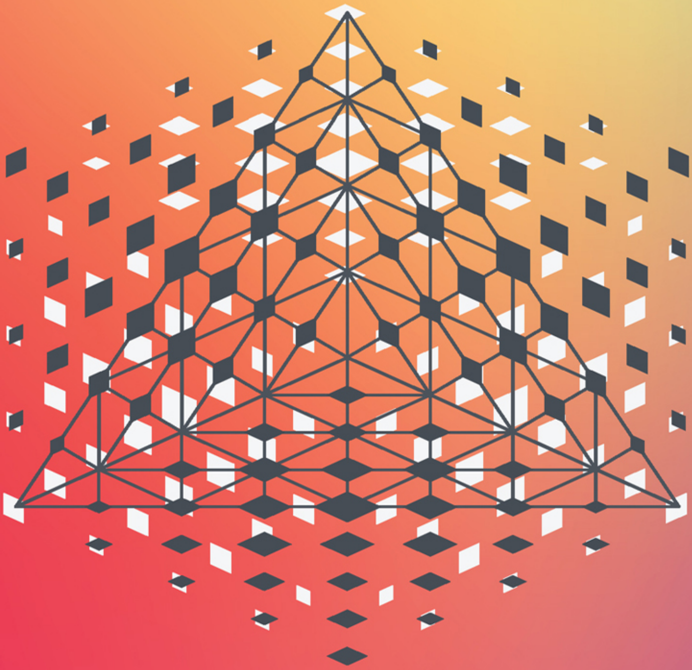
Addison Wesley Data & Analytics Series

Foundational Python for Data Science



Kennedy Behrman

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

Foundational Python for Data Science

This page intentionally left blank

Foundational Python for Data Science

Kennedy R. Behrman

Addison-Wesley

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

Many of the designations used 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 with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

Visit us on the Web: informit.com/aw.

Library of Congress Control Number: 2021940284

Copyright © 2022 Pearson Education, Inc.

All rights reserved. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit www.pearson.com/permissions/.

ISBN-13: 978-0-13-662435-6
ISBN-10: 0-13-662435-9

ScoutAutomatedPrintCode

Editor-in-Chief
Mark Taub

Acquisitions Editor
Malobika Chakraborty

Development Editor
Mark Renfrow

Managing Editor
Sandra Schroeder

Senior Project Editor
Lori Lyons

Copy Editor
Kitty Wilson

Production Manager
Aswini Kumar/codeMantra

Indexer
Timothy Wright

Proofreader
Abigail Manheim

Compositor
codeMantra

❖

This book is dedicated to Tatiana, Itta, and Maple,
who is probably still under the bed.

❖

Contents at a Glance

Preface xiii

I: Learning Python in a Notebook Environment 1

1 Introduction to Notebooks **3**

2 Fundamentals of Python **13**

3 Sequences **25**

4 Other Data Structures **37**

5 Execution Control **55**

6 Functions **67**

II: Data Science Libraries 83

7 NumPy **85**

8 SciPy **103**

9 Pandas **113**

10 Visualization Libraries **135**

11 Machine Learning Libraries **153**

12 Natural Language Toolkit **159**

III: Intermediate Python 171

13 Functional Programming **173**

14 Object-Oriented Programming **187**

15 Other Topics **201**

A Answers to End-of-Chapter Questions 215

Index **221**

Table of Contents

Preface xiii

I: Learning Python in a Notebook Environment 1

1 Introduction to Notebooks 3

Running Python Statements 4

Jupyter Notebooks 4

Google Colab 5

- Colab Text Cells 6
- Colab Code Cells 9
- Colab Files 9
- Managing Colab Documents 10
- Colab Code Snippets 11
- Existing Collections 11
- System Aliases 11
- Magic Functions 12

Summary 12

Questions 12

2 Fundamentals of Python 13

Basic Types in Python 14

- High-Level Versus Low-Level Languages 15
- Statements 15

Performing Basic Math Operations 21

Using Classes and Objects with Dot Notation 22

Summary 22

Questions 23

3 Sequences 25

Shared Operations 25

- Testing Membership 26
- Indexing 26
- Slicing 27
- Interrogation 27
- Math Operations 28

Lists and Tuples 29

Creating Lists and Tuples 29
Adding and Removing List Items 30
Unpacking 31
Sorting Lists 32
Strings 32
Ranges 34
Summary 35
Questions 35

4 Other Data Structures 37
Dictionaries 37
Creating Dictionaries 38
Accessing, Adding, and Updating by Using Keys 38
Removing Items from Dictionaries 39
Dictionary Views 40
Checking to See If a Dictionary Has a Key 43
The `get` Method 43
Valid Key Types 44
The `hash` Method 45
Sets 46
Set Operations 48
Frozensets 53
Summary 53
Questions 53

5 Execution Control 55
Compound Statements 55
Compound Statement Structure 56
Evaluating to `True` or `False` 56
`if` Statements 59
`while` Loops 62
`for` Loops 63
`break` and `continue` Statements 64
Summary 65
Questions 65

6 Functions 67
Defining Functions 67
Control Statement 68

Docstrings 68
Parameters 69
Return Statements 75
Scope in Functions 75
Decorators 76
Anonymous Functions 80
Summary 81
Questions 81

II: Data Science Libraries 83

7 NumPy 85
Installing and Importing NumPy 86
Creating Arrays 86
Indexing and Slicing 89
Element-by-Element Operations 91
Filtering Values 92
Views Versus Copies 94
Some Array Methods 95
Broadcasting 98
NumPy Math 100
Summary 102
Questions 102

