PYTHON 3 AND DATA ANALYTICS

POCKET PRIMER

O. CAMPESATO

Python 3 and Data Analytics

Pocket Primer

LICENSE, DISCLAIMER OF LIABILITY, AND LIMITED WARRANTY

By purchasing or using this book and disc (the "Work"), you agree that this license grants permission to use the contents contained herein, including the disc, but does not give you the right of ownership to any of the textual content in the book / disc or ownership to any of the information or products contained in it. *This license does not permit uploading of the Work onto the Internet or on a network (of any kind) without the written consent of the Publisher.* Duplication or dissemination of any text, code, simulations, images, etc. contained herein is limited to and subject to licensing terms for the respective products, and permission must be obtained from the Publisher or the owner of the content, etc., in order to reproduce or network any portion of the textual material (in any media) that is contained in the Work.

Mercury Learning and Information ("MLI" or "the Publisher") and anyone involved in the creation, writing, or production of the companion disc, accompanying algorithms, code, or computer programs ("the software"), and any accompanying Web site or software of the Work, cannot and do not warrant the performance or results that might be obtained by using the contents of the Work. The author, developers, and the Publisher have used their best efforts to ensure the accuracy and functionality of the textual material and/or programs contained in this package; we, however, make no warranty of any kind, express or implied, regarding the performance of these contents or programs. The Work is sold "as is" without warranty (except for defective materials used in manufacturing the book or due to faulty workmanship).

The author, developers, and the publisher of any accompanying content, and anyone involved in the composition, production, and manufacturing of this work will not be liable for damages of any kind arising out of the use of (or the inability to use) the algorithms, source code, computer programs, or textual material contained in this publication. This includes, but is not limited to, loss of revenue or profit, or other incidental, physical, or consequential damages arising out of the use of this Work.

The sole remedy in the event of a claim of any kind is expressly limited to replacement of the book and/or disc, and only at the discretion of the Publisher. The use of "implied warranty" and certain "exclusions" vary from state to state, and might not apply to the purchaser of this product.

Companion files for this title are available by writing to the publisher at info@merclearning.com.

PYTHON 3 AND DATA ANALYTICS

Pocket Primer

Oswald Campesato

MERCURY LEARNING AND INFORMATION

Dulles, Virginia
Boston, Massachusetts
New Delhi

Publisher: David Pallai
Mercury Learning and Information
22841 Quicksilver Drive
Dulles, VA 20166
info@merclearning.com
www.merclearning.com
800-232-0223

O. Campesato. *Python 3 and Data Analytics Pocket Primer.*
ISBN: 978-1-68392-654-2

Library of Congress Control Number: 2021934305

212223321 This book is printed on acid-free paper in the United States of America.

I'd like to dedicate this book to my parents –
may this bring joy and happiness into their lives.

CONTENTS

PREFACE

WHAT IS THE PRIMARY VALUE PROPOSITION FOR THIS BOOK?

This book contains a fast-paced introduction to as much relevant information about data analytics as possible in a book of this size. At the same time, please keep in mind: *you will not become an expert in data analytics by reading this book.*

However, you will be exposed to a variety of features of `NumPy` and `Pandas`, how to write regular expressions (with the accompanying appendix), and how to perform many data cleaning tasks. Keep in mind that some topics are presented in a cursory manner for two main reasons. First, it's important that you be exposed to these concepts. In some cases, you will find topics that might pique your interest, and hence motivate you to learn more about them through self-study; in other cases, you will probably be satisfied with a brief introduction. In other words, you will decide whether or not to delve into more detail regarding the topics in this book. Second, a full treatment of all the topics that are covered in this book would significantly increase its length. This is contrary to the series design as "primers." *It's important for you to decide if this approach is suitable for your needs and learning style: if not, you can select one or more of the plethora of data analytics books that are available.*

THE TARGET AUDIENCE

The book is intended primarily for people who have worked with Python and are interested in learning about several important Python libraries, such as `NumPy` and `Pandas`.

It is also intended to reach an international audience of readers with highly diverse backgrounds in various age groups. While many readers know how to read English, their native spoken language is not English (which could be their

second, third, or even fourth language). Consequently, this book uses standard English rather than colloquial expressions that might be confusing to those readers. As you know, many people learn by different types of imitation, which includes reading, writing, or hearing new material. The book takes these points into consideration in order to provide a comfortable and meaningful learning experience for the intended readers.

WHAT WILL I LEARN FROM THIS BOOK?

The first chapter contains a quick tour of basic Python 3, followed by a chapter which introduces you to data types and data cleaning tasks, such as working with datasets that contain different types of data, and how to handle missing data. The third and fourth chapters introduce you to `NumPy` and `Pandas` (and many code samples).

The fifth chapter contains fundamental concepts in probability and statistics, such as mean, mode, and variance and correlation matrices. You will also learn about Gini impurity, entropy, and KL-divergence. The book covers eigenvalues, eigenvectors, and PCA (principal component analysis).

The sixth and final chapter of this book delves into data visualization with Matplotlib, Seaborn, and an example of a rendering of graphics effects in Bokeh. Finally, there is an appendix for regular expressions, with enough examples so you can understand most regular expressions that you will encounter in your code.

WHY ARE THE CODE SAMPLES PRIMARILY IN PYTHON?

Most of the code samples are short (usually less than one page and sometimes less than half a page), and if need be, you can easily and quickly copy/paste the code into a new Jupyter notebook. For the Python code samples that reference a CSV file, you do not need any additional code in the corresponding Jupyter notebook to access the CSV file. Moreover, the code samples execute quickly, so you won't need to avail yourself of the free GPU that is provided in Google Colaboratory.

If you do decide to use Google Colaboratory, you can easily copy/paste the Python code into a notebook, and also use the upload feature to upload existing Jupyter notebooks. Keep in mind the following point: if the Python code references a CSV file, make sure that you include the appropriate code snippet (as explained in Chapter 1) to access the CSV file in the corresponding Jupyter notebook in Google Colaboratory.

DO I NEED TO LEARN THE THEORY PORTIONS OF THIS BOOK?

Once again, the answer depends on the extent to which you plan to become involved in data analytics. For example, if you plan to study machine learning, then you will probably learn how to create and train a model, which is a

task that is performed after data cleaning tasks. In general, you will probably need to learn everything that you encounter in this book if you are planning to become a machine learning engineer.

WHY DOES THIS BOOK INCLUDE SKLEARN MATERIAL?

First, keep in mind that the Sklearn material in this book is minimalistic because this book is not about machine learning. Second, the Sklearn material is located in Chapter 6 where you will learn about some of the Sklearn built-in datasets. If you decide to delve into machine learning, you will have already been introduced to some aspects of Sklearn.

WHY IS A REGEX APPENDIX INCLUDED IN THIS BOOK?

Regular expressions are supported in multiple languages (including Java and JavaScript) and they enable you to perform complex tasks with very compact, regular expressions. Alas, regular expressions can seem arcane and too complex to learn in a reasonable amount of time. Fortunately, there is good news: Chapter 2 contains some Pandas-based code samples that use regular expressions to perform tasks that might otherwise be more complicated.

If you plan to use Pandas extensively or you plan to work on NLP-related tasks, then the code samples in the appendix will be very useful for you because they are more than adequate for solving certain types of tasks, such as removing HTML tags. Moreover, the knowledge you gain will transfer instantly to other languages that support regular expressions.

GETTING THE MOST FROM THIS BOOK

Some programmers learn well from prose, others learn well from sample code (and lots of it), which means that there's no single style that can be used for everyone.

Moreover, some programmers want to run the code first, see what it does, and then return to the code to delve into the details (and others use the opposite approach).

Consequently, there are various types of code samples in this book: some are short, some are long, and other code samples "build" from earlier code samples.

WHAT DO I NEED TO KNOW FOR THIS BOOK?

Current knowledge of Python 3.x is the most helpful skill. Knowledge of other programming languages (such as Java) can also be helpful because of the exposure to programming concepts and constructs. The less technical knowledge that you have, the more diligence will be required in order to understand the various topics that are covered.

If you want to be sure that you can grasp the material in this book, glance through some of the code samples to get an idea of how much is familiar to you and how much is new for you.

DON'T THE COMPANION FILES OBVIATE THE NEED FOR THIS BOOK?

The companion files contain all the code samples to save you time and effort from the error-prone process of manually typing code into a text file. There are situations, however, in which you might not have easy access to the companion files. Furthermore, the code samples in the book provide explanations that are not available on the companion files.

DOES THIS BOOK CONTAIN PRODUCTION-LEVEL CODE SAMPLES?

The primary purpose of the code samples in this book is to show you Python-based libraries for solving a variety of data-related tasks in conjunction with acquiring a rudimentary understanding of statistical concepts. Clarity has higher priority than writing more compact code that is more difficult to understand (and possibly more prone to bugs). If you decide to use any of the code in this book in a production website, you ought to subject that code to the same rigorous analysis as the other parts of your code base.

WHAT ARE THE NON-TECHNICAL PREREQUISITES FOR THIS BOOK?

Although the answer to this question is more difficult to quantify, it's very important to have strong desire to learn about data analytics, along with the motivation and discipline to read and understand the code samples.

HOW DO I SET UP A COMMAND SHELL?

If you are a Mac user, there are three methods. The first is to use `Finder` to navigate to `Applications > Utilities` and then double click on the `Utilities` application. Next, if you already have a command shell available, you can launch a new command shell by typing the following command:

```
open /Applications/Utilities/Terminal.app
```

A second method for Mac users is to open a new command shell on a MacBook from a command shell that is already visible, simply by clicking `command+n` in that command shell, and your Mac will launch another command shell.

If you are a PC user, you can install Cygwin (open source *https://cygwin.com/*) that simulates bash commands, or use another toolkit such as MKS (a commercial product). Please read the online documentation that describes the

download and installation process. Note that custom aliases are not automatically set if they are defined in a file other than the main start-up file (such as .bash_login).

COMPANION FILES

All the code samples and figures in this book may be obtained by writing to the publisher at info@merclearning.com.

WHAT ARE THE "NEXT STEPS" AFTER FINISHING THIS BOOK?

The answer to this question varies widely, mainly because the answer depends heavily on your objectives. If you are interested primarily in NLP, then you can learn more advanced concepts, such as attention, transformers, and the BERT-related models.

If you are primarily interested in machine learning, there are some subfields of machine learning, such as deep learning and reinforcement learning (and deep reinforcement learning) which might appeal to you. Fortunately, there are many resources available, and you can perform an Internet search for those resources. One other point: the aspects of machine learning for you to learn depend on your interests: the needs of a machine learning engineer, data scientist, manager, student or software developer are all different.

Oswald Campesato
March 2021

CHAPTER 1

INTRODUCTION TO PYTHON

This chapter contains an introduction to Python with information about useful tools for installing Python modules, basic Python constructs, and how to work with some data types in Python.

The first part of this chapter covers how to install Python, some Python environment variables, and how to use the Python interpreter. You will see Python code samples and also how to save Python code in text files that you can launch from the command line. The second part of this chapter shows you how to work with simple data types such as numbers, fractions, and strings. The final part of this chapter discusses exceptions and how to use them in Python scripts.

NOTE *The Python scripts in this book are for Python 3 and more details are provided in a subsequent section.*

TOOLS FOR PYTHON

The Anaconda distribution is available for Windows, Linux, and Mac, and it's downloadable here:

http://continuum.io/downloads.

Anaconda is well-suited for modules such as `NumPy` (discussed in Chapter 3) this and `scipy` (not discussed in this book), and if you are a Windows user, Anaconda appears to be a better alternative.

easy_install and pip

Both `easy_install` and `pip` are very easy to use when you need to install Python modules.

Whenever you need to install a Python module (and there are many in this book), use either `easy_install` or `pip` with the following syntax:

```
easy_install <module-name>
pip install <module-name>
```

NOTE *Python-based modules are easier to install, whereas modules with code written in C are usually faster but more difficult in terms of installation.*

virtualenv

The `virtualenv` tool enables you to create isolated Python environments, and its home page is here:

http://www.virtualenv.org/en/latest/virtualenv.html.

`virtualenv` addresses the problem of preserving the correct dependencies and versions (and indirectly, permissions) for different applications. If you are a Python novice, you might not need `virtualenv` right now, but keep this tool in mind.

IPython

Another very good tool is `IPython` (which won a Jolt award), and its home page is here:

http://ipython.org/install.html.

Type `ipython` to invoke `IPython` from the command line:

```
ipython
```

The preceding command displays the following output:

```
Python 3.9.1 (v3.9.1:1e5d33e9b9, Dec  7 2020, 12:44:01)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.20.0 -- An enhanced Interactive Python. Type '?'
for help.
In [1]:
```

Now type a question mark ("?") at the prompt and you will see some useful information, a portion of which is here:

```
IPython -- An enhanced Interactive Python
=========================================

IPython offers a fully compatible replacement for the
standard Python interpreter, with convenient shell
features, special commands, command history mechanism and
output results caching.

At your system command line, type 'ipython -h' to see
the command line options available. This document only
describes interactive features.
```

```
GETTING HELP
------------

Within IPython you have various way to access help:

  ?         -> Introduction and overview of IPython's
                                        features (this screen).
  object?   -> Details about 'object'.
  object??  -> More detailed, verbose information about 'object'.
  %quickref -> Quick reference of all IPython specific
                                            syntax and magics.

  help      -> Access Python's own help system.

If you are in terminal IPython you can quit this screen by
pressing 'q'.
```

Finally, simply type `quit` at the command prompt and you will exit the `IPython` shell.

The next section shows you how to check whether or not Python is installed on your machine, and also where you can download Python.

PYTHON INSTALLATION

Before you download anything, check if you have Python already installed on your machine (which is likely if you have a MacBook or a Linux machine) by typing the following command in a command shell:

```
python -V
```

The output for the MacBook used in this book is here:

```
Python 3.9.1
```

NOTE 1 *Install Python 3.9.1 (or as close as possible to this version) on your machine if you want to use the same version of Python that was used to test the Python scripts in this book. However, the Python scripts in this book will probably also work with versions of Python that are in the range 3.7.x to 3.9.1.*

NOTE 2 *When you install Python 3.x, the executable might be python3 instead of python, and be sure to use python3 if it is version 3.x of Python.*

If you need to install Python on your machine, navigate to the Python home page and select the downloads link or navigate directly to this website:
http://www.python.org/download/.

In addition, PythonWin is available for Windows, and its home page is here:
http://www.cgl.ucsf.edu/Outreach/pc204/pythonwin.html.

Use any text editor that can create, edit, and save Python scripts and save them as plain text files (don't use Microsoft Word).

After you have Python installed and configured on your machine, you are ready to work with the Python scripts in this book.

