques for analyzing and comparing such texts.

8.26 (Research: Grammarly) Copy and paste State of the Union speeches into the free version of Grammarly or similar software. Compare the reading grade levels for speeches from several presidents.