8 SciPy 103
SciPy Overview 103
The `scipy.misc` Submodule 104
The `scipy.special` Submodule 105
The `scipy.stats` Submodule 105
Discrete Distributions 105
Continuous Distributions 108
Summary 111
Questions 111

9 Pandas 113
About DataFrames 113
Creating DataFrames 114
Creating a DataFrame from a Dictionary 114

Creating a DataFrame from a List of Lists 115
Creating a DataFrame from a File 116
Interacting with DataFrame Data 117
Heads and Tails 117
Descriptive Statistics 118
Accessing Data 120
Bracket Syntax 121
Optimized Access by Label 123
Optimized Access by Index 124
Masking and Filtering 125
Pandas Boolean Operators 126
Manipulating DataFrames 127
Manipulating Data 129
The `replace` Method 131
Interactive Display 133
Summary 133
Questions 133

10 Visualization Libraries 135
`matplotlib` 135
Styling Plots 137
Labeled Data 140
Plotting Multiple Sets of Data 141
Object-Oriented Style 143
Seaborn 144
Seaborn Themes 145
Plotly 148
Bokeh 149
Other Visualization Libraries 151
Summary 151
Questions 151

11 Machine Learning Libraries 153
Popular Machine Learning Libraries 153
How Machine Learning Works 154
Transformations 154
Splitting Test and Training Data 155
Training and Testing 156

Learning More About Scikit-learn 157
Summary 157
Questions 157

12 Natural Language Toolkit 159
NLTK Sample Texts 159
Frequency Distributions 161
Text Objects 165
Classifying Text 166
Summary 169
Exercises 169

III: Intermediate Python 171

13 Functional Programming 173
Introduction to Functional Programming 173
Scope and State 174
Depending on Global State 174
Changing State 175
Changing Mutable Data 176
Functional Programming Functions 177
List Comprehensions 179
List Comprehension Basic Syntax 179
Replacing `map` and `filter` 180
Multiple Variables 181
Dictionary Comprehensions 181
Generators 182
Generator Expressions 182
Generator Functions 183
Summary 184
Questions 185

14 Object-Oriented Programming 187
Grouping State and Function 187
Classes and Instances 188
Private Methods and Variables 190
Class Variables 190
Special Methods 191
Representation Methods 192

Rich Comparison Methods 192
Math Operator Methods 195
Inheritance 196
Summary 199
Questions 199

15 Other Topics 201
Sorting 201
Lists 201
Reading and Writing Files 204
Context Managers 205
`datetime` Objects 206
Regular Expressions 207
Character Sets 208
Character Classes 209
Groups 209
Named Groups 210
Find All 210
Find Iterator 211
Substitution 211
Substitution Using Named Groups 211
Compiling Regular Expressions 211
Summary 212
Questions 212

A Answers to End-of-Chapter Questions 215

Index 221

Preface

The Python language has been around for a long time and worn many hats. Its original implementation was started by Guido van Rossum in 1989 as a tool for system administration as an alternative to Bash scripts and C programs.[1] Since its public release in 1991, it has evolved for use in a myriad of industries. These include everything from web-development, film, government, science, and business.[2]

I was first introduced to Python working in the film industry, where we used it to automate data management across departments and locations. In the last decade, Python has become a dominant tool in Data Science.

This dominance evolved due to two developments: the Jupyter notebook, and powerful third-party libraries. In 2001 Fernando Perez began the IPython project, an interactive Python environment inspired by Maple and Mathematica notebooks.[3] By 2014, the notebook-specific part of the project was split off as the Jupyter project. These notebooks have excelled for scientific and statistical work environments. In parallel with this development, third-party libraries for scientific and statistical computing were developed for Python. With so many applications, the functionality available to a Python programmer has grown immensely. With specialized packages for everything from opening web sockets to processing natural language text, there is more available than a beginning developer needs.

This project was the brainchild of Noah Gift.[4] In his work as an educator, he found that students of Data Science did not have a resource to learn just the parts of Python they needed. There were many general Python books and books about Data Science, but not resources for learning just the Python needed to get started in Data Science. That is what we have attempted to provide here. This book will not teach the Python needed to set up a web page or perform system administration. It is also not intended to teach you Data Science, but rather the Python needed to learn Data Science.

I hope you will find this guide a good companion in your quest to grow your Data Science knowledge.

Example Code

Most of the code shown in examples in this book can be found on GitHub at: https://github.com/kbehrman/foundational-python-for-data-science.

1 https://docs.python.org/3/faq/general.html#why-was-python-created-in-the-first-place

2 https://www.python.org/success-stories/

3 http://blog.fperez.org/2012/01/ipython-notebook-historical.html

4 https://noahgift.com

Figure Credits

Figure	**Credit Attribution**
Cover	Boris Znaev/Shutterstock
Cover	Mark.G/Shutterstock
Figure 1-01	Screenshot of Colab Dialogue © 2021 Google
Figure 1-02	Screenshot of Renaming Notebook © 2021 Google
Figure 1-03	Screenshot of Google Drive © 2021 Google
Figure 1-04	Screenshot of Editing Text Cells © 2021 Google
Figure 1-05	Screenshot of Formatting Text © 2021 Google
Figure 1-06	Screenshot of Lists © 2021 Google
Figure 1-07	Screenshot of Headings © 2021 Google
Figure 1-08	Screenshot of Table of Contents © 2021 Google
Figure 1-09	Screenshot of Hiding Cells © 2021 Google
Figure 1-10	Screenshot of LaTeX Example © 2021 Google
Figure 1-11	Screenshot of A Files © 2021 Google
Figure 1-12	Screenshot of Upload Files © 2021 Google
Figure 1-13	Screenshot of Mount Google Drive © 2021 Google
Figure 1-14	Screenshot of Code Snippets © 2021 Google