SETTING THE PATH ENVIRONMENT VARIABLE (WINDOWS ONLY)

The PATH environment variable specifies a list of directories that are searched whenever you specify an executable program from the command line. A very good guide to setting up your environment so that the Python executable is always available in every command shell is to follow the instructions here:

http://www.blog.pythonlibrary.org/2011/11/24/python-101-setting-up-python-on-windows/.

LAUNCHING PYTHON ON YOUR MACHINE

There are three different ways to launch Python:

- Use the Python Interactive Interpreter
- Launch Python scripts from the command line
- Use an IDE

The next section shows you how to launch the Python interpreter from the command line, and later in this chapter you will learn how to launch Python scripts from the command line and also about Python IDEs.

NOTE *The emphasis in this book is to launch Python scripts from the command line or to enter code in the Python interpreter.*

The Python Interactive Interpreter

Launch the Python interactive interpreter from the command line by opening a command shell and typing the following command:

```
python
```

You will see the following prompt (or something similar):

```
Python 3.9.1 (v3.9.1:1e5d33e9b9, Dec  7 2020, 12:44:01)
[Clang 12.0.0 (clang-1200.0.32.2)] on darwin
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

Now type the expression `2 + 7` at the prompt:

```
>>> 2 + 7
```

Python displays the following result:

```
9
>>>
```

Press `ctrl-d` to exit the Python shell.

You can launch any Python script from the command line by preceding it with the word "python". For example, if you have a Python script `myscript.py` that contains Python commands, launch the script as follows:

```
python myscript.py
```

As a simple illustration, suppose that the Python script `myscript.py` contains the following Python code:

```
print('Hello World from Python')
print('2 + 7 = ', 2+7)
```

When you launch the preceding Python script, you will see the following output:

```
Hello World from Python
2 + 7 =  9
```

PYTHON IDENTIFIERS

A Python identifier is the name of a variable, function, class, module, or other Python object, and a valid identifier conforms to the following rules:

- starts with a letter A to Z or a to z or an underscore (_)
- zero or more letters, underscores, and digits (0 to 9)

NOTE *Python identifiers cannot contain characters such as @, $, and %.*

Python is a case-sensitive language, so `Abc` and `abc` are different identifiers in Python.

In addition, Python has the following naming convention:

- Class names start with an uppercase letter and all other identifiers with a lowercase letter
- an initial underscore is used for private identifiers
- two initial underscores is used for strongly private identifiers

A Python identifier with two initial underscore and two trailing underscore characters indicates a language-defined special name.

LINES, INDENTATION, AND MULTILINES

Unlike other programming languages (such as Java or Objective-C), Python uses indentation instead of curly braces for code blocks. Indentation must be consistent in a code block, as shown here:

```
if True:
    print("ABC")
    print("DEF")
else:
    print("ABC")
    print("DEF")
```

Multiline statements in Python can terminate with a new line or the backslash ("\") character, as shown here:

```
total = x1 + \
        x2 + \
        x3
```

Obviously, you can place `x1`, `x2`, and `x3` on the same line, so there is no reason to use three separate lines; however, this functionality is available in case you need to add a set of variables that does not fit on a single line.

You can specify multiple statements in one line by using a semicolon (";") to separate each statement, as shown here:

```
a=10; b=5; print(a); print(a+b)
```

The output of the preceding code snippet is here:

```
10
15
```

NOTE *The use of semicolons and the continuation character are discouraged in Python.*

QUOTATION AND COMMENTS IN PYTHON

Python allows single (‘), double (“), and triple (‘’’ or “””) quotes for string literals, provided that they match at the beginning and the end of the string. You can use triple quotes for strings that span multiple lines. The following examples are legal Python strings:

```
word = 'word'
line = "This is a sentence."
para = """This is a paragraph. This paragraph contains
more than one sentence."""
```

A string literal that begins with the letter "r" (for "raw") treats everything as a literal character and "escapes" the meaning of meta characters, as shown here:

```
a1 = r'\n'
a2 = r'\r'
a3 = r'\t'
print('a1:',a1,'a2:',a2,'a3:',a3)
```

The output of the preceding code block is here:

```
a1: \n a2: \r a3: \t
```

You can embed a single quote in a pair of double quotes (and vice versa) in order to display a single quote or a double quote. Another way to accomplish the same result is to precede a single or double quote with a backslash ("\") character. The following code block illustrates these techniques:

```
b1 = "'"
b2 = '"'
b3 = '\''
b4 = "\""
print('b1:',b1,'b2:',b2)
print('b3:',b3,'b4:',b4)
```

The output of the preceding code block is here:

```
b1: ' b2: "
b3: ' b4: "
```

A hash sign (#) that is not inside a string literal is the character that indicates the beginning of a comment. Moreover, all characters after the # and up to the physical line end are part of the comment (and ignored by the Python interpreter). Consider the following code block:

```
#!/usr/bin/python
# First comment
print("Hello, Python!")  # second comment
```

This will produce following result:

```
Hello, Python!
```

A comment may be on the same line after a statement or expression:

```
name = "Tom Jones" # This is also comment
```

You can comment multiple lines as follows:

```
# This is comment one
# This is comment two
# This is comment three
```

A blank line in Python is a line containing only white space, a comment, or both.

SAVING YOUR CODE IN A MODULE

Earlier you saw how to launch the Python interpreter from the command line and then enter Python commands. However, everything that you type in the Python interpreter is only valid for the current session: if you exit the interpreter and then launch the interpreter again, your previous definitions are no longer valid. Fortunately, Python enables you to store code in a text file, as discussed in the next section.

A *module* in Python is a text file that contains Python statements. In the previous section, you saw how the Python interpreter enables you to test code snippets whose definitions are valid for the current session. If you want to retain the code snippets and other definitions, place them in a text file so that you can execute that code outside of the Python interpreter.

The outermost statements in Python are executed from top to bottom when the module is imported for the first time, which will then set up its variables and functions.

A Python module can be run directly from the command line, as shown here:

```
python first.py
```

As an illustration, place the following two statements in a text file called `first.py`:

```
x = 3
print(x)
```

Now type the following command:

```
python first.py
```

The output from the preceding command is `3`, which is the same as executing the preceding code from the Python interpreter.

When a Python module is run directly, the special variable `__name__` is set to `__main__`. You will often see the following type of code in a Python module:

```
if __name__ == '__main__':
    # do something here
    print('Running directly')
```

The preceding code snippet enables Python to determine if a Python module was launched from the command line or imported into another Python module.

SOME STANDARD MODULES IN PYTHON

The Python Standard Library provides many modules that can simplify your own Python scripts. A list of the Standard Library modules is here:

http://www.python.org/doc/.

Some of the most important Python modules include `cgi`, `math`, `os`, `pickle`, `random`, `re`, `socket`, `sys`, `time`, and `urllib`.

The code samples in this book use the modules `math`, `os`, `random`, `re`, `socket`, `sys`, `time`, and `urllib`. You need to import these modules in order to use them in your code. For example, the following code block shows you how to import four standard Python modules:

```
import datetime
import re
import sys
import time
```

The code samples in this book import one or more of the preceding modules, as well as other Python modules.

THE `HELP()` AND `DIR()` FUNCTIONS

An Internet search for Python-related topics usually returns a number of links with useful information. Alternatively, you can check the official Python documentation site: docs.python.org.

In addition, Python provides the `help()` and `dir()` functions that are accessible from the Python interpreter. The `help()` function displays documentation strings, whereas the `dir()` function displays defined symbols.

For example, if you type `help(sys)`, you will see documentation for the `sys` module, whereas `dir(sys)` displays a list of the defined symbols.

Type the following command in the Python interpreter to display the string-related methods in Python:

```
>>> dir(str)
```

The preceding command generates the following output:

```
['__add__', '__class__', '__contains__', '__delattr__',
'__doc__', '__eq__', '__format__', '__ge__', '__
getattribute__', '__getitem__', '__getnewargs__', '__
getslice__', '__gt__', '__hash__', '__init__', '__le__',
'__len__', '__lt__', '__mod__', '__mul__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__rmod__', '__rmul__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__', '_formatter_field_name_
split', '_formatter_parser', 'capitalize', 'center',
'count', 'decode', 'encode', 'endswith', 'expandtabs',
'find', 'format', 'index', 'isalnum', 'isalpha', 'isdigit',
'islower', 'isspace', 'istitle', 'isupper', 'join',
'ljust', 'lower', 'lstrip', 'partition', 'replace',
'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit',
'rstrip', 'split', 'splitlines', 'startswith', 'strip',
'swapcase', 'title', 'translate', 'upper', 'zfill']
```

The preceding list gives you a consolidated "dump" of built-in functions (including some that are discussed later in this chapter). Although the `max()` function obviously returns the maximum value of its arguments, the purpose of other functions such as `filter()` or `map()` is not immediately apparent (unless you have used them in other programming languages). In any case, the preceding list provides a starting point for finding out more about various Python built-in functions that are not discussed in this chapter.

Note that while `dir()` does not list the names of built-in functions and variables, you can obtain this information from the standard module `__builtin__` that is automatically imported under the name `__builtins__`:

```
>>> dir(__builtins__)
```

The following command shows you how to get more information about a function:

```
help(str.lower)
```

The output from the preceding command is here:

```
Help on method_descriptor:

lower(...)
    S.lower() -> string

    Return a copy of the string S converted to lowercase.
(END)
```

Check the online documentation and also experiment with `help()` and `dir()` when you need additional information about a particular function or module.

COMPILE TIME AND RUNTIME CODE CHECKING

Python performs some compile-time checking, but most checks (including type, name, and so forth) are *deferred* until code execution. Consequently, if your Python code references a user-defined function that that does not exist, the code will compile successfully. In fact, the code will fail with an exception *only* when the code execution path references the nonexistent function.

As a simple example, consider the following Python function `myFunc` that references the nonexistent function called `DoesNotExist`:

```
def myFunc(x):
    if x == 3:
        print(DoesNotExist(x))
    else:
        print('x: ',x)
```

The preceding code will only fail when the `myFunc` function is passed the value 3, after which Python raises an error.

Now that you understand some basic concepts (such as how to use the Python interpreter) and how to launch your custom Python modules, the next section discusses primitive data types in Python.

SIMPLE DATA TYPES IN PYTHON

Python supports primitive data types, such as numbers (integers, floating point numbers, and exponential numbers), strings, and dates. Python also supports more complex data types, such as lists (or arrays), tuples, and dictionaries. The next several sections discuss some of the Python primitive data types, along with code snippets that show you how to perform various operations on those data types.

WORKING WITH NUMBERS

Python provides arithmetic operations for manipulating numbers in a straightforward manner that is similar to other programming languages. The following examples involve arithmetic operations on integers:

```
>>> 2+2
4
>>> 4/3
1
>>> 3*8
24
```

The following example assigns numbers to two variables and computes their product:

```
>>> x = 4
>>> y = 7
>>> x * y
28
```

The following examples demonstrate arithmetic operations involving integers:

```
>>> 2+2
4
>>> 4/3
1
>>> 3*8
24
```

Notice that division ("/") of two integers is actually truncation in which only the integer result is retained. The following example converts a floating-point number into exponential form:

```
>>> fnum = 0.00012345689000007
>>> "%.14e"%fnum
'1.23456890000070e-04'
```

You can use the `int()` function and the `float()` function to convert strings to numbers:

```
word1 = "123"
word2 = "456.78"
var1 = int(word1)
var2 = float(word2)
print("var1: ",var1," var2: ",var2)
```

The output from the preceding code block is here:

```
var1:  123  var2:  456.78
```

Alternatively, you can use the `eval()` function:

```
word1 = "123"
word2 = "456.78"
var1 = eval(word1)
var2 = eval(word2)
print("var1: ",var1," var2: ",var2)
```

If you attempt to convert a string that is not a valid integer or a floating-point number, Python raises an exception, so it's advisable to place your code in a `try/except` block (discussed later in this chapter).

Working with Other Bases

Numbers in Python are in base 10 (the default), but you can easily convert numbers to other bases. For example, the following code block initializes the variable `x` with the value `1234`, and then displays that number in base `2`, `8`, and `16`, respectively:

```
>>> x = 1234
>>> bin(x) '0b10011010010'
>>> oct(x) '0o2322'
>>> hex(x) '0x4d2'
```

Use the `format()` function if you want to suppress the `0b`, `0o`, or `0x` prefixes, as shown here:

```
>>> format(x, 'b') '10011010010'
>>> format(x, 'o') '2322'
>>> format(x, 'x') '4d2'
```

Negative integers are displayed with a negative sign:

```
>>> x = -1234
>>> format(x, 'b') '-10011010010'
>>> format(x, 'x') '-4d2'
```

The `chr()` Function

The Python `chr()` function takes a positive integer as a parameter and converts it to its corresponding alphabetic value (if one exists). The letters `A`

through Z have decimal representation of 65 through 91 (which corresponds to hexadecimal 41 through 5b), and the lowercase letters a through z have decimal representation 97 through 122 (hexadecimal 61 through 7b).

Here is an example of using the chr() function to print uppercase A:

```
>>> x=chr(65)
>>> x
'A'
```

The following code block prints the ASCII values for a range of integers:

```
result = ""
for x in range(65,91):
  print(x, chr(x))
  result = result+chr(x)+' '
print("result: ",result)
```

NOTE *Python 2 uses ASCII strings, whereas Python 3 uses UTF-8.*

You can represent a range of characters with the following line:

```
for x in range(65,91):
```

However, the following equivalent code snippet is more intuitive:

```
for x in range(ord('A'), ord('Z')):
```

If you want to display the result for lowercase letters, change the preceding range from (65,91) to either of the following statements:

```
for x in range(65,91):
for x in range(ord('a'), ord('z')):
```

The round() Function in Python

The Python round() function enables you to round decimal values to the nearest precision:

```
>>> round(1.23, 1)
1.2
>>> round(-3.42,1)
-3.4
```

Formatting Numbers in Python

Python allows you to specify the number of decimal places of precision to use when printing decimal numbers, as shown here:

```
>>> x = 1.23456
>>> format(x, '0.2f')
'1.23'
>>> format(x, '0.3f')
```

```
'1.235'
>>> 'value is {:0.3f}'.format(x) 'value is 1.235'
>>> from decimal import Decimal
>>> a = Decimal('4.2')
>>> b = Decimal('2.1')
>>> a + b
Decimal('6.3')
>>> print(a + b)
6.3
>>> (a + b) == Decimal('6.3')
True
>>> x = 1234.56789
>>> # Two decimal places of accuracy
>>> format(x, '0.2f')
'1234.57'
>>> # Right justified in 10 chars, one-digit accuracy
>>> format(x, '>10.1f')
' 1234.6'
>>> # Left justified
>>> format(x, '<10.1f') '1234.6 '
>>> # Centered
>>> format(x, '^10.1f') ' 1234.6 '
>>> # Inclusion of thousands separator
>>> format(x, ',')
'1,234.56789'
>>> format(x, '0,.1f')
'1,234.6'
```

WORKING WITH FRACTIONS

Python supports the `Fraction()` function (which is defined in the `fractions` module) that accepts two integers that represent the numerator and the denominator (which must be nonzero) of a fraction. Several examples of defining and manipulating fractions in Python are shown here:

```
>>> from fractions import Fraction
>>> a = Fraction(5, 4)
>>> b = Fraction(7, 16)
>>> print(a + b)
27/16
>>> print(a * b) 35/64
>>> # Getting numerator/denominator
>>> c = a * b
>>> c.numerator
35
>>> c.denominator 64
>>> # Converting to a float >>> float(c)
0.546875
>>> # Limiting the denominator of a value
>>> print(c.limit_denominator(8))
4
>>> # Converting a float to a fraction >>> x = 3.75
>>> y = Fraction(*x.as_integer_ratio())
```

```
>>> y
Fraction(15, 4)
```

Before delving into Python code samples that work with strings, the next section briefly discusses Unicode and UTF-8, both of which are character encodings.

UNICODE AND UTF-8

A Unicode string consists of a sequence of numbers that are between `0` and `0x10ffff`, where each number represents a group of bytes. An encoding is the manner in which a Unicode string is translated into a sequence of bytes. Among the various encodings, UTF-8 (Unicode Transformation Format) is perhaps the most common, and it's also the default encoding for many systems. The digit 8 in UTF-8 indicates that the encoding uses 8-bit numbers, whereas UTF-16 uses 16-bit numbers (but this encoding is less common).

The ASCII character set is a subset of UTF-8, so a valid ASCII string can be read as a UTF-8 string without any re-encoding required. In addition, a Unicode string can be converted into a UTF-8 string.

WORKING WITH UNICODE

Python supports Unicode, which means that you can render characters in different languages. Unicode data can be stored and manipulated in the same way as strings. Create a Unicode string by prepending the letter "u", as shown here:

```
>>> u'Hello from Python!'
u'Hello from Python!'
```

Special characters can be included in a string by specifying their Unicode value. For example, the following Unicode string embeds a space (which has the Unicode value 0x0020) in a string:

```
>>> u'Hello\u0020from Python!'
u'Hello from Python!'
```

Listing 1.1 displays the contents of `Unicode1.py` that illustrates how to display a string of characters in Japanese and another string of characters in Chinese (Mandarin).

LISTING 1.1: Unicode1.py

```
chinese1 = u'\u5c07\u63a2\u8a0e HTML5 \u53ca\u5176\u4ed6'
hiragana = u'D3 \u306F \u304B\u3063\u3053\u3043\u3043 \
                                                u3067\u3059!'

print('Chinese:',chinese1)
print('Hiragana:',hiragana)
```

The output of Listing 1.2 is here:

```
Chinese: 將探討 HTML5 及其他
Hiragana: D3 は かっこぃぃ です!
```

The next portion of this chapter shows you how to "slice and dice" text strings with built-in Python functions.

WORKING WITH STRINGS

A string in Python2 is a sequence of ASCII-encoded bytes. You can concatenate two strings using the "+" operator. The following example prints a string and then concatenates two single-letter strings:

```
>>> 'abc'
'abc'
>>> 'a' + 'b'
'ab'
```

You can use "+" or "*" to concatenate identical strings, as shown here:

```
>>> 'a' + 'a' + 'a'
'aaa'
>>> 'a' * 3
'aaa'
```

You can assign strings to variables and print them using the `print()` statement:

```
>>> print('abc')
abc
>>> x = 'abc'
>>> print(x)
abc
>>> y = 'def'
>>> print(x + y)
abcdef
```

You can "unpack" the letters of a string and assign them to variables, as shown here:

```
>>> str = "World"
>>> x1,x2,x3,x4,x5 = str
>>> x1
'W'
>>> x2
'o'
>>> x3
'r'
>>> x4
'l'
>>> x5
'd'
```

The preceding code snippets show you how easy it is to extract the letters in a text string. You can extract substrings of a string as shown in the following examples:

```
>>> x = "abcdef"
>>> x[0]
'a'
>>> x[-1]
'f'
>>> x[1:3]
'bc'
>>> x[0:2] + x[5:]
'abf'
```

However, you will cause an error if you attempt to subtract two strings, as you probably expect:

```
>>> 'a' - 'b'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for -: 'str' and
'str'
```

The `try/except` construct in Python (discussed later in this chapter) enables you to handle the preceding type of exception more gracefully.

Comparing Strings

You can use the methods `lower()` and `upper()` to convert a string to lowercase and uppercase, respectively, as shown here:

```
>>> 'Python'.lower()
'python'
>>> 'Python'.upper()
'PYTHON'
>>>
```

The methods `lower()` and `upper()` are useful for performing a case insensitive comparison of two ASCII strings. Listing 1.2 displays the contents of `Compare.py` that uses the `lower()` function in order to compare two ASCII strings.

LISTING 1.2: Compare.py

```
x = 'Abc'
y = 'abc'

if(x == y):
  print('x and y: identical')
elif (x.lower() == y.lower()):
  print('x and y: case insensitive match')
else:
  print('x and y: different')
```

Since x contains mixed case letters and y contains lowercase letters, Listing 1.2 displays the following output:

```
x and y: different
```

Formatting Strings in Python

Python provides the functions `string.lstring()`, `string.rstring()`, and `string.center()` for positioning a text string so that it is left-justified, right-justified, and centered, respectively. As you saw in a previous section, Python also provides the `format()` method for advanced interpolation features.

Now enter the following commands in the Python interpreter:

```
import string

str1 = 'this is a string'
print(string.ljust(str1, 10))
print(string.rjust(str1, 40))
print(string.center(str1,40))
```

The output is shown here:

```
this is a string
                         this is a string
            this is a string
```

UNINITIALIZED VARIABLES AND THE VALUE NONE IN PYTHON

Python distinguishes between an uninitialized variable and the value `None`. The former is a variable that has not been assigned a value, whereas the value `None` is a value that indicates "no value". Collections and methods often return the value `None`, and you can test for the value `None` in conditional logic.

The next portion of this chapter shows you how to "slice and dice" text strings with built-in Python functions.

SLICING AND SPLICING STRINGS

Python enables you to extract substrings of a string (called "slicing") using array notation. Slice notation is `start:stop:step`, where the start, stop, and step values are integers that specify the start value, end value, and the increment value. The interesting part about slicing in Python is that you can use the value `-1`, which operates from the right side instead of the left side of a string.

Some examples of slicing a string are here:

```
text1 = "this is a string"
print('First 7 characters:',text1[0:7])
print('Characters 2-4:',text1[2:4])
print('Right-most character:',text1[-1])
print('Right-most 2 characters:',text1[-3:-1])
```

The output from the preceding code block is here:

```
First 7 characters: this is
Characters 2-4: is
Right-most character: g
Right-most 2 characters: in
```

Later in this chapter you will see how to insert a string in the middle of another string.

Testing for Digits and Alphabetic Characters

Python enables you to examine each character in a string and then test whether that character is a bona fide digit or an alphabetic character. This section provides a simple introduction to regular expressions.

Listing 1.3 displays the contents of `CharTypes.py` that illustrates how to determine if a string contains digits or characters. Although we have not discussed if statements in Python, the examples in Listing 1.3 are straightforward.

LISTING 1.3: CharTypes.py

```
str1 = "4"
str2 = "4234"
str3 = "b"
str4 = "abc"
str5 = "a1b2c3"

if(str1.isdigit()):
  print("this is a digit:",str1)

if(str2.isdigit()):
  print("this is a digit:",str2)

if(str3.isalpha()):
  print("this is alphabetic:",str3)

if(str4.isalpha()):
  print("this is alphabetic:",str4)

if(not str5.isalpha()):
  print("this is not pure alphabetic:",str5)

print("capitalized first letter:",str5.title())
```

Listing 1.3 initializes some variables, followed by two conditional tests that check whether or not `str1` and `str2` are digits using the `isdigit()` function. The next portion of Listing 1.3 checks if `str3`, `str4`, and `str5` are alphabetic strings using the `isalpha()` function. The output of Listing 1.3 is here:

```
this is a digit: 4
this is a digit: 4234
this is alphabetic: b
this is alphabetic: abc
this is not pure alphabetic: a1b2c3
capitalized first letter: A1B2C3
```

SEARCH AND REPLACE A STRING IN OTHER STRINGS

Python provides methods for searching and also for replacing a string in a second text string. Listing 1.4 displays the contents of `FindPos1.py` that shows you how to use the find function to search for the occurrence of one string in another string.

LISTING 1.4: FindPos1.py

```
item1 = 'abc'
item2 = 'Abc'
text = 'This is a text string with abc'

pos1 = text.find(item1)
pos2 = text.find(item2)

print('pos1=',pos1)
print('pos2=',pos2)
```

Listing 1.4 initializes the variables `item1`, `item2`, and `text`, and then searches for the index of the contents of `item1` and `item2` in the string text. The Python `find()` function returns the column number where the first successful match occurs; otherwise, the `find()` function returns a `-1` if a match is unsuccessful.

The output from launching Listing 1.4 is here:

```
pos1= 27
pos2= -1
```

In addition to the `find()` method, you can use the `in` operator when you want to test for the presence of an element, as shown here:

```
>>> lst = [1,2,3]
>>> 1 in lst
True
```

Listing 1.5 displays the contents of `Replace1.py` that shows you how to replace one string with another string.

LISTING 1.5: Replace1.py

```
text = 'This is a text string with abc'
print('text:',text)
text = text.replace('is a', 'was a')
print('text:',text)
```

Listing 1.5 starts by initializing the variable text and then printing its contents. The next portion of Listing 1.5 replaces the occurrence of "is a" with "was a" in the string text, and then prints the modified string. The output from launching Listing 1.5 is here:

```
text: This is a text string with abc
text: This was a text string with abc
```

REMOVE LEADING AND TRAILING CHARACTERS

Python provides the functions `strip()`, `lstrip()`, and `rstrip()` to remove characters in a text string. Listing 1.6 displays the contents of `Remove1.py` that shows you how to search for a string.

LISTING 1.6: Remove1.py

```
text = '   leading and trailing white space   '
print('text1:','x',text,'y')

text = text.lstrip()
print('text2:','x',text,'y')

text = text.rstrip()
print('text3:','x',text,'y')
```

Listing 1.6 starts by concatenating the letter `x` and the contents of the variable `text`, and then printing the result. The second part of Listing 1.6 removes the leading white spaces in the string `text` and then appends the result to the letter `x`. The third part of Listing 1.6 removes the trailing white spaces in the string `text` (note that the leading white spaces have already been removed) and then appends the result to the letter `x`.

The output from launching Listing 1.6 is here:

```
text1: x    leading and trailing white space    y
text2: x leading and trailing white space    y
text3: x leading and trailing white space y
```

If you want to remove extra white spaces inside a text string, use the `replace()` function as discussed in the previous section. The following example illustrates how this can be accomplished, which also contains the `re` module for regular expressions:

```
import re
text = 'a    b'
a = text.replace(' ', '')
b = re.sub('\s+', ' ', text)

print(a)
print(b)
```

The result is here:

```
ab
a b
```

PRINTING TEXT WITHOUT NEW LINE CHARACTERS

If you need to suppress white space and a new line between objects output with multiple `print()` statements, you can use concatenation or the `write()` function.

The first technique is to concatenate the string representations of each object using the `str()` function prior to printing the result. For example, run the following statement in Python:

```
x = str(9)+str(0xff)+str(-3.1)
print('x: ',x)
```

The output is shown here:

```
x:  9255-3.1
```

The preceding line contains the concatenation of the numbers `9` and `255` (which is the decimal value of the hexadecimal number `0xff` and `-3.1`).

Incidentally, you can use the `str()` function with modules and user-defined classes. An example involving the Python built-in module `sys` is here:

```
>>> import sys
>>> print(str(sys))
<module 'sys' (built-in)>
```

The following code snippet illustrates how to use the `write()` function to display a string:

```
import sys
write = sys.stdout.write
write('123')
write('123456789')
```

The output is here:

```
1233
1234567899
```

TEXT ALIGNMENT

Python provides the methods `ljust()`, `rjust()`, and `center()` for aligning text. The `ljust()` and `rjust()` functions left justify and right justify a text string, respectively, whereas the `center()` function will center a string. An example is shown in the following code block:

```
text = 'Hello World'
text.ljust(20)
'Hello World '
>>> text.rjust(20)
' Hello World'
>>> text.center(20)
' Hello World '
```

You can use the Python `format()` function to align text. Use the `<`, `>`, or `^` characters, along with a desired width, in order to right justify, left justify, and center the text, respectively. The following examples illustrate how you can specify text justification:

```
>>> format(text, '>20')
'         Hello World'
>>>
>>> format(text, '<20')
'Hello World         '
>>>
>>> format(text, '^20')
'    Hello World     '
>>>
```

WORKING WITH DATES

Python provides a rich set of date-related functions that are documented here:

http://docs.python.org/2/library/datetime.html.

Listing 1.7 displays the contents of the Python script `Datetime2.py` that displays various date-related values, such as the current date and time; the day of the week, month, and year; and the time in seconds since the epoch.

LISTING 1.7: Datetime2.py

```
import time
import datetime

print("Time in seconds since the epoch: %s" %time.time())
print("Current date and time: " , datetime.datetime.now())
print("Or like this: " ,datetime.datetime.now().
                                  strftime("%y-%m-%d-%H-%M"))

print("Current year: ", datetime.date.today().strftime("%Y"))
print("Month of year: ", datetime.date.today().strftime("%B"))
print("Week number of the year: ", datetime.date.today().
                                              strftime("%W"))
print("Weekday of the week: ", datetime.date.today().
                                              strftime("%w"))
print("Day of year: ", datetime.date.today().strftime("%j"))
print("Day of the month : ", datetime.date.today().strftime("%d"))
print("Day of week: ", datetime.date.today().strftime("%A"))
```

Listing 1.8 displays the output generated by running the code in Listing 1.7.

LISTING 1.8: datetime2.out

```
Time in seconds since the epoch: 1375144195.66
Current date and time:  2013-07-29 17:29:55.664164
Or like this:  13-07-29-17-29
Current year:  2013
Month of year:  July
Week number of the year:  30
Weekday of the week:  1
Day of year:  210
```

```
Day of the month :  29
Day of week:  Monday
```

Python also enables you to perform arithmetic calculations with date-related values, as shown in the following code block:

```
>>> from datetime import timedelta
>>> a = timedelta(days=2, hours=6)
>>> b = timedelta(hours=4.5)
>>> c = a + b
>>> c.days
2
>>> c.seconds
37800
>>> c.seconds / 3600
10.5
>>> c.total_seconds() / 3600
58.5
```

Converting Strings to Dates

Listing 1.9 displays the contents of `String2Date.py` that illustrates how to convert a string to a date, and also how to calculate the difference between two dates.

LISTING 1.9: String2Date.py

```
from datetime import datetime

text = '2014-08-13'
y = datetime.strptime(text, '%Y-%m-%d')
z = datetime.now()
diff = z - y
print('Date difference:',diff)
```

The output from Listing 1.9 is shown here:

```
Date difference: -210 days, 18:58:40.197130
```

EXCEPTION HANDLING IN PYTHON

Unlike JavaScript you cannot add a number and a string in Python. Fortunately, you can detect an illegal operation using the `try/except` construct in Python, which is similar to the `try/catch` construct in languages such as JavaScript and Java.

An example of a `try/except` block is here:

```
try:
  x = 4
  y = 'abc'
  z = x + y
except:
  print('cannot add incompatible types:', x, y)
```

When you run the preceding code in Python, the `print()` statement in the `except` code block is executed because the variables `x` and `y` have incompatible types.

Earlier in the chapter you also saw that subtracting two strings throws an exception:

```
>>> 'a' - 'b'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for -: 'str' and 'str'
```

A simple way to handle this situation is to use a `try/except` block:

```
>>> try:
...  print('a' - 'b')
... except TypeError:
...  print('TypeError exception while trying to subtract
                                                  two strings')
... except:
...  print('Exception while trying to subtract two strings')
...
```

The output from the preceding code block is here:

```
TypeError exception while trying to subtract two strings
```

As you can see, the preceding code block specifies the finer-grained exception called `TypeError`, followed by a "generic" `except` code block to handle all other exceptions that might occur during the execution of your Python code. This style resembles the exception handling in Java code.

Listing 1.10 displays the contents of `Exception1.py` that illustrates how to handle various types of exceptions, which includes an exception due to a missing file.

LISTING 1.10: Exception1.py

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except IOError as err:
    print("I/O error: {0}".format(err))
except ValueError:
    print("Could not convert data to an integer.")
except:
    print("Unexpected error:", sys.exc_info()[0])
    raise
```

Listing 1.10 contains a `try` block followed by three `except` statements. If an error occurs in the `try` block, the first `except` statement is compared with the type of exception that occurred. If there is a match, then the subsequent `print()` statement is executed, and the program terminates. If not, a similar test is performed with the second `except` statement. If neither `except` statement matches the exception, the third `except` statement handles the exception, which involves printing a message and then "raising" an exception.

Note that you can also specify multiple exception types in a single statement, as shown here:

```
except (NameError, RuntimeError, TypeError):
    print('One of three error types occurred')
```

The preceding code block is more compact, but you do not know which of the three error types occurred. Python allows you to define custom exceptions, but this topic is beyond the scope of this book.

HANDLING USER INPUT

Python enables you to read user input from the command line via the `input()` function or the `raw_input()` function. Typically, you assign user input to a variable, which will contain all characters that users enter from the keyboard. User input terminates when users press the `<return>` key (which is included with the input characters). Listing 1.11 displays the contents of `UserInput1.py` that prompts users for their name and then uses interpolation to display a response.

LISTING 1.11: UserInput1.py

```
userInput = input("Enter your name: ")
print ("Hello %s, my name is Python" % userInput)
```

The output of Listing 1.11 is here (assume that the user entered the word `Dave`):

```
Hello Dave, my name is Python
```

The `print()` statement in Listing 1.11 uses string interpolation via `%s`, which substitutes the value of the variable after the `%` symbol. This functionality is obviously useful when you want to specify something that is determined at runtime.

User input can cause exceptions (depending on the operations that your code performs), so it's important to include exception-handling code.

Listing 1.12 displays the contents of `UserInput2.py` that prompts users for a string and attempts to convert the string to a number in a `try/except` block.

LISTING 1.12: UserInput2.py

```
userInput = input("Enter something: ")

try:
  x = 0 + eval(userInput)
  print('you entered the number:',userInput)
except:
  print(userInput,'is a string')
```

Listing 1.12 adds the number `0` to the result of converting a user's input to a number. If the conversion was successful, a message with the user's input is displayed. If the conversion failed, the `except` code block consists of a `print()` `and` statement that displays a message.

NOTE *This code sample uses the* `eval()` *function, which should be avoided so that your code does not evaluate arbitrary (and possibly destructive) commands.*

Listing 1.13 displays the contents of `UserInput3.py` that prompts users for two numbers and attempts to compute their sum in a pair of `try/except` blocks.

LISTING 1.13: UserInput3.py

```
sum = 0

msg = 'Enter a number:'
val1 = input(msg)

try:
  sum = sum + eval(val1)
except:
  print(val1,'is a string')

msg = 'Enter a number:'
val2 = input(msg)

try:
  sum = sum + eval(val2)
except:
  print(val2,'is a string')

print('The sum of',val1,'and',val2,'is',sum)
```

Listing 1.13 contains two `try` blocks, each of which is followed by an `except` statement. The first `try` block attempts to add the first user-supplied number to the variable `sum`, and the second `try` block attempts to add the second user-supplied number to the previously entered number. An error message occurs if either input string is not a valid number; if both are valid numbers, a message is displayed containing the input numbers and their sum. Be sure to read the caveat regarding the `eval()` function that is mentioned earlier in this chapter.

COMMAND-LINE ARGUMENTS

Python provides a `getopt` module to parse command-line options and arguments, and the Python `sys` module provides access to any command-line arguments via the `sys.argv`. This serves two purposes:

- `sys.argv` is the list of command-line arguments
- `len(sys.argv)` is the number of command-line arguments

Here `sys.argv[0]` is the program name, so if the Python program is called `test.py`, it matches the value of `sys.argv[0]`.

Now you can provide input values for a Python program on the command line instead of providing input values by prompting users for their input. As an example, consider the script `test.py` shown here:

```
#!/usr/bin/python
import sys
print('Number of arguments:',len(sys.argv),'arguments')
print('Argument List:', str(sys.argv))
```

Now run the previous script as follows:

```
python test.py arg1 arg2 arg3
```

This will produce following result:

```
Number of arguments: 4 arguments.
Argument List: ['test.py', 'arg1', 'arg2', 'arg3']
```

The ability to specify input values from the command line provides useful functionality. For example, suppose that you have a custom Python class that contains the methods `add` and `subtract` to add and subtract a pair of numbers.

You can use command-line arguments in order to specify which method to execute on a pair of numbers, as shown here:

```
python MyClass add 3 5
python MyClass subtract 3 5
```

This functionality is very useful because you can programmatically execute different methods in a Python class, which means that you can write unit tests for your code as well.

Listing 1.14 displays the contents of `Hello.py` that shows you how to use `sys.argv` to check the number of command-line parameters.

LISTING 1.14: Hello.py

```
import sys

def main():
  if len(sys.argv) >= 2:
```

```
    name = sys.argv[1]
  else:
    name = 'World'
  print('Hello', name)
# Standard boilerplate to invoke the main() function
if __name__ == '__main__':
  main()
```

Listing 1.14 defines the `main()` function that checks the number of command-line parameters: if this value is at least `2`, then the variable `name` is assigned the value of the second parameter (the first parameter is `Hello.py`), otherwise `name` is assigned the value `Hello`. The `print()` statement then prints the value of the variable `name`.

The final portion of Listing 1.14 uses conditional logic to determine whether or not to execute the `main()` function.

SUMMARY

This chapter showed you how to work with numbers and perform arithmetic operations on numbers, and then you learned how to work with strings and use string operations. The next chapter introduces you to data and how to perform various data-related operations.

CHAPTER 2

WORKING WITH DATA

This chapter introduces you to various data types (along with their differences), how to scale data values, and various techniques for handling missing data values. If most of the material in this chapter is new to you, be assured that it's not necessary to understand everything in this chapter. On the other hand, it's still a good idea to read as much material as you can absorb, and perhaps return to this chapter again after you have completed some of the other chapters in this book.

The first part of this chapter contains an overview of different types of data and an explanation of how to normalize and standardize a set of numeric values by calculating the mean and standard deviation of a set of numbers. You will see how to map categorical data to a set of integers and how to perform a one-hot encoding.

The second part of this chapter discusses missing data, outliers, and anomalies, and also some techniques for handling these scenarios. The third section discusses imbalanced data and the use of `SMOTE (Synthetic Minority Oversampling Technique)` to deal with imbalanced classes in a dataset.

The fourth section discusses ways to evaluate classifiers such as `LIME` and `ANOVA`. This section also contains details regarding the bias-variance trade-off and various types of statistical bias.

WHAT ARE DATASETS?

In simple terms, a dataset is a source of data (such as a text file) that contains rows and columns of data. Each row is typically called a "data point", and each column is called a "feature". A dataset can be a CSV (comma separated values), TSV (tab separated values), Excel spreadsheet, a table in an RDMBS (Relational Database Management Systems), a document in a NoSQL database, the output from a Web service, and so forth. As you will see, someone

needs to analyze the dataset to determine which features are the most important and which features can be safely ignored in order to train a model with the given dataset.

A dataset can vary from very small (a couple of features and 100 rows) to very large (more than 1,000 features and more than one million rows). If you are unfamiliar with the problem domain, then you might struggle to determine the most important features in a large dataset. In this situation, you might need a "domain expert" who understands the importance of the features, their interdependencies (if any), and whether or not the data values for the features are valid. In addition, there are algorithms (called dimensionality reduction algorithms) that can help you determine the most important features. For example, PCA (Principal Component Analysis) is one such algorithm, which is discussed in more detail later in this chapter.

Data Preprocessing

Data preprocessing is the initial step that involves validating the contents of a dataset, which involves making decisions about missing and incorrect data values:

- dealing with missing data values
- cleaning "noisy" text-based data
- removing HTML tags
- removing emoticons
- dealing with emojis/emoticons
- filtering data
- grouping data
- handling currency and date formats (i18n)

Cleaning data is a subset of data wrangling that involves removing unwanted data as well as handling missing data. In the case of text-based data, you might need to remove HTML tags, punctuation, and so forth. In the case of numeric data, it's less likely (though still possible) that alphabetic characters are mixed together with numeric data. However, a dataset with numeric features might have incorrect values or missing values (discussed later). In addition, calculating the minimum, maximum, mean, median, and standard deviation of the values of a feature obviously pertain only to numeric values.

After the preprocessing step is completed, data wrangling is performed, which refers to transforming data into a new format. You might have to combine data from multiple sources into a single dataset. For example, you might need to convert between different units of measurement (such as date formats, currency values, and so forth) so that the data values can be represented in a consistent manner in a dataset.

In case you didn't already know, currency and date values are part of something that is called i18n (internationalization), whereas l10n (localization) targets a specific nationality, language, or region. Hard-coded values (such as text

strings) can be stored as resource strings in a file that's often called a resource bundle, where each string is referenced via a code. Each language has its own resource bundle.

DATA TYPES

If you have written computer programs, then you know that explicit data types exist in many programming languages such as C, C++, Java, TypeScript, and so forth. Some programming languages, such as JavaScript and awk, do not require initializing variables with an explicit type: the type of a variable is inferred dynamically via an implicit type system (i.e., one that is not directly exposed to a developer).

In machine learning, datasets can contain features that have different types of data, such as a combination of one or more of the following:

- numeric data (integer/floating point and discrete/continuous)
- character/categorical data (different languages)
- date-related data (different formats)
- currency data (different formats)
- binary data (yes/no, 0/1, and so forth)
- nominal data (multiple unrelated values)
- ordinal data (multiple and related values)

Consider a dataset that contains real estate data, which can have as many as thirty columns (or even more), often with the following features:

- the number of bedrooms in a house: numeric value and a discrete value
- the number of square feet: a numeric value and (probably) a continuous value
- the name of the city: character data
- the construction date: a date value
- the selling price: a currency value and probably a continuous value
- the "for sale" status: binary data (either "yes" or "no")

An example of nominal data is the seasons in a year: although many (most?) countries have four distinct seasons, some countries have two distinct seasons. However, keep in mind that seasons can be associated with different temperature ranges (summer versus winter). An example of ordinal data is an employee pay grade: 1=entry level, 2=one year of experience, and so forth. Another example of nominal data is a set of colors, such as {Red, Green, Blue}.

An example of binary data is the pair {Male, Female}, and some datasets contain a feature with these two values. If such a feature is required for training a model, first convert {Male, Female} to a numeric counterpart, such as {0,1}. Similarly, if you need to include a feature whose values are the previous set of colors, you can replace {Red, Green, Blue} with the values {0,1,2}. Categorical data is discussed in more detail later in this chapter.

PREPARING DATASETS

If you have the good fortune to inherit a dataset that is in pristine condition, then data cleaning tasks (discussed later) are vastly simplified: in fact, it might not be necessary to perform *any* data cleaning for the dataset. On the other hand, if you need to create a dataset that combines data from multiple datasets that contain different formats for dates and currency, then you need to perform a conversion to a common format.

If you need to train a model that includes features that have categorical data, then you need to convert that categorical data to numeric data. For instance, the Titanic dataset contains a feature called "gender", which is either male or female. As you will see later in this chapter, Pandas makes it extremely simple to "map" male to 0 and female to 1.

Discrete Data Versus Continuous Data

As a simple rule of thumb: discrete data is a set of values that can be counted, whereas continuous data must be measured. Discrete data can "reasonably" fit in a drop-down list of values, but there is no exact value for making such a determination. One person might think that a list of 500 values is discrete, whereas another person might think it's continuous.

For example, the list of provinces of Canada and the list of states of the United States are discrete data values, but is the same true for the number of countries in the world (roughly 200) or for the number of languages in the world (more than 7,000)?

On the other hand, values for temperature, humidity, and barometric pressure are considered continuous. Currency is also treated as continuous, even though there is a measurable difference between two consecutive values. The smallest unit of currency for U.S. currency is one penny, which is 1/100th of a dollar (accounting-based measurements use the "mil", which is 1/1,000th of a dollar).

Continuous data types can have subtle differences. For example, someone who is 200 centimeters tall is twice as tall as someone who is 100 centimeters tall; the same is true for 100 kilograms versus 50 kilograms. However, temperature is different: 80 degrees Fahrenheit is not twice as hot as 40 degrees Fahrenheit.