Register Your Book

Register your copy of *Foundational Python for Data Science* on the InformIT site for convenient access to updates and/or corrections as they become available. To start the registration process, go to informit.com/register and log in or create an account. Enter the product ISBN **9780136624356** and click Submit. Look on the Registered Products tab for an Access Bonus Content link next to this product, and follow that link to access any available bonus materials. If you would like to be notified of exclusive offers on new editions and updates, please check the box to receive email from us.

Acknowledgments

The idea for this book first came from Noah Gift. It is he who really identified the need for a specialized introduction to Python targeted at students of Data Science. Thank you for that, Noah. And thank you to Colin Erdman who, acting as technical editor, brought an attention to detail that was much appreciated and needed. I also want to thank the Pearson team, including Malobika Chakraborty, who guided me through the whole process, Mark Renfrow, who came in and helped get the project done, and Laura Lewin, who helped get it going.

About the Author

Kennedy Behrman is a veteran software engineer. He began using Python to manage digital assets in the visual effects industry and has used it extensively since. He has authored various books and training programs around Python education. He currently works as a senior data engineer at Envestnet.

This page intentionally left blank

3

Sequences

Errors using inadequate data are much less than those using no data at all.

Charles Babbage

In This Chapter

- Shared sequence operations
- Lists and tuples
- Strings and string methods
- Ranges

In Chapter 2, "Fundamentals of Python," you learned about collections of types. This chapter introduces the group of built-in types called *sequences*. A sequence is an ordered, finite collection. You might think of a sequence as a shelf in a library, where each book on the shelf has a location and can be accessed easily if you know its place. The books are ordered, with each book (except those at the ends) having books before and after it. You can add books to the shelf, and you can remove them, and it is possible for the shelf to be empty. The built-in types that comprise a sequence are lists, tuples, strings, binary strings, and ranges. This chapter covers the shared characteristics and specifics of these types.

Shared Operations

The sequences family shares quite a bit of functionality. Specifically, there are ways of using sequences that are applicable to most of the group members. There are operations that relate to sequences having a finite length, for accessing the items in a sequence, and for creating a new sequence based a sequence's content.

Testing Membership

You can test whether an item is a member of a sequence by using the `in` operation. This operation returns `True` if the sequence contains an item that evaluates as equal to the item in question, and it returns `False` otherwise. The following are examples of using `in` with different sequence types:

```
'first' in ['first', 'second', 'third']
True

23 in (23,)
True

'b' in 'cat'
False

b'a' in b'ieojjza'
True
```

You can use the keyword `not` in conjunction with `in` to check whether something is absent from a sequence:

```
'b' not in 'cat'
True
```

The two places you are most likely to use `in` and `not in` are in an interactive session to explore data and as part of an `if` statement (see Chapter 5, "Execution Control").

Indexing

Because a sequence is an ordered series of items, you can access an item in a sequence by using its position, or *index*. Indexes start at zero and go up to one less than the number of items. In an eight-item sequence, for example, the first item has an index of zero, and the last item an index of seven.

To access an item by using its index, you use square brackets around the index number. The following example defines a string and accesses its first and last substrings using their index numbers:

```
name = "Ignatius"
name[0]
'I'

name[4]
't'
```

You can also index counting back from the end of a sequence by using negative index numbers:

```
name[-1]
's'

name[-2]
'u'
```

Slicing

You can use indexes to create new sequences that represent subsequences of the original. In square brackets, supply the beginning and ending index numbers of the subsequence separated by a colon, and a new sequence is returned:

```
name = "Ignatius"
name[2:5]
'nat'
```

The subsequence that is returned contains items starting from the first index and up to, but not including, the ending index. If you leave out the beginning index, the subsequence starts at the beginning of the parent sequence; if you leave out the end index, the subsequence goes to the end of the sequence:

```
name[:5]
'Ignat'

name[4:]
'tius'
```

You can use negative index numbers to create slices counting from the end of a sequence. This example shows how to grab the last three letters of a string:

```
name[-3:]
'ius'
```

If you want a slice to skip items, you can provide a third argument that indicates what to count by. So, if you have a list sequence of integers, as shown earlier, you can create a slice just by using the starting and ending index numbers:

```
scores = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18]
scores[3:15]
[3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

But you can also indicate the step to take, such as counting by threes:

```
scores[3:15:3]
[3, 6, 9, 12]
```

To count backward, you use a negative step:

```
scores[18:0:-4]
[18, 14, 10, 6, 2]
```

Interrogation

You can perform shared operations on sequences to glean information about them. Because a sequence is finite, it has a length, which you can find by using the `len` function:

```
name = "Ignatius"
len(name)
8
```

You can use the `min` and `max` functions to find the minimum and maximum items, respectively:

```
scores = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18]
min(scores)
0

max(name)
'u'
```

These methods assume that the contents of a sequence can be compared in a way that implies an ordering. For sequence types that allow for mixed item types, an error occurs if the contents cannot be compared:

```
max(['Free', 2, 'b'])
----------------------------------------------------------------------
TypeError                              Traceback (most recent call last)
<ipython-input-15-d8babe38f9d9> in <module>()
----> 1 max(['Free', 2, 'b'])
TypeError: '>' not supported between instances of 'int' and 'str'
```

You can find out how many times an item appears in a sequence by using the `count` method:

```
name.count('a')
1
```

You can get the index of an item in a sequence by using the `index` method:

```
name.index('s')
7
```

You can use the result of the `index` method to create a slice up to an item, such as a letter in a string:

```
name[:name.index('u')]
'Ignati'
```

Math Operations

You can perform addition and multiplication with sequences of the same type. When you do, you conduct these operations on the sequence, not on its contents. So, for example, adding the list [1] to the list [2] will produce the list [1,2], not [3]. Here is an example of using the plus (`+`) operator to create a new string from three separate strings:

```
"prefix" + "-" + "postfix"
'prefix-postfix'
```

The multiplication (`*`) operator works by performing multiple additions on the whole sequence, not on its contents:

```
[0,2] * 4
[0, 2, 0, 2, 0, 2, 0, 2]
```

This is a useful way of setting up a sequence with default values. For example, say that you want to track scores for a set number of participants in a list. You can initialize that list so that it has an initial score for each participant by using multiplication:

```
num_participants = 10
scores = [0] * num_participants
scores
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Lists and Tuples

Lists and tuples are sequences that can hold objects of any type. Their contents can be of mixed types, so you can have strings, integers, instances, floats, and anything else in the same list. The items in lists and tuples are separated by commas. The items in a list are enclosed in square brackets, and the items in a tuple are enclosed in parentheses. The main difference between lists and tuples is that lists are mutable, and tuples are immutable. This means that you can change the contents of a list, but once a tuple is created, it cannot be changed. If you want to change the contents of a tuple, you need to make a new one based on the content of the current one. Because of the mutability difference, lists have more functionality than tuples—and they also use more memory.

Creating Lists and Tuples

You create a list by using the list constructor, `list()`, or by just using the square bracket syntax. To create a list with initial values, for example, simply supply the values in brackets:

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

You can create tuples by using the tuple constructor, `tuple()`, or using parentheses. If you want to create a tuple with a single item, you must follow that item with a comma, or Python will interpret the parentheses not as indicating a tuple but as indicating a logical grouping. You can also create a tuple without parentheses by just putting a comma after an item. Listing 3.1 provides examples of tuple creation.

Listing 3.1 **Creating Tuples**

```
tup = (1,2)
tup
(1,2)

tup = (1,)
tup
(1,)

tup = 1,2,
tup
(1,2)
```

Warning

A common but subtle bug occurs when you leave a trailing comma behind an argument to a function. It turns the argument into a tuple containing the original argument. So the second argument to the function `my_function(1, 2,)` will be `(2,)` and not `2`.

You can also use the list or tuple constructors with a sequence as an argument. The following example uses a string and creates a list of the items the string contains:

```
name = "Ignatius"
letters = list(name)
letters
['I', 'g', 'n', 'a', 't', 'i', 'u', 's']
```

Adding and Removing List Items

You can add items to a list and remove items from a list. To conceptualize how it works, think of a list as a stack of books. The most efficient way to add items to a list is to use the `append` method, which adds an item to the end of the list, much as you could easily add a book to the top of a stack. To add an item to a different position in the list, you can use the `insert` method, with the index number where you wish to position the new item as an argument. This is less efficient than using the `append` method as the other items in the list may need to move to make room for the new item; however, this is typically an issue only in very large lists. Listing 3.2 shows examples of appending and inserting.

Listing 3.2 **Appending and Inserting List Items**

```
flavours = ['Chocolate', 'Vanilla']
flavours
['Chocolate', 'Vanilla']

flavours.append('SuperFudgeNutPretzelTwist')
flavours
['Chocolate', 'Vanilla', 'SuperFudgeNutPretzelTwist']

flavours.insert(0,"sourMash")
flavours
['sourMash', 'Chocolate', 'Vanilla', 'SuperFudgeNutPretzelTwist']
```

To remove an item from a list, you use the `pop` method. With no argument, this method removes the last item. By using an optional index argument, you can specify a specific item. In either case, the item is removed from the list and returned.

The following example pops the last item off the list and then pops off the item at index 0. You can see that both items are returned when they are popped and that they are then gone from the list:

```
flavours.pop()
'SuperFudgeNutPretzelTwist'
```

```
flavours.pop(0)
'sourMash'

flavours
 ['Chocolate', 'Vanilla']
```

To add the contents of one list to another, you use the `extend` method:

```
deserts = ['Cookies', 'Water Melon']
desserts
['Cookies', 'Water Melon']

desserts.extend(flavours)
desserts
['Cookies', 'Water Melon', 'Chocolate', 'Vanilla']
```

This method modifies the first list so that it now has the contents of the second list appended to its contents.