Furthermore, keep in mind that the meaning of the word "continuous" in mathematics is not necessarily the same as continuous in machine learning. In the former, a continuous function (let's say in the 2D Euclidean plane) can have an uncountably infinite number of values. On the other hand, a feature in a dataset that can have more values than can be "reasonably" displayed in a drop-down list is treated *as though* it's a continuous variable.

For instance, values for stock prices are discrete: they must differ by at least a penny (or some other minimal unit of currency), which is to say, it's meaningless to say that the stock price changes by one-millionth of a penny. However, since there are "so many" possible stock values, it's treated as a continuous

variable. The same comments apply to car mileage, ambient temperature, barometric pressure, and so forth.

"Binning" Continuous Data

With the previous section in mind, the concept of "binning" refers to subdividing a set of values into multiple intervals, and then treating all the numbers in the same interval as though they had the same value.

As a simple example, suppose that a feature in a dataset contains the age of people in a dataset. The range of values is approximately between 0 and 120, and we could "bin" them into 12 equal intervals, where each consists of 10 values: 0 through 9, 10 through 19, 20 through 29, and so forth.

However, partitioning the values of people's ages as described in the preceding paragraph can be problematic. Suppose that person A, person B, and person C are 29, 30, and 39, respectively. Then person A and person B are probably more similar to each other than person B and person C, but because of the way in which the ages are partitioned, B is classified as closer to C than to A. In fact, binning can increase Type I errors (false positive) and Type II errors (false negative), as discussed in this blog post (along with some alternatives to binning):

https://medium.com/@peterflom/why-binning-continuous-data-is-almost-always-a-mistake-ad0b3a1d141f.

As another example, using quartiles is even more coarse-grained than the earlier age-related binning example. The issue with binning pertains to the consequences of classifying people in different bins, even though they are in close proximity to each other. For instance, some people struggle financially because they earn a meager wage, and they are disqualified from financial assistance because their salary is higher than the cutoff point for receiving any assistance.

Scaling Numeric Data via Normalization

A range of values can vary significantly, and it's important to note that they often need to be scaled to a smaller range, such as values in the range [–1,1] or [0,1], which you can do via the `tanh` function or the `sigmoid` function, respectively.

For example, measuring a person's height in terms of meters involves a range of values between 0.50 meters and 2.5 meters (in the vast majority of cases), whereas measuring height in terms of centimeters ranges between 50 centimeters and 250 centimeters: these two units differ by a factor of 100. A person's weight in kilograms generally varies between 5 kilograms and 200 kilograms, whereas measuring weight in grams differs by a factor of 1,000. Distances between objects can be measured in meters or in kilometers, which also differ by a factor of 1,000.

In general, use units of measure so that the data values in multiple features belong to a similar range of values. In fact, some machine learning algorithms require scaled data, often in the range of [0,1] or [–1,1]. In addition to the

tanh and sigmoid function, there are other techniques for scaling data, such as "standardizing" data (think Gaussian distribution) and "normalizing" data (linearly scaled so that the new range of values is in [0,1]).

The following examples involve a floating point variable X with different ranges of values that will be scaled so that the new values are in the interval [0,1].

Example 1: if the values of X are in the range [0,2], then X/2 is in the range [0,1].
Example 2: if the values of X are in the range [3,6], then X-3 is in the range [0,3], and (X-3)/3 is in the range [0,1].
Example 3: if the values of X are in the range [–10,20], then X +10 is in the range [0,30], and (X +10)/30 is in the range of [0,1].

In general, suppose that X is a random variable whose values are in the range [a,b], where a < b. You can scale the data values by performing two steps:

```
Step 1: X-a is in the range [0,b-a]
Step 2: (X-a)/(b-a) is in the range [0,1]
```

If X is a random variable that has the values {x1, x2, x3, . . ., xn}, then the formula for normalization involves mapping each xi value to (xi – min)/(max – min), where min is the minimum value of X and max is the maximum value of X.

As a simple example, suppose that the random variable X has the values {-1, 0, 1}. Then min and max are 1 and –1, respectively, and the normalization of {-1, 0, 1} is the set of values {(-1-(-1))/2, (0-(-1))/2, (1-(-1))/2}, which equals {0, 1/2, 1}.

Scaling Numeric Data via Standardization

The standardization technique involves finding the mean mu and the standard deviation sigma, and then mapping each xi value to (xi – mu)/sigma. Recall the following formulas:

```
mu = [SUM (x) ]/n
variance(x) = [SUM (x – xbar)*(x-xbar)]/n
sigma = sqrt(variance)
```

As a simple illustration of standardization, suppose that the random variable X has the values {–1, 0, 1}. Then mu and sigma are calculated as follows:

```
mu       = (SUM xi)/n = (-1 + 0 + 1)/3 = 0

variance = [SUM (xi- mu)^2]/n
         = [(-1-0)^2 + (0-0)^2 + (1-0)^2]/3
         = 2/3

sigma    = sqrt(2/3) = 0.816 (approximate value)
```

Hence, the standardization of {-1, 0, 1} is {-1/0.816, 0/0.816, 1/0.816}, which in turn equals the set of values {-1.2254, 0, 1.2254}.

As another example, suppose that the random variable X has the values {-6, 0, 6}. Then mu and sigma are calculated as follows:

```
mu       = (SUM xi)/n = (-6 + 0 + 6)/3 = 0

variance = [SUM (xi- mu)^2]/n
         = [(-6-0)^2 + (0-0)^2 + (6-0)^2]/3
         = 72/3
         = 24

sigma    = sqrt(24) = 4.899 (approximate value)
```

Hence, the standardization of {-6, 0, 6} is {-6/4.899, 0/4.899, 6/4.899}, which in turn equals the set of values {-1.2247, 0, 1.2247}.

In the preceding two examples, the mean equals 0 in both cases, but the variance and standard deviation are significantly different. One other point: the normalization of a set of values *always* produces a set of numbers between 0 and 1.

On the other hand, the standardization of a set of values can generate numbers that are less than –1 and greater than 1: this will occur when sigma is less than the minimum value of every term |mu – xi|, where the latter is the absolute value of the difference between mu and each xi value. In the preceding example, the minimum difference equals 1, whereas sigma is 0.816, and therefore the largest standardized value is greater than 1.

What to Look for in Categorical Data

This section contains various suggestions for handling inconsistent data values, and you can determine which ones to adopt based on any additional factors that are relevant to your particular task. For example, consider dropping columns that have very low cardinality (equal to or close to 1), as well as numeric columns with zero or very low variance.

Next, check the contents of categorical columns for inconsistent spellings or errors. A good example pertains to the gender category, which can consist of a combination of the following values:

```
male
Male
female
Female
m
f
M
F
```

The preceding categorical values for gender can be replaced with two categorical values (unless you have a valid reason to retain some of the other values). Moreover, if you are training a model whose analysis involves a single gender,

then you need to determine which rows (if any) of a dataset must be excluded. Also check categorical data columns for redundant or missing white spaces.

Check for data values that have multiple data types, such as a numerical column with numbers as numerals and some numbers as strings or objects. Also ensure consistent data formats: numbers as integers or floating numbers and ensure that dates have the same format (for example, do not mix `mm/dd/yyyy` date formats with another date format, such as `dd/mm/yyyy`).

Mapping Categorical Data to Numeric Values

Character data is often called categorical data, examples of which include people's names, home or work addresses, email addresses, and so forth. Many types of categorical data involve short lists of values. For example, the days of the week and the months in a year involve seven and twelve distinct values, respectively. Notice that the days of the week have a relationship: each day has a previous day and a next day, and similarly for the months of a year.

On the other hand, the colors of an automobile are independent of each other: the color red is not "better" or "worse" than the color blue. However, cars of a certain color can have a statistically higher number of accidents, but we won't address this case here.

There are several well-known techniques for mapping categorical values to a set of numeric values. A simple example where you need to perform this conversion involves the gender feature in the Titanic dataset. This feature is one of the relevant features for training a machine learning model. The gender feature has {M, F} as its set of possible values. As you will see later in this chapter, Pandas makes it very easy to convert the set of values {M,F} to the set of values {0,1}.

Another mapping technique involves mapping a set of categorical values to a set of consecutive integer values. For example, the set {Red, Green, Blue} can be mapped to the set of integers {0,1,2}. The set {Male, Female} can be mapped to the set of integers {0,1}. The days of the week can be mapped to {0,1,2,3,4,5,6}. Note that the first day of the week depends on the country: in some cases it's Sunday, and in other cases it's Monday.

Another technique is called *one-hot encoding*, which converts each value to a *vector* (check Wikipedia if you need a refresher regarding vectors). Thus, {Male, Female} can be represented by the vectors `[1,0]` and `[0,1]`, and the colors {Red, Green, Blue} can be represented by the vectors `[1,0,0]`, `[0,1,0]`, and `[0,0,1]`.

If you vertically "line up" the two vectors for gender, they form a 2x2 identity matrix, and doing the same for the colors will form a 3x3 identity matrix, as shown here:

```
[1,0,0]
[0,1,0]
[0,0,1]
```

If you are familiar with matrices, you probably noticed that the preceding set of vectors looks like the 3x3 identity matrix. In fact, this technique generalizes in a straightforward manner. Specifically, if you have n distinct categorical values, you can map each of those values to one of the vectors in an nxn identity matrix.

As another example, the set of titles `{"Intern", "Junior", "Mid-Range", "Senior", "Project Leader", "Dev Manager"}` have a hierarchical relationship in terms of their salaries (which can also overlap, but we'll gloss over that detail for now).

Another set of categorical data involves the season of the year: `{"Spring", "Summer", "Autumn", "Winter"}` and while these values are generally independent of each other, there are cases in which the season is significant. For example, the values for the monthly rainfall, average temperature, crime rate, or foreclosure rate can depend on the season, month, week, or even the day of the year.

If a feature has a large number of categorical values, then a one-hot encoding will produce many additional columns for each data point. Since the majority of the values in the new columns equal 0, this can increase the sparsity of the dataset, which in turn can result in more overfitting and hence adversely affect the accuracy of machine learning algorithms that you adopt during the training process.

Another solution is to use a sequence-based solution in which N categories are mapped to the integers 1, 2, . . . , N. Another solution involves examining the row frequency of each categorical value. For example, suppose that N equals 20, and there are three categorical values that occur in 95% of the values for a given feature. You can try the following:

1. assign the values 1, 2, and 3 to those three categorical values
2. assign numeric values that reflect the relative frequency of those categorical values
3. assign the category "OTHER" to the remaining categorical values
4. delete the rows whose categorical values belong to the 5%

Working with Dates

The format for a calendar date varies among different countries, and this belongs to something called *localization* of data (not to be confused with `i18n`, which is data internationalization). Some examples of date formats are shown as follows (and the first four are probably the most common):

```
MM/DD/YY
MM/DD/YYYY
DD/MM/YY
DD/MM/YYYY
YY/MM/DD
M/D/YY
```

```
D/M/YY
YY/M/D
MMDDYY
DDMMYY
YYMMDD
```

If you need to combine data from datasets that contain different date formats, then converting the disparate date formats to a single common date format will ensure consistency.

Working with Currency

The format for currency depends on the country, which includes different interpretations for a "," and "." in currency (and decimal values in general). For example, 1,124.78 equals "one thousand one hundred twenty-four point seven eight" in the United States, whereas 1.124,78 has the same meaning in Europe (i.e., the "." symbol and the "," symbol are interchanged).

If you need to combine data from datasets that contain different currency formats, then you probably need to convert all the disparate currency formats to a single common currency format. There is another detail to consider: currency exchange rates can fluctuate on a daily basis, which in turn can affect the calculation of taxes, late fees, and so forth. Although you might be fortunate enough where you won't have to deal with these issues, it's still worth being aware of them.

MISSING DATA, ANOMALIES, AND OUTLIERS

Although missing data is not directly related to checking for anomalies and outliers, in general you will perform all three of these tasks. Each task involves a set of techniques to help you perform an analysis of the data in a dataset, and the following subsections describe some of those techniques.

Missing Data

How you decide to handle missing data depends on the specific dataset. Here are some ways to handle missing data (the first three techniques are manual techniques, and the other techniques are algorithms):

1. replace missing data with the mean/median/mode value
2. infer ("impute") the value for missing data
3. delete rows with missing data
4. isolation forest (tree-based algorithm)
5. minimum covariance determinant
6. local outlier factor
7. one-class SVM (Support Vector Machines)

In general, replacing a missing numeric value with zero is a risky choice: this value is obviously incorrect if the values of a feature are between 1,000 and

5,000. For a feature that has numeric values, replacing a missing value with the average value is better than the value zero (unless the average equals zero); also consider using the median value. For categorical data, consider using the mode to replace a missing value.

If you are not confident that you can impute a "reasonable" value, consider dropping the row with a missing value, and then train a model with the imputed value and also with the deleted row.

One problem that can arise after removing rows with missing values is that the resulting dataset is too small. In this case, consider using `SMOTE`, which is discussed later in this chapter, in order to generate synthetic data.

Anomalies and Outliers

In simplified terms, an outlier is an abnormal data value that is outside the range of "normal" values. For example, a person's height in centimeters is typically between 30 centimeters and 250 centimeters. Hence, a data point (e.g., a row of data in a spreadsheet) with a height of 5 centimeters or a height of 500 centimeters is an outlier. The consequences of these outlier values are unlikely to involve a significant financial or physical loss (though they could adversely affect the accuracy of a trained model).

Anomalies are also outside the "normal" range of values (just like outliers), and they are typically more problematic than outliers: anomalies can have more "severe" consequences than outliers. For example, consider the scenario in which someone who lives in California suddenly makes a credit card purchase in New York. If the person is on vacation (or a business trip), then the purchase is an outlier (it's "outside" the typical purchasing pattern), but it's not an issue. However, if that person was in California when the credit card purchase was made, then it's more likely to be credit card fraud, as well as an anomaly.

Unfortunately, there is no simple way to *decide* how to deal with anomalies and outliers in a dataset. Although you can drop rows that contain outliers, keep in mind that doing so might deprive the dataset—and therefore the trained model—of valuable information. You can try modifying the data values (described as follows), but again, this might lead to erroneous inferences in the trained model. Another possibility is to train a model with the dataset that contains anomalies and outliers, and then train a model with a dataset from which the anomalies and outliers have been removed. Compare the two results and see if you can infer anything meaningful regarding the anomalies and outliers.

Outlier Detection

Although the decision to keep or drop outliers is your decision to make, there are some techniques available that help you detect outliers in a dataset. This section contains a short list of some techniques, along with a very brief description and links for additional information.

Perhaps *trimming* is the simplest technique (apart from dropping outliers), which involves removing rows whose feature value is in the upper 5% range or

the lower 5% range. *Winsorizing* the data is an improvement over trimming: set the values in the top 5% range equal to the maximum value in the 95th percentile, and set the values in the bottom 5% range equal to the minimum in the 5th percentile.