Nested List Initialization

There is a tricky bug that bites beginning Python developers. It involves combining list mutability with the nature of multiplying sequences. If you want to initialize a list containing four sublists, you might try multiplying a single list in a list like this:

```
lists = [[]] * 4
lists
[[], [], [], []]
```

This appears to have worked, until you modify one of the sublists:

```
lists[-1].append(4)
lists
[[4], [4], [4], [4]]
```

All of the sublists are modified! This is because the multiplication only initializes one list and references it four times. The references look independent until you try modifying one. The solution to this is to use a list comprehension (discussed further in Chapter 13, "Functional Programming"):

```
lists = [[] for _ in range(4)]
lists[-1].append(4)
lists
 [[], [], [], [4]]
```

Unpacking

You can assign values to multiple variables from a list or tuple in one line:

```
a, b, c = (1,3,4)
a
1
```

```
b
3

c
4
```

Or, if you want to assign multiple values to one variable while assigning single ones to the others, you can use a `*` next to the variable that will take multiple values. Then that variable will absorb all the items not assigned to other variables:

```
*first, middle, last = ['horse', 'carrot', 'swan', 'burrito', 'fly']
first
['horse', 'carrot', 'swan']

last
'fly'

middle
'burrito'
```

Sorting Lists

For lists you can use built-in `sort` and `reverse` methods that can change the order of the contents. Much like the sequence `min` and `max` functions, these methods work only if the contents are comparable, as shown in these examples:

```
name = "Ignatius"
letters = list(name)
letters
['I', 'g', 'n', 'a', 't', 'i', 'u', 's']

letters.sort()
letters
['I', 'a', 'g', 'i', 'n', 's', 't', 'u']

letters.reverse()
letters
['u', 't', 's', 'n', 'i', 'g', 'a', 'I']
```

Strings

A string is a sequence of characters. In Python, strings are Unicode by default, and any Unicode character can be part of a string. Strings are represented as characters surrounded by quotation marks. Single or double quotations both work, and strings made with them are equal:

```
'Here is a string'
'Here is a string'

"Here is a string" == 'Here is a string'
True
```

If you want to include quotation marks around a word or words within a string, you need to use one type of quotation marks—single or double—to enclose that word or words and use the other type of quotation marks to enclose the whole string. The following example shows the word *is* enclosed in double quotation marks and the whole string enclosed in single quotation marks:

```
'Here "is" a string'
'Here "is" a string'
```

You enclose multiple-line strings in three sets of double quotation marks as shown in the following example:

```
a_very_large_phrase = """
Wikipedia is hosted by the Wikimedia Foundation,
a non-profit organization that also hosts a range of other projects.
"""
```

With Python strings you can use special characters, each preceded by a backslash. The special characters include `\t` for tab, `\r` for carriage return, and `\n` for newline. These characters are interpreted with special meaning during printing. While these characters are generally useful, they can be inconvenient if you are representing a Windows path:

```
windows_path = "c:\row\the\boat\now"
print(windows_path)

ow heoat
      ow
```

For such situations, you can use Python's raw string type, which interprets all characters literally. You signify the raw string type by prefixing the string with an `r`:

```
windows_path = r"c:\row\the\boat\now"
print(windows_path)
c:\row\the\boat\now
```

As demonstrated in Listing 3.3, there are a number of string helper functions that enable you to deal with different capitalizations.

Listing 3.3 **String Helper Functions**

```
captain = "Patrick Tayluer"
captain
'Patrick Tayluer'

captain.capitalize()
'Patrick tayluer'

captain.lower()
'patrick tayluer'

captain.upper()
'PATRICK TAYLUER'
```

```
captain.swapcase()
'pATRICK tAYLUER'

captain = 'patrick tayluer'
captain.title()
'Patrick Tayluer'
```

Python 3.6 introduced format strings, or f-strings. You can insert values into f-strings at runtime by using replacement fields, which are delimited by curly braces. You can insert any expression, including variables, into the replacement field. An f-string is prefixed with either an `F` or an `f`, as shown in this example:

```
strings_count = 5
frets_count = 24
f"Noam Pikelny's banjo has {strings_count} strings and {frets_count} frets"
'Noam Pikelny's banjo has 5 strings and 24 frets'
```

This example shows how to insert a mathematic expression into the replacement field:

```
a = 12
b = 32
f"{a} times {b} equals {a*b}"
'12 times 32 equals 384'
```

This example shows how to insert items from a list into the replacement field:

```
players = ["Tony Trischka", "Bill Evans", "Alan Munde"]
f"Performances will be held by {players[1]}, {players[0]}, and {players[2]}"
'Performances will be held by Bill Evans, Tony Trischka, and Alan Munde'
```

Ranges

Using range objects is an efficient way to represent a series of numbers, ordered by value. They are largely used for specifying the number of times a loop should run. Chapter 5 introduces loops. Range objects can take start (optional), end, and step (optional) arguments. Much as with slicing, the start is included in the range, and the end is not. Also as with slicing, you can use negative steps to count down. Ranges calculate numbers as you request them, and so they don't need to store more memory for large ranges. Listing 3.4 demonstrates how to create ranges with and without the optional arguments. This listing makes lists from the ranges so that you can see the full contents that the range would supply.

Listing 3.4 **Creating Ranges**

```
range(10)
range(0, 10)

list(range(1, 10))
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
list(range(0,10,2))
[0, 2, 4, 6, 8]

list(range(10, 0, -2))
[10, 8, 6, 4, 2]
```

Summary

This chapter covers the import group of types known as sequences. A sequence is an ordered, finite collection of items. Lists and tuples can contain mixed types. Lists can be modified after creation, but tuples cannot. Strings are sequences of text. Range objects are used to describe ranges of numbers. Lists, strings, and ranges are among the most commonly used types in Python.

Questions

1. How would you test whether `a` is in the list `my_list`?
2. How would you find out how many times `b` appears in a string named `my_string`?
3. How would you add `a` to the end of the list `my_list`?
4. Are the strings `'superior'` and `"superior"` equal?
5. How would you make a range going from 3 to 13?

Index

A

anonymous functions, 80

Anscombe, F., 135

apply() method, 132–133

arguments, 30

arrays

- broadcasting, 98–99
  - expanding dimensions, 99–100
- changing values in, 91
- copies, changing values in, 95
- creating, 86–88
  - one-dimensional, 87
  - two-dimensional, 88
  - using reshape method, 88–89
- element-by-element operations, 91–92
- filtering values, 92–94
- indexing, 89–90
- matrix operations, 96–97
- methods, 95–96
- one-dimensional, 87
- sequences and, 91
- setting type automatically, 97
- setting type explicitly, 97–98
- slicing, 89–90
- two-dimensional, 88
  - indexing and slicing, 90
- views, 94
  - changing values in, 94

assert statements, 16–17

assignment statements, 17

attributes, 22
axes, 136, 143–144

B

binomial distribution, 105–107
Bokeh, 149–150
Boolean operators, 14, 58–59, 125
  DataFrames and, 126–127
bracket syntax, 121–122
break statement, 64
break statements, 19
broadcasting, 98–99
  expanding dimensions, 99–100
built-in types, 14

C

cells, 4–5
character classes, 209
character sets, 208–209
classes, 22, 188–189
  datetime.date, 207
  inheritance, 196–199
  variables, 190–191
classifier classes, 166
code blocks, 56, 63–64
collocations() method, 165
columns
  creating, 128
  updating, 129
comparison operators, 57–58, 93–94
compiling regular expressions, 211–212
compound statements, 55
  if, 59–62
  structure, 56
concordance() method, 165
constructors
  dict(), 38
  list(), 29
  tuple(), 29
context managers, 205
continue statements, 19
continuous distributions, 108
  exponential distribution, 110
  normal distribution, 108–110
  uniform distribution, 110–111
control statements, 56, 68
copies, changing values in, 95
corpus readers, 160
  loading text, 160–161
  tokenizers, 161
corpuses, downloading, 166–167
creating
  arrays, 86–88
    one-dimensional, 87
    two-dimensional, 88
    using reshape method, 88–89
  columns, 128
  DataFrames, 114
    from a dictionary, 114–115
    from a file, 116
    from a list of lists, 115–116
  datetime object, 206
  dictionaries, 38
  lists, 29–30
  tuples, 29–30

D

DataFrames, 113
  accessing data, 120–121
  apply() method, 132–133
  Boolean operators, 126–127
  bracket syntax, 121–122
  columns
    creating, 128
    updating, 129

creating, 114
from a dictionary, 114–115
from a file, 116
from a list of lists, 115–116
data manipulation, 129–131
describe method, 118
exclude argument, 120
include keyword, 119–120
percentiles argument, 118–119
head method, 117
interacting with, 117
interactive display, 133
manipulating, 127–128, 129
masking and filtering, 125–126
methods, 128
optimized access
by index, 124
by label, 123–124
replace method, 131–132
sorting, 204
tail method, 118
datetime object, 207
creating, 206
setting the time zone, 207
translating strings to, 207
decorators, 76–77, 79
syntax, 79–80
del() function, 40
delete statements, 18
describe method, 118
exclude argument, 120
include keyword, 119–120
percentiles argument, 118–119
dict() constructor, 38
dict_key view, 41–42
dictionaries, 37–39
checking for keys, 43
creating, 38
creating DataFrames from, 114–115
get method, 43–44
hash() method, 45
key/value pairs
adding, 39
updating, 39
removing items, 39–40
valid key types, 44–45
dictionary comprehensions, 181
dictionary views, 40–42
dict_key, 41–42
key_item, 42
difference() method, 51
discrete distributions, 105
binomial distribution, 105–107
Poisson distribution, 107–108
disjoint sets, 48
dispersion_plot() method, 165–166
docstrings, 68–69
dot notation, 22
downloading, corpuses, 166–167

E

elif statement, 62
else statement, 61
equality operators, 56–57, 125
estimators, 156
exceptions, 18–19
exponential distribution, 110
expressions, 16
generator, 182–183
extend method, 31

F

figures, 136
fileids() method, 160

files
- creating DataFrames from, 116
- Google Colab, 9–10
- opening, 205
- reading and writing, 204–205

filter() function, 179
- replacing with a list comprehension, 180

filtering, DataFrames, 125–126

find iterator, 211

findall() method, 165

flattening nested lists, 167

for loops, 63–64

FreqDist class
- built-in plot method, 164
- methods, 164

frequency distributions, 161–162
- filtering stopwords, 163–164
- removing punctuation, 162–163

frozensets, 53

f-strings, 34

functional programming, 173, 174–175
- changing mutable data, 176–177
- dictionary comprehensions, 181
- filter() function, 179
- generator(s), 182
  - expressions, 182–183
  - functions, 183–184
- lambda functions, 179
- list comprehensions, 179
  - conditionals and, 181
  - multiple variables, 181
  - replacing map() and filter() with, 180
  - syntax, 179–180
- map() function, 177–178
- operator module, 179
- reduce() function, 178, 179
- scope, 173–174
  - inheriting, 174
  - outer, 175–176
- state, 174