The *Minimum Covariance Determinant* is a covariance-based technique, and a Python-based code sample that uses this technique is downloadable here:

https://scikit-learn.org/stable/modules/outlier_detection.html.

The *Local Outlier Factor* (LOF) technique is an unsupervised technique that calculates a local anomaly score via the kNN (k Nearest Neighbor) algorithm. Documentation and short code samples that use LOF are here:

https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.LocalOutlierFactor.html.

Two other techniques involve the *Huber* and the *Ridge* classes, both of which are included as part of Sklearn. Huber error is less sensitive to outliers because it's calculated via linear loss, similar to MAE (Mean Absolute Error). A code sample that compares Huber and Ridge is downloadable here:

https://scikit-learn.org/stable/auto_examples/linear_model/plot_huber_vs_ridge.html.

You can also explore the Theil-Sen estimator and RANSAC, which are "robust" against outliers, and additional information is here:

https://scikit-learn.org/stable/auto_examples/linear_model/plot_theilsen.html and

https://en.wikipedia.org/wiki/Random_sample_consensus.

Four algorithms for outlier detection are discussed here:

https://www.kdnuggets.com/2018/12/four-techniques-outlier-detection.html.

One other scenario involves "local" outliers. For example, suppose that you use kMeans (or some other clustering algorithm) and determine that a value is an outlier with respect to one of the clusters. While this value is not necessarily an "absolute" outlier, detecting such a value might be important for your use case.

What Is Data Drift?

The value of data is based on its accuracy, its relevance, and its age. Data drift refers to data that has become less relevant over time. For example, online purchasing patterns in 2010 are probably not as relevant as data from 2020 because of various factors (such as the profile of different types of customers). Keep in mind that there might be multiple factors that can influence data drift in a specific dataset.

Two techniques are domain classifier and the black-box shift detector, both of which are discussed here:

https://blog.dataiku.com/towards-reliable-mlops-with-drift-detectors.

WHAT IS IMBALANCED CLASSIFICATION?

Imbalanced classification involves datasets with imbalanced classes. For example, suppose that class A has 99% of the data and class B has 1%. Which classification algorithm would you use? Unfortunately, classification algorithms don't work so well with this type of imbalanced dataset. Here is a list of several well-known techniques for handling imbalanced datasets:

- Random resampling rebalances the class distribution
- Random oversampling duplicates data in the minority class
- Random undersampling deletes examples from the majority class
- SMOTE

Random resampling transforms the training dataset into a new dataset, which is quite effective for imbalanced classification problems.

The *random undersampling* technique removes samples from the dataset, and involves the following:

- randomly remove samples from majority class
- can be performed with or without replacement
- alleviates imbalance in the dataset
- may increase the variance of the classifier
- may discard useful or important samples

However, random undersampling does not work so well with a dataset that has a 99%/1% split into two classes. Moreover, undersampling can result in losing information that is useful for a model.

Instead of random undersampling, another approach involves generating new samples from a minority class. The first technique involves oversampling examples in the minority class and duplicate examples from the minority class.

There is another technique that is better than the preceding technique, which involves the following:

- synthesize new examples from minority class
- a type of data augmentation for tabular data
- this technique can be very effective
- generate new samples from minority class

Another well-known technique is called SMOTE, which involves data augmentation (i.e., synthesizing new data samples) well before you use a classification algorithm. SMOTE was initially developed by means of the kNN algorithm (other options are available), and it can be an effective technique for handling imbalanced classes.

Yet another option to consider is the Python package `imbalanced-learn` in the `scikit-learn-contrib` project. This project provides various re-sampling techniques for datasets that exhibit class imbalance. More details are here:

https://github.com/scikit-learn-contrib/imbalanced-learn.

WHAT IS SMOTE?

`SMOTE` is a technique for synthesizing new samples for a dataset. This technique is based on linear interpolation:

Step 1: select samples that are close in the feature space
Step 2: draw a line between the samples in the feature space
Step 3: draw a new sample at a point along that line

A more detailed explanation of the `SMOTE` algorithm is here:

- select a random sample "a" from the minority class
- now find k nearest neighbors for that example
- select a random neighbor "b" from the nearest neighbors
- create a line L that connects "a" and "b"
- randomly select one or more points "c" on line L

If need be, you can repeat this process for the other (k-1) nearest neighbors in order to distribute the synthetic values more evenly among the nearest neighbors.

SMOTE Extensions

The initial `SMOTE` algorithm is based on the kNN classification algorithm, which has been extended in various ways, such as replacing kNN with SVM. A list of `SMOTE` extensions is shown as follows:

- selective synthetic sample generation
- Borderline-SMOTE (kNN)
- Borderline-SMOTE (SVM)
- Adaptive Synthetic Sampling (ADASYN)

Perform an Internet search for more details about these algorithms, and also navigate to the following URL:

https://en.wikipedia.org/wiki/Oversampling_and_undersampling_in_data_analysis.

ANALYZING CLASSIFIERS (OPTIONAL)

This section is marked optional because its contents pertain to machine learning classifiers, which are not the focus of this book. However, it's still worthwhile to glance through the material, or perhaps return to this section after you have a basic understanding of machine learning classifiers.

Several well-known techniques are available for analyzing the quality of machine learning classifiers. Two techniques are LIME and ANOVA, both of which are discussed in the following subsections.

What Is LIME?

LIME is an acronym for Local Interpretable Model-Agnostic Explanations. LIME is a model-agnostic technique that can be used with machine learning models. The intuition for this technique is straightforward: make small random changes to data samples and then observe the manner in which predictions change (or not). The intuition involves changing the output (slightly) and then observing what happens to the output.

By way of analogy, consider food inspectors who test for bacteria in truckloads of perishable food. Clearly, it's infeasible to test every food item in a truck (or a train car), so inspectors perform "spot checks" that involve testing randomly selected items. In an analogous fashion, LIME makes small changes to input data in random locations and then analyzes the changes in the associated output values.

However, there are two caveats to keep in mind when you use LIME with input data for a given model:

1. the actual changes to input values is model-specific
2. this technique works on input that is interpretable

Examples of interpretable input include machine learning classifiers (such as trees and random forests) and NLP techniques such as BoW (Bag of Words). Non-interpretable input involves "dense" data, such as a word embedding (which is a vector of floating point numbers).

You could also substitute your model with another model that involves interpretable data, but then you need to evaluate how accurate the approximation is to the original model.

What Is ANOVA?

ANOVA is an acronym for *analysis of variance*, which attempts to analyze the differences among the mean values of a sample that's taken from a population. ANOVA enables you to test if multiple mean values are equal. More importantly, ANOVA can assist in reducing Type I (false positive) errors and Type II errors (false negative) errors. For example, suppose that person A is diagnosed with cancer and person B is diagnosed as healthy, and that both diagnoses are incorrect. Then the result for person A is a false positive whereas the result for person B is a false negative. In general, a test result of false positive is *much* preferable to a test result of false negative.

ANOVA pertains to the design of experiments and hypothesis testing, which can produce meaningful results in various situations. For example, suppose that a dataset contains a feature that can be partitioned into several "reasonably" homogenous groups. Next, analyze the variance in each group and perform

comparisons with the goal of determining different sources of variance for the values of a given feature.

For more information about ANOVA, navigate to the following link: *https://en.wikipedia.org/wiki/Analysis_of_variance*.

THE BIAS-VARIANCE TRADE-OFF

This section is presented from the viewpoint of machine learning, but the concepts of bias and variance are highly relevant outside of machine learning, so it's probably still worthwhile to read this section as well as the previous section.

Bias in machine learning can be due to an error from wrong assumptions in a learning algorithm. High bias might cause an algorithm to miss relevant relations between features and target outputs (underfitting). Prediction bias can occur because of "noisy" data, an incomplete feature set, or a biased training sample.

Error due to bias is the difference between the expected (or average) prediction of your model and the correct value that you want to predict. Repeat the model building process multiple times, and gather new data each time, and also perform an analysis to produce a new model. The resulting models have a range of predictions because the underlying datasets have a degree of randomness. Bias measures the extent to which the predictions for these models deviate from the correct value.

Variance in machine learning is the expected value of the squared deviation from the mean. High variance can/might cause an algorithm to model the random noise in the training data, rather than the intended outputs (aka overfitting). Moreover, adding parameters to a model increases its complexity, increases the variance, and decreases the bias.

The point to remember: dealing with bias and variance involves dealing with underfitting and overfitting.

Error due to variance is the variability of a model prediction for a given data point. As before, repeat the entire model building process, and the variance is the extent to which predictions for a given point vary among different "instances" of the model.

If you have worked with datasets and performed data analysis, you already know that finding well-balanced samples can be difficult or highly impractical. Moreover, performing an analysis of the data in a dataset is vitally important, yet there is no guarantee that you can produce a dataset that is 100% "clean".

A *biased statistic* is a statistic that is systematically different from the entity in the population that is being estimated. In more casual terminology, if a data sample "favors" or "leans" toward one aspect of the population, then the sample has bias. For example, if you prefer movies that are comedies more so than any other type of movie, then clearly you are more likely to select a comedy instead of a dramatic movie or a science fiction movie. Thus, a frequency graph

of the movie types in a sample of your movie selections will be more closely clustered around comedies.

On the other hand, if you have a wide-ranging set of preferences for movies, then the corresponding frequency graph will be more varied, and therefore have a larger spread of values.

As a simple example, suppose that you are given an assignment that involves writing a term paper on a controversial subject that has many opposing viewpoints. Since you want a bibliography that supports your well-balanced term paper that takes into account multiple viewpoints, your bibliography will contain a wide variety of sources. In other words, your bibliography will have a larger variance and a smaller bias. On the other hand, if most (or all) the references in your bibliography espouse the same point of view, then you will have a smaller variance and a larger bias (it's just an analogy, so it's not a perfect counterpart to bias vs. variance).

The bias-variance trade-off can be stated in simple terms: in general, reducing the bias in samples can increase the variance, whereas reducing the variance tends to increase the bias.

Types of Bias in Data

In addition to the bias-variance trade-off that is discussed in the previous section, there are several types of bias, some of which are listed as follows:

- Availability Bias
- Confirmation Bias
- False Causality
- Sunk Cost Fallacy
- Survivorship Bias

Availability bias is akin to making a "rule" based on an exception. For example, there is a known link between smoking cigarettes and cancer, but there are exceptions. If you find someone who has smoked three packs of cigarettes on a daily basis for four decades and is still healthy, can you assert that smoking does not lead to cancer?

Confirmation bias refers to the tendency to focus on data that confirms one's beliefs and simultaneously ignore data that contradicts a belief.

False causality occurs when you incorrectly assert that the occurrence of a particular event causes another event to occur as well. One of the most well-known examples involves ice cream consumption and violent crime in New York during the summer. Since more people eat ice cream in the summer, that "causes" more violent crime, which is a false causality. Other factors, such as the increase in temperature, may be linked to the increase in crime. However, it's important to distinguish between correlation and causality: the latter is a much stronger link than the former, and it's also more difficult to establish causality instead of correlation.

Sunk cost refers to something (often money) that has been spent or incurred that cannot be recouped. A common example pertains to gambling at a casino: people fall into the pattern of spending more money in order to recoup a substantial amount of money that has already been lost. While there are cases in which people do recover their money, in many (most?) cases people simply incur an even greater loss because they continue to spend their money. Hence the expression, "it's time to cut your losses and walk away."

Survivorship bias refers to analyzing a particular subset of "positive" data while ignoring the "negative" data. This bias occurs in various situations, such as being influenced by individuals who recount their rags-to-riches success story ("positive" data) while ignoring the fate of the people (which is often a very high percentage) who did not succeed (the "negative" data) in a similar quest. So, while it's certainly possible for an individual to overcome many difficult obstacles in order to succeed, is the success rate one in one thousand (or even lower)?

SUMMARY

This chapter started with an explanation of datasets, a description of data wrangling, and details regarding various types of data. Then you learned about techniques for scaling numeric data, such as normalization and standardization. You saw how to convert categorical data to numeric values, and how to handle dates and currency.

Then you learned some of the nuances of missing data, anomalies, and outliers, and techniques for handling these scenarios. You also learned about imbalanced data and evaluating the use of `SMOTE` to deal with imbalanced classes in a dataset. In addition, you learned about classifiers using two techniques, `LIME` and `ANOVA`. Finally, you learned about the bias-variance trade-off and various types of statistical bias.

CHAPTER 3

INTRODUCTION TO NUMPY

This chapter provides a quick introduction to the Python `NumPy` package, which provides very useful functionality, not only for "regular" Python scripts, but also for Python-based scripts with TensorFlow. For instance, you will see `NumPy` code samples containing loops, arrays, and lists. You will also learn about dot products, the `reshape()` method (very useful!), how to plot with `Matplotlib` (discussed in more detail in Chapter 6), and examples of linear regression.

The first part of this chapter briefly discusses `NumPy` and some of its useful features. The second part contains examples of working arrays in `NumPy`, and contrasts some of the APIs for lists with the same APIs for arrays. In addition, you will see how easy it is to compute the exponent-related values (square, cube, and so forth) of elements in an array.

The second part of the chapter introduces subranges, which are very useful (and frequently used) for extracting portions of datasets in machine learning tasks. In particular, you will see code samples that handle negative (–1) subranges for vectors as well as for arrays, because they are interpreted one way for vectors and a different way for arrays.

The third part of this chapter delves into other `NumPy` methods, including the `reshape()` method, which is extremely useful (and very common) when working with image files: some `TensorFlow` APIs require converting a 2D array of `(R,G,B)` values into a corresponding one-dimensional vector.

The fourth part of this chapter delves into linear regression, the mean squared error (MSE), and how to calculate MSE with the `NumPy linspace()` API.

WHAT IS NUMPY?

`NumPy` is a Python library that provides many convenience methods and also better performance. `NumPy` provides a core library for scientific computing in

Python, with performant multidimensional arrays and good vectorized math functions, along with support for linear algebra and random numbers.

NumPy is modeled after MATLAB, with support for lists, arrays, and so forth. NumPy is easier to use than MATLAB, and it's very common in TensorFlow code as well as Python code.

Useful NumPy Features

The NumPy package provides the *ndarray* object that encapsulates multidimensional arrays of homogeneous data types. Many ndarray operations are performed in compiled code in order to improve performance.

Keep in mind the following important differences between NumPy arrays and the standard Python sequences. First, NumPy arrays have a fixed size, whereas Python lists can expand dynamically. Second, NumPy arrays are homogeneous, which means that the elements in a NumPy array must have the same data type. Third, NumPy arrays support more efficient execution (and require less code) of various types of operations on large numbers of data.