functions, 15, 67
- anonymous, 80
- control statement, 68
- datetime.now(), 206
- decorators, 76–77, 79
  - syntax, 79–80
- del(), 40
- docstring, 68–69
- generator, 183–184
- helper, 33–34
- lambda, 179
- len, 27
- max, 28
- min, 28
- nested, 77
- nested wrapping, 78–79
- open(), 204–205
- as a parameter, 78
- parameters, 69–70
  - default value, 71–72
  - keyword assignments, 70–71
  - keyword wildcard, 74–75
  - mutable defaults, 72–73
  - positional wildcard, 74
  - positional-only, 73
- re.compile(), 211
- re.findall(), 210
- re.finditer(), 211
- re.match(), 207–208
- re.search(), 208
- return statements, 75
- reversed, 41
- scope, 75–76
- sorted(), 202–204
- wrapping, 77–78

future statements, 20

G

generator(s), 182
- expressions, 182–183
- functions, 183–184

get method, 43–44
global statements, 20
Google Colab, 5–6
- code cells, 9
- Code Snippets, 11
- existing collections and, 11
- files, 9–10
- headings, 7–8
- LaTeX, 8–9
- notebooks, managing, 10
- shell commands, 11–12
- system aliases, 11–12
- text cells, 6–8

groups, 209–210
- named, 210

H

hash() method, 45
head method, 117
helper functions, 33–34
high-level programming languages, 15

I

if statements, 59–62
immutable objects, 44–45
import statements, 19–20
index method, 28
indexing, 26
- arrays, 89–90
- DataFrames and, 124

inequality operators, 56–57
inheritance, 196–199
inheriting scope, 174
installing, NumPy, 86
instances, 188
interacting with DataFrame data, 117
interrogation, 27–28
intersections, 51
ints, 14
- numerator attribute, 22

issuperset() method, 50
items() method, 40

J-K

JSON files, opening and reading, 205
Jupyter notebooks, 4–5
Keras, 153
key_item view, 42
keys() method, 40
key/value pairs, 37
- adding, 39
- updating, 39

L

labels, DataFrames access and, 123–124
lambda functions, 80, 179
LaTeX, 8–9
len function, 27
libraries. *See also* NumPy; SciPy
- machine learning, 153–154
- SciPy, 103
- third-party, 85
- visualization, matplotlib, 135–136

list comprehensions, 179
- conditionals and, 181
- multiple variables, 181
- replacing map() and filter() with, 180
- syntax, 179–180

list() constructor, 29

lists, 29
- adding and removing items, 30–31
- creating, 29–30
- creating DataFrames from, 115–116
- flattening, 167
- nested, 31
- sorting, 32, 201–204
- unpacking, 31–32

loops
- break statement, 64
- for, 63–64
- while, 62–63

low-level programming languages, 15

M

machine learning, 153. *See also* Scikit-learn
- overfitting, 155
- splitting test and training data, 155–156
- supervised versus unsupervised learning, 154
- transformations, 154–155

magic functions, 12

manipulating DataFrames, 127–128, 129

map() function, 177–178
- replacing with a list comprehension, 180

Markdown, 6

math operations, 21–22

math operator methods, 195–196

matplotlib, 135–136
- colors, 139
- creating multiple axes, 143–144
- labeled data, 140–141
- line styles, 138
- marker types, 137–138
- object-oriented style, 143
- plotting multiple sets of data, 141–143
- styling plots, 137, 139–140

matrix operations, 96–97

max function, 28

methods, 188–190
- to_bytes(), 187–188
- apply(), 132–133
- arrays and, 95–96
- collocations(), 165
- concordance(), 165
- count, 28
- DataFrames, 128
- describe, 118
  - exclude argument, 120
  - include keyword, 119–120
  - percentiles argument, 118–119
- difference(), 51
- disjoint(), 48
- dispersion_plot(), 165–166
- extend, 31
- fileids(), 160
- findall(), 165
- get, 43–44
- hash(), 45
- head, 117
- index, 28
- inheritance, 196–199
- intersection(), 51
- issuperset(), 50
- items(), 40
- keys(), 40
- math operator, 195–196
- min(), 125
- pop, 30
- private, 190
- public, 190
- replace, 131–132
- representation, 192
- reverse, 32
- rich comparison, 192–195

similar(), 165
sort, 32
sort(), 201–202
special, 191
subset(), 49
symmetric difference(), 51
tail, 118
union(), 50
values(), 40

min function, 28
min() method, 125
MinMaxScaler transformer, 154–155
multiple statements, 16
mutable objects, 44, 176–177

N

named groups, 210
substitution and, 211
natural language processing, 159
***Natural Language Processing with Python*, 169**
nested functions, 77
nested lists, 31
nested wrapping functions, 78–79
NLTK (Natural Language Toolkit), 159
classifier classes, 166
defining features, 168
downloading corpuses, 166–167
flattening nested lists, 167
labeling data, 167
training and testing, 168–169
corpus readers, 160
loading text, 160–161
tokenizers, 161
fileids() method, 160
FreqDist class
built-in plot method, 164
methods, 164
frequency distributions, 161–162
filtering stopwords, 163–164
removing punctuation, 162–163
sample texts, 159–160
Text class, 165
collocations() method, 165
concordance() method, 165
dispersion_plot() method, 165–166
findall() method, 165
similar() method, 165
NoneType, 15
nonlocal statements, 20
normal distribution, 108–110
notebooks, 4–5
Google Colab, 5–6
Jupyter, 4–5
managing, 10
numerics, 14
NumPy. *See also* arrays; SciPy
creating arrays, 86–87
installing and importing, 86
polynomials, 100–101

O

object-oriented programming, 187
classes, 188–189
variables, 190–191
inheritance, 196–199
instances, 188
methods, 188–190
math operator, 195–196
representation, 192
rich comparison, 192–195
special, 191
objects, 187–188
private methods, 190

objects, 22, 187–188
- datetime, creating, 206
- evaluation, 59
- immutable, 44–45
- mutable, 44, 176–177
- range, 34–35

one-dimensional arrays, 87
open() function, 204–205
in operator, 26, 40
or operator, 59
operators, 21–22
- Boolean, 58–59
- Boolean operators, 126–127
- comparison, 57–58, 93–94
- equality/inequality, 56–57, 125
- in, 40
- math, 28–29
- or, 59
- walrus, 60

overfitting, 155

P

packages, zoneinfo, 207
Pandas DataFrames. *See* DataFrames
parameters
- default value, 71–72
- functions as, 78
- keyword assignments, 70–71
- keyword wildcard, 74–75
- mutable defaults, 72–73
- positional wildcard, 74
- positional-only, 73

parser, 14
pass statements, 18
Plotly, 148–149
Poisson distribution, 107–108
polynomials, 100–101
pop method, 30
print statements, 20–21
private methods, 190
procedural programming, 174
programming languages, high-level versus low-level, 15
proper subsets, 49
public methods, 190
Punkt tokenizer, 161
Python, types, 14–15
PyTorch, 154

Q-R

quotation marks, strings and, 33
raise statements, 18–19
ranges, 34–35
raw strings, 33
reading files, 204–205
re.compile() function, 211
reduce() function, 178, 179
re.findall() function, 210
re.finditer() function, 211
regular expressions, 207–208
- compiling, 211–212
- groups, 209–210
- named groups, 210
- substitution, 211
  - using named groups, 211

re.match() function, 207–208
removing, items from dictionaries, 39–40
replace method, 131–132
representation methods, 192
re.search() function, 208
return statements, 18, 75
reverse method, 32
rich comparison methods, 192–195
running statements, 4

S

Scikit-learn, 154
- estimators, 156
- MinMaxScaler transformer, 154–155
- splitting test and training data, 155–156
- training a model, 156
- training and testing, 156
- tutorials, 157

SciPy, 103
- continuous distributions, 108
  - exponential distribution, 110
  - normal distribution, 108–110
  - uniform distribution, 110–111
- discrete distributions, 105
  - binomial distribution, 105–107
  - Poisson distribution, 107–108

scipy.misc submodule, 104–105

scipy.special submodule, 105

scipy.stats submodule, 105

scope, 20, 75–76, 173–174
- inheriting, 174

Seaborn, 144–145
- plot types, 148
- themes, 145–147

sequences, 14, 25
- arrays and, 91
- frozensets and, 53
- indexing, 26
- interrogation, 27–28
- intersections, 51
- lists, 29
  - adding and removing items, 30–31
  - nested, 31
  - sorting, 32
  - unpacking, 31–32
- math operations, 28–29
- slicing, 27
- testing membership, 26
- tuples, 29
  - unpacking, 31–32

sets, 46–48
- difference between, 51
- disjoint, 48
- proper subsets, 49
- subsets and, 49
- supersets and, 50
- symmetric difference, 51
- union, 50
- updating, 51–52

shared operations, 25

similar() method, 165

slicing, 27
- arrays, 89–90
- DataFrames, 122

sort() method, 201–202

sort method, 32

sorted() function, 202–204

sorting, lists, 32, 201–204

special characters, 33

statements, 15–16
- assert, 16–17
- assignment, 17
- break, 19, 64
- code blocks, 56, 63–64
- continue, 19, 64–65
- delete, 18
- elif, 62
- else, 61
- expression, 16
- future, 20
- global, 20
- if, 59–62
- import, 19–20
- multiple, 16
- nonlocal, 20

pass, 18
print, 20–21
raise, 18–19
return, 18, 75
running, 4
yield, 18
stopwords, 163–164
strings, 14, 32–33
f-, 34
helper functions, 33–34
quotation marks and, 33
raw, 33
special characters, 33
translating to datetime object, 207
submodules
scipy.misc, 104–105
scipy.special, 105
scipy.stats, 105
subset() method, 49
substitution, 211
supersets, 50
symmetric difference() method, 51
syntax
bracket, 121–122
decorators, 79–80
list comprehensions, 179–180

T

tail method, 118
TensorFlow, 153
text cells, 6–8
Text class, 165
collocations() method, 165
concordance() method, 165
dispersion_plot() method, 165–166
findall() method, 165
similar() method, 165
third-party libraries, 85
time series data, 206
time zone, setting for datetime object, 207
to_bytes() method, 187–188
tokenizers, 161
transformations, 154–155
tuple() constructor, 29
tuples, 29
creating, 29–30
unpacking, 31–32
two-dimensional arrays, 88
indexing and slicing, 90
types, 14–15. *See also* sequences

U

uniform distribution, 110–111
union() method, 50
updating
columns, -129
sets, 51–52

V

values() method, 40
variables, 190–191
views, 94
changing values in, 94
visualization libraries, 151
Bokeh, 149–150
matplotlib, 135–136
colors, 139
creating multiple axes, 143–144
labeled data, 140–141
line styles, 138
marker types, 137–138
object-oriented style, 143
plotting multiple sets of data, 141–143
styling plots, 137, 139–140
Plotly, 148–149

  Seaborn, 144–145
    plot types, 148
    themes, 145–147

W

walrus operator, 60

while loops, 62–63

wrapping functions, 77–78

writing file data, 204–205

X-Y-Z

yield statements, 18

zoneinfo package, 207