Now that you have a general idea about NumPy, let's delve into some examples that illustrate how to work with NumPy arrays, which is the topic of the next section.

WHAT ARE NUMPY ARRAYS?

An *array* is a set of consecutive memory locations used to store data. Each item in the array is called an *element*. The number of elements in an array is called the *dimension* of the array. A typical array declaration is shown here:

```
arr1 = np.array([1,2,3,4,5])
```

The preceding code snippet declares arr1 as an array of five elements, which you can access via arr1[0] through arr1[4]. Notice that the first element has an index value of 0, the second element has an index value of 1, and so forth. Thus, if you declare an array of 100 elements, then the 100th element has index value of 99.

NOTE *The first position in a* NumPy *array has index 0.*

NumPy treats arrays as vectors. Math ops are performed element by element. Remember the following difference: "doubling" an array *multiplies* each element by 2, whereas "doubling" a list *appends* a list to itself.

Listing 3.1 displays the contents of nparray1.py that illustrates some operations on a NumPy array.

LISTING 3.1: nparray1.py

```
import numpy as np

list1 = [1,2,3,4,5]
print(list1)
```

```
arr1  = np.array([1,2,3,4,5])
print(arr1)

list2 = [(1,2,3),(4,5,6)]
print(list2)

arr2  = np.array([(1,2,3),(4,5,6)])
print(arr2)
```

Listing 3.1 defines the variables `list1` and `list2` (which are Python lists), as well as the variables `arr1` and `arr2` (which are arrays), and prints their values. The output from launching Listing 3.1 is here:

```
[1, 2, 3, 4, 5]
[1 2 3 4 5]
[(1, 2, 3), (4, 5, 6)]
[[1 2 3]
 [4 5 6]]
```

As you can see, Python lists and arrays are very easy to define, and now we're ready to look at some loop operations for lists and arrays.

WORKING WITH LOOPS

Listing 3.2 displays the contents of `loop1.py` that illustrates how to iterate through the elements of a `NumPy` array and a Python list.

LISTING 3.2: loop1.py

```
import numpy as np

list = [1,2,3]
arr1 = np.array([1,2,3])

for e in list:
  print(e)

for e in arr1:
  print(e)

list1 = [1,2,3,4,5]
```

Listing 3.2 initializes the variable `list`, which is a Python list, and also the variable `arr1`, which is a `NumPy` array. The next portion of Listing 3.2 contains two loops, each of which iterates through the elements in `list` and `arr1`. As you can see, the syntax is identical in both loops. The output from launching Listing 3.2 is here:

```
1
2
3
1
```

```
2
3
```

APPENDING ELEMENTS TO ARRAYS (1)

Listing 3.3 displays the contents of `append1.py` that illustrates how to append elements to a `NumPy` array and a Python list.

LISTING 3.3: append1.py

```
import numpy as np

arr1 = np.array([1,2,3])

# these do not work:
#arr1.append(4)
#arr1 = arr1 + [5]

arr1 = np.append(arr1,4)
arr1 = np.append(arr1,[5])

for e in arr1:
  print(e)

arr2 = arr1 + arr1

for e in arr2:
  print(e)
```

Listing 3.3 initializes the variable `list`, which is a Python list, and also the variable `arr1`, which is a `NumPy` array. The output from launching Listing 3.3 is here:

```
1
2
3
4
5
2
4
6
8
10
```

APPENDING ELEMENTS TO ARRAYS (2)

Listing 3.4 displays the contents of `append2.py` that illustrates how to append elements to a `NumPy` array and a Python list.

LISTING 3.4: append2.py

```
import numpy as np

arr1 = np.array([1,2,3])
arr1 = np.append(arr1,4)

for e in arr1:
  print(e)

arr1 = np.array([1,2,3])
arr1 = np.append(arr1,4)

arr2 = arr1 + arr1

for e in arr2:
  print(e)
```

Listing 3.4 initializes the variable `arr1`, which is a `NumPy` array. Notice that `NumPy` arrays do not have an "append" method: this method is available through `NumPy` itself. Another important difference between Python lists and `NumPy` arrays: the "+" operator *concatenates* Python lists, whereas this operator *doubles* the elements in a `NumPy` array. The output from launching Listing 3.4 is here:

```
1
2
3
4
2
4
6
8
```

MULTIPLYING LISTS AND ARRAYS

Listing 3.5 displays the contents of `multiply1.py` that illustrates how to multiply elements in a Python list and a `NumPy` array.

LISTING 3.5: multiply1.py

```
import numpy as np

list1 = [1,2,3]
arr1  = np.array([1,2,3])
print('list:  ',list1)
print('arr1:  ',arr1)
print('2*list:',2*list)
print('2*arr1:',2*arr1)
```

Listing 3.5 contains a Python list called `list` and a `NumPy` array called `arr1`. The `print()` statements display the contents of `list` and `arr1` as well

as the result of doubling `list1` and `arr1`. Recall that "doubling" a Python list is different from doubling a Python array, which you can see in the output from launching Listing 3.5:

```
('list:  ', [1, 2, 3])
('arr1:  ', array([1, 2, 3]))
('2*list:', [1, 2, 3, 1, 2, 3])
('2*arr1:', array([2, 4, 6]))
```

DOUBLING THE ELEMENTS IN A LIST

Listing 3.6 displays the contents of `double_list1.py` that illustrates how to double the elements in a Python list.

LISTING 3.6: double_list1.py

```
import numpy as np

list1 = [1,2,3]
list2 = []

for e in list1:
  list2.append(2*e)

print('list1:',list1)
print('list2:',list2)
```

Listing 3.6 contains a Python list called `list1` and an empty `NumPy` list called `list2`. The next code snippet iterates through the elements of `list1` and appends them to the variable `list2`. The pair of `print()` statements display the contents of `list1` and `list2` to show you that they are the same. The output from launching Listing 3.6 is here:

```
('list: ', [1, 2, 3])
('list2:', [2, 4, 6])
```

LISTS AND EXPONENTS

Listing 3.7 displays the contents of `exponent_list1.py` that illustrates how to compute exponents of the elements in a Python list.

LISTING 3.7: exponent_list1.py

```
import numpy as np

list1 = [1,2,3]
list2 = []

for e in list1:
  list2.append(e*e) # e*e = squared
```

```
print('list1:',list1)
print('list2:',list2)
```

Listing 3.7 contains a Python list called `list1` and an empty `NumPy` list called `list2`. The next code snippet iterates through the elements of `list1` and appends the square of each element to the variable `list2`. The pair of `print()` statements display the contents of `list1` and `list2`. The output from launching Listing 3.7 is here:

```
('list1:', [1, 2, 3])
('list2:', [1, 4, 9])
```

ARRAYS AND EXPONENTS

Listing 3.8 displays the contents of `exponent_array1.py` that illustrates how to compute exponents of the elements in a `NumPy` array.

LISTING 3.8: exponent_array1.py

```
import numpy as np

arr1 = np.array([1,2,3])
arr2 = arr1**2
arr3 = arr1**3

print('arr1:',arr1)
print('arr2:',arr2)
print('arr3:',arr3)
```

Listing 3.8 contains a `NumPy` array called `arr1` followed by two `NumPy` arrays called `arr2` and `arr3`. Notice the compact manner in which the `NumPy` `arr2` is initialized with the square of the elements in in `arr1`, followed by the initialization of the `NumPy` array `arr3` with the cube of the elements in `arr1`. The three `print()` statements display the contents of `arr1`, `arr2`, and `arr3`. The output from launching Listing 3.8 is here:

```
('arr1:', array([1, 2, 3]))
('arr2:', array([1, 4, 9]))
('arr3:', array([ 1,  8, 27]))
```

MATH OPERATIONS AND ARRAYS

Listing 3.9 displays the contents of `mathops_array1.py` that illustrates how to perform math operations of the elements in a `NumPy` array.

LISTING 3.9: mathops_array1.py

```
import numpy as np

arr1 = np.array([1,2,3])
```

```
sqrt = np.sqrt(arr1)
log1 = np.log(arr1)
exp1 = np.exp(arr1)

print('sqrt:',sqrt)
print('log1:',log1)
print('exp1:',exp1)
```

Listing 3.9 contains a `NumPy` array called `arr1` followed by three `NumPy` arrays called `sqrt`, `log1`, and `exp1` that are initialized with the square root, the log, and the exponential value of the elements in `arr1`, respectively. The three `print()` statements display the contents of `sqrt`, `log1`, and `exp1`. The output from launching Listing 3.9 is here:

```
('sqrt:', array([1.        , 1.41421356, 1.73205081]))
('log1:', array([0.        , 0.69314718, 1.09861229]))
('exp1:', array([2.71828183, 7.3890561,  20.08553692]))
```

WORKING WITH "–1" SUBRANGES WITH VECTORS

Listing 3.10 displays the contents of `npsubarray2.py` that illustrates how to display subranges of the elements in a `NumPy` array.

LISTING 3.10: npsubarray2.py

```
import numpy as np

# -1 => "all except the last element in …" (row or col)

arr1  = np.array([1,2,3,4,5])
print('arr1:',arr1)
print('arr1[0:-1]:',arr1[0:-1])
print('arr1[1:-1]:',arr1[1:-1])
print('arr1[::-1]:', arr1[::-1]) # reverse!
```

Listing 3.10 contains a `NumPy` array called `arr1` followed by four `print` statements, each of which displays a different subrange of values in `arr1`. The output from launching Listing 3.10 is here:

```
('arr1:',       array([1, 2, 3, 4, 5]))
('arr1[0:-1]:', array([1, 2, 3, 4]))
('arr1[1:-1]:', array([2, 3, 4]))
('arr1[::-1]:', array([5, 4, 3, 2, 1]))
```

WORKING WITH "–1" SUBRANGES WITH ARRAYS

Listing 3.11 displays the contents of `np2darray2.py` that illustrates how to display subranges of the elements in a `NumPy` array.

LISTING 3.11: np2darray2.py

```
import numpy as np

# -1 => "the last element in …" (row or col)

arr1  = np.array([(1,2,3),(4,5,6),(7,8,9),(10,11,12)])
print('arr1:',         arr1)
print('arr1[-1,:]:',  arr1[-1,:])
print('arr1[:,-1]:',  arr1[:,-1])
print('arr1[-1:,-1]:',arr1[-1:,-1])
```

Listing 3.11 contains a NumPy array called arr1 followed by four print statements, each of which displays a different subrange of values in arr1. The output from launching Listing 3.11 is here:

```
(arr1:', array([[1,  2,  3],
                [4,  5,  6],
                [7,  8,  9],
                [10, 11, 12]]))
(arr1[-1,:]]',   array([10, 11, 12]))
(arr1[:,-1]:',   array([3,  6,  9, 12]))
(arr1[-1:,-1]]', array([12]))
```

OTHER USEFUL NUMPY METHODS

In addition to the NumPy methods that you saw in the code samples prior to this section, the following (often intuitively named) NumPy methods are also very useful:

- The method np.zeros() initializes an array with 0 values.
- The method np.ones() initializes an array with 1 value.
- The method np.empty() initializes an array with 0 values.
- The method np.arange() provides a range of numbers.
- The method np.shape() displays the shape of an object.
- The method np.reshape() <= *very useful!*
- The method np.linspace() <= *useful in regression.*
- The method np.mean() computes the mean of a set of numbers.
- The method np.std() computes the standard deviation of a set of numbers.

Although the np.zeros() and np.empty() both initialize a 2D array with 0, np.zeros() requires less execution time. You could also use np.full(size, 0), but this method is the slowest of all three methods.

The reshape() method and the linspace() method are very useful for changing the dimensions of an array and generating a list of numeric values, respectively. The reshape() method often appears in TensorFlow code, and the linspace() method is useful for generating a set of numbers in linear regression (discussed later). The mean() and std() methods are useful for

calculating the mean and the standard deviation of a set of numbers. For example, you can use these two methods in order to resize the values in a Gaussian distribution so that their mean is 0 and the standard deviation is 1. This process is called *standardizing* a Gaussian distribution.

ARRAYS AND VECTOR OPERATIONS

Listing 3.12 displays the contents of `array_vector.py` that illustrates how to perform vector operations on the elements in a `NumPy` array.

LISTING 3.12: array_vector.py

```
import numpy as np

a = np.array([[1,2], [3, 4]])
b = np.array([[5,6], [7,8]])

print('a:       ', a)
print('b:       ', b)
print('a + b:   ', a+b)
print('a - b:   ', a-b)
print('a * b:   ', a*b)
print('a / b:   ', a/b)
print('b / a:   ', b/a)
print('a.dot(b):',a.dot(b))
```

Listing 3.12 contains two `NumPy` arrays called `a` and `b` followed by eight `print` statements, each of which displays the result of "applying" a different arithmetic operation to the `NumPy` arrays `a` and `b`. The output from launching Listing 3.12 is here:

```
('a    :   ', array([[1, 2], [3, 4]]))
('b    :   ', array([[5, 6], [7, 8]]))
('a + b:   ', array([[ 6,  8], [10, 12]]))
('a - b:   ', array([[-4, -4], [-4, -4]]))
('a * b:   ', array([[ 5, 12], [21, 32]]))
('a / b:   ', array([[0, 0], [0, 0]]))
('b / a:   ', array([[5, 3], [2, 2]]))
('a.dot(b):', array([[19, 22], [43, 50]]))
```

NUMPY AND DOT PRODUCTS (1)

Listing 3.13 displays the contents of `dotproduct1.py` that illustrates how to perform the dot product on the elements in a `NumPy` array.

LISTING 3.13: dotproduct1.py

```
import numpy as np

a = np.array([1,2])
b = np.array([2,3])
```

```
dot2 = 0
for e,f in zip(a,b):
  dot2 += e*f

print('a:   ',a)
print('b:   ',b)
print('a*b: ',a*b)
print('dot1:',a.dot(b))
print('dot2:',dot2)
```

Listing 3.13 contains two NumPy arrays called a and b followed by a simple loop that computes the dot product of a and b. The next section contains five print statements that display the contents of a and b, as well as their inner product that's calculated in three different ways. The output from launching Listing 3.13 is here:

```
('a:   ', array([1, 2]))
('b:   ', array([2, 3]))
('a*b: ', array([2, 6]))
('dot1:', 8)
('dot2:', 8)
```

NUMPY AND DOT PRODUCTS (2)

NumPy arrays support a "dot" method for calculating the inner product of an array of numbers, which uses the same formula that you use for calculating the inner product of a pair of vectors. Listing 3.14 displays the contents of dotproduct2.py that illustrates how to calculate the dot product of two NumPy arrays.

LISTING 3.14: dotproduct2.py

```
import numpy as np

a = np.array([1,2])
b = np.array([2,3])

print('a:          ',a)
print('b:          ',b)
print('a.dot(b):   ',a.dot(b))
print('b.dot(a):   ',b.dot(a))
print('np.dot(a,b):',np.dot(a,b))
print('np.dot(b,a):',np.dot(b,a))
```

Listing 3.14 contains two NumPy arrays called a and b followed by six print statements that display the contents of a and b, and also their inner product that's calculated in three different ways. The output from launching Listing 3.14 is here:

```
('a:          ', array([1, 2]))
('b:          ', array([2, 3]))
('a.dot(b):   ', 8)
```

```
('b.dot(a):   ', 8)
('np.dot(a,b):', 8)
('np.dot(b,a):', 8)
```

NUMPY AND THE LENGTH OF VECTORS

The *norm* of a vector (or an array of numbers) is the length of a vector, which is the square root of the dot product of a vector with itself. NumPy also provides the "sum" and "square" functions that you can use to calculate the norm of a vector.

Listing 3.15 displays the contents of array_norm.py that illustrates how to calculate the magnitude ("norm") of a NumPy array of numbers.

LISTING 3.15: array_norm.py

```
import numpy as np

a = np.array([2,3])
asquare = np.square(a)
asqsum  = np.sum(np.square(a))
anorm1  = np.sqrt(np.sum(a*a))
anorm2  = np.sqrt(np.sum(np.square(a)))
anorm3  = np.linalg.norm(a)

print('a:      ',a)
print('asquare:',asquare)
print('asqsum: ',asqsum)
print('anorm1: ',anorm1)
print('anorm2: ',anorm2)
print('anorm3: ',anorm3)
```

Listing 3.15 contains an initial NumPy array called a, followed by the NumPy array asquare and the numeric values asqsum, anorm1, anorm2, and anorm3. The NumPy array asquare contains the square of the elements in the NumPy array a, and the numeric value asqsum contains the sum of the elements in the NumPy array asquare. Next, the numeric value anorm1 equals the square root of the sum of the square of the elements in a. The numeric value anorm2 is the same as anorm1, computed in a slightly different fashion. Finally, the numeric value anorm3 is equal to anorm2, but as you can see, anorm3 is calculated via a single NumPy method, whereas anorm2 requires a succession of NumPy methods.

The last portion of Listing 3.15 consists of six print statements, each of which displays the computed values. The output from launching Listing 3.15 is here:

```
('a:      ', array([2, 3]))
('asquare:', array([4, 9]))
('asqsum: ', 13)
('anorm1: ', 3.605551275463989)
```

```
('anorm2: ', 3.605551275463989)
('anorm3: ', 3.605551275463989)
```

NUMPY AND OTHER OPERATIONS

NumPy provides the "*" operator to multiply the components of two vectors to produce a third vector whose components are the products of the corresponding components of the initial pair of vectors. This operation is called a "Hadamard" product, which is the name of a famous mathematician. If you then add the components of the third vector, the sum is equal to the inner product of the initial pair of vectors.

Listing 3.16 displays the contents of otherops.py that illustrates how to perform other operations on a NumPy array.

LISTING 3.16: otherops.py

```
import numpy as np

a = np.array([1,2])
b = np.array([3,4])

print('a:           ',a)
print('b:           ',b)
print('a*b:         ',a*b)
print('np.sum(a*b): ',np.sum(a*b))
print('(a*b.sum()): ',(a*b).sum())
```

Listing 3.16 contains two NumPy arrays called a and b followed five print statements that display the contents of a and b, their Hadamard product, and also their inner product that's calculated in two different ways. The output from launching Listing 3.16 is here:

```
('a:           ', array([1, 2]))
('b:           ', array([3, 4]))
('a*b:         ', array([3, 8]))
('np.sum(a*b): ', 11)
('(a*b.sum()): ', 11)
```

NUMPY AND THE RESHAPE() METHOD

NumPy arrays support the "reshape" method that enables you to restructure the dimensions of an array of numbers. In general, if a NumPy array contains m elements, where m is a positive integer, then that array can be restructured as an m1 x m2 NumPy array, where m1 and m2 are positive integers such that m1*m2 = m.

Listing 3.17 displays the contents of numpy_reshape.py that illustrates how to use the reshape() method on a NumPy array.

LISTING 3.17: numpy_reshape.py

```
import numpy as np

x = np.array([[2, 3], [4, 5], [6, 7]])
print(x.shape) # (3, 2)

x = x.reshape((2, 3))
print(x.shape) # (2, 3)
print('x1:',x)

x = x.reshape((-1))
print(x.shape) # (6,)
print('x2:',x)

x = x.reshape((6, -1))
print(x.shape) # (6, 1)
print('x3:',x)

x = x.reshape((-1, 6))
print(x.shape) # (1, 6)
print('x4:',x)
```

Listing 3.17 contains a `NumPy` array called `x` whose dimensions are 3x2, followed by a set of invocations of the `reshape()` method that reshape the contents of `x`. The first invocation of the `reshape()` method changes the shape of `x` from 3x2 to 2x3. The second invocation changes the shape of `x` from 2x3 to 6x1. The third invocation changes the shape of `x` from 1x6 to 6x1. The final invocation changes the shape of `x` from 6x1 to 1x6 again.

Each invocation of the `reshape()` method is followed by a `print()` statement so that you can see the effect of the invocation. The output from launching Listing 3.17 is here:

```
(3, 2)
(2, 3)
('x1:', array([[2, 3, 4],
       [5, 6, 7]]))
(6,)
('x2:', array([2, 3, 4, 5, 6, 7]))
(6, 1)
('x3:', array([[2],
       [3],
       [4],
       [5],
       [6],
       [7]]))
(1, 6)
```

CALCULATING THE MEAN AND STANDARD DEVIATION

If you need to review these concepts from statistics (and perhaps also the mean, median, and mode as well), please read the appropriate online tutorials.

NumPy provides various built-in functions that perform statistical calculations, such as the following list of methods:

- np.linspace() <= useful for regression
- np.mean()
- np.std()

The np.linspace() method generates a set of equally spaced numbers between a lower bound and an upper bound. The np.mean() and np.std() methods calculate the mean and standard deviation, respectively, of a set of numbers. Listing 3.18 displays the contents of sample_mean_std.py that illustrates how to calculate statistical values from a NumPy array.

LISTING 3.18: sample_mean_std.py

```
import numpy as np

x2 = np.arange(8)
print('mean = ',x2.mean())
print('std  = ',x2.std())

x3 = (x2 - x2.mean())/x2.std()
print('x3 mean = ',x3.mean())
print('x3 std  = ',x3.std())
```

Listing 3.18 contains a NumPy array x2 that consists of the first eight integers. Next, the mean() and std() that are "associated" with x2 are invoked in order to calculate the mean and standard deviation, respectively, of the elements of x2. The output from launching Listing 3.18 is here:

```
('a:           ', array([1, 2]))
('b:           ', array([3, 4]))
```

CODE SAMPLE WITH MEAN AND STANDARD DEVIATION

The code sample in this section extends the code sample in the previous section with additional statistical values, and the code in Listing 3.19 can be used for any data distribution. Keep in mind that the code sample uses random numbers simply for the purposes of illustration: after you have launched the code sample, replace those numbers with values from a CSV file or some other dataset containing meaningful values.

Moreover, this section does not provide details regarding the meaning of quartiles, but you can learn about quartiles here:

https://en.wikipedia.org/wiki/Quartile.

Listing 3.19 displays the contents of stat_values.py that illustrates how to display various statistical values from a NumPy array of random numbers.

LISTING 3.19: stat_values.py

```
import numpy as np

from numpy import percentile
from numpy.random import rand

# generate data sample
data = np.random.rand(1000)

# calculate quartiles, min, and max
quartiles = percentile(data, [25, 50, 75])
data_min, data_max = data.min(), data.max()

# print summary information
print('Minimum:  %.3f' % data_min)
print('Q1 value: %.3f' % quartiles[0])
print('Median:   %.3f' % quartiles[1])
print('Mean Val: %.3f' % data.mean())
print('Std Dev:  %.3f' % data.std())
print('Q3 value: %.3f' % quartiles[2])
print('Maximum:  %.3f' % data_max)
```

The data sample (shown in bold) in Listing 3.19 is from a uniform distribution between 0 and 1. The `NumPy percentile()` function calculates a linear interpolation (average) between observations, which is needed to calculate the median on a sample with an even number of values. As you can surmise, the `NumPy` functions `min()` and `max()` calculate the smallest and largest values in the data sample. The output from launching Listing 3.19 is here:

```
Minimum:  0.000
Q1 value: 0.237
Median:   0.500
Mean Val: 0.495
Std Dev:  0.295
Q3 value: 0.747
Maximum:  0.999
```

Trimmed Mean and Weighted Mean

In addition to the arithmetic mean, there are variants that are known as a weighted mean and a trimmed mean (also called a truncated mean).

A *trimmed mean* is a robust estimate (i.e., a metric that is not sensitive to outliers). As a simple example of a trimmed mean, suppose that you have five scores for the evaluation of a product: simply drop the highest and lowest scores and then compute the average of the remaining three scores. If you have multiple sets of five scores, repeat the preceding process and then compute the average of the set of trimmed mean values.

A *weighted mean* is useful when sample data does not represent different groups in a dataset. Assigning a larger weight to groups that are underrepresented yields a weighted mean that more accurately represents the various

groups in the dataset. However, keep in mind that outliers can affect the mean as well as the weighted mean.

The weighted mean is the same as the expected value. In case you are unfamiliar with the notion of an expected value, suppose that the set `P = {p1,p2,...,pn}` is a probability distribution, which means that the numeric values in the set P must be nonnegative and have a sum equal to 1. In addition, suppose that `V = {v1,v2,...,vn}` is a set of numeric scores that are assigned to n feature of a product M. The values in the set V are probably positive integers in some range (e.g., between 1 and 10).

Then the *expected value* E for that product is computed as follows:

```
E = p1*v1 + p2*v2 + ... + pn*vn
```

The final chapter contains more detailed information about `matplotlib` in order to plot various charts and graphs. However, the Python code samples in the next several sections contain some rudimentary APIs from `matplotlib`. The code samples start with simple examples of line segments, followed by an introduction to linear regression.

WORKING WITH LINES IN THE PLANE (OPTIONAL)

This section contains a short review of lines in the Euclidean plane, so you can skip this section if you are comfortable with this topic. A minor point that's often overlooked is that lines in the Euclidean plane have infinite length. If you select two distinct points of a line, then all the points between those two selected points is a *line segment*. A *ray* is a "half infinite" line: when you select one point as an endpoint, then all the points on one side of the line constitutes a ray.

For example, the points in the plane whose y-coordinate is 0 is a line and also the x-axis, whereas the points between (0,0) and (1,0) on the x-axis form a line segment. In addition, the points on the x-axis that are to the right of (0,0) form a ray, and the points on the x-axis that are to the left of (0,0) also form a ray.

For simplicity and convenience, in this book we'll use the terms "line" and "line segment" interchangeably, and now let's delve into the details of lines in the Euclidean plane. Just in case you're a bit fuzzy on the details, here is the equation of a (nonvertical) line in the Euclidean plane:

```
y = m*x + b
```

The value of `m` is the slope of the line and the value of `b` is the y-intercept (i.e., the place where the nonvertical line intersects the y-axis). In case you're wondering, the following form for a line in the plane is a more general equation that includes vertical lines:

```
a*x + b*y + c = 0
```

However, we won't be working with vertical lines, so we'll stick with the first formula. Figure 3.1 displays three horizontal lines whose equations (from top to bottom) are `y = 3`, `y = 0`, and `y = -3`, respectively.

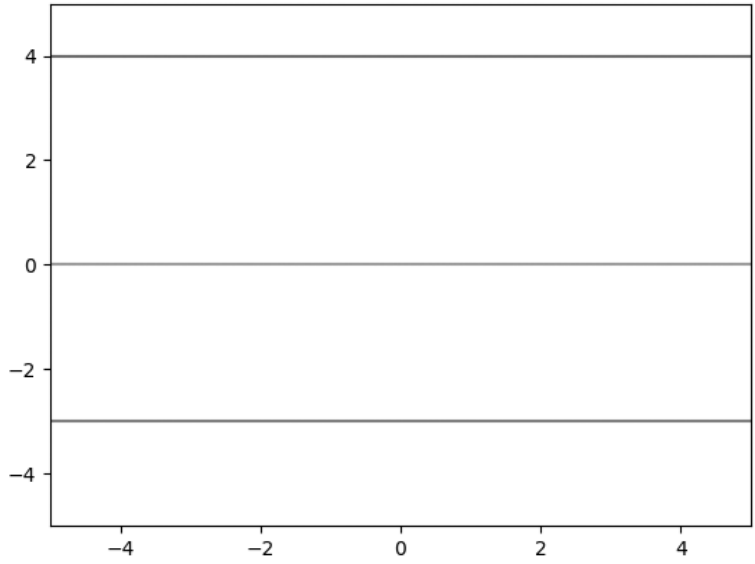

FIGURE 3.1 A graph of three horizontal line segments.

Figure 3.2 displays two slanted lines whose equations are y = x and y = −x, respectively.

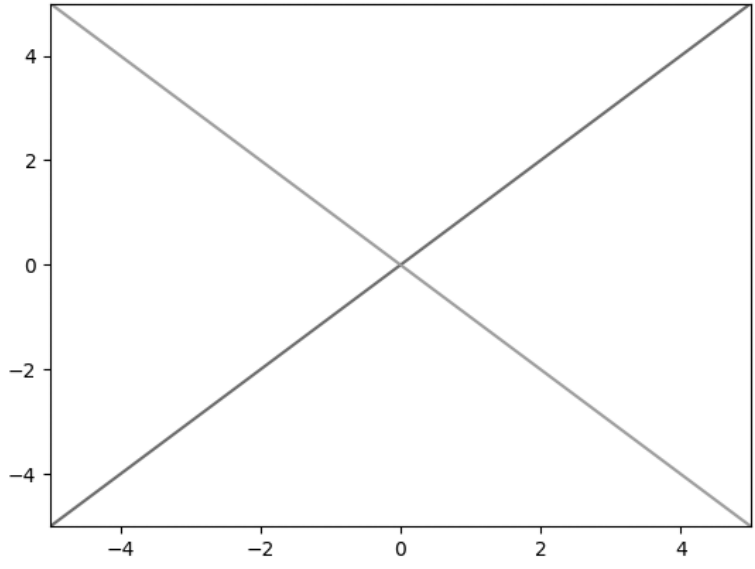

FIGURE 3.2 A graph of two diagonal line segments.

Figure 3.3 displays two slanted parallel lines whose equations are `y = 2*x` and `y = 2*x+3`, respectively.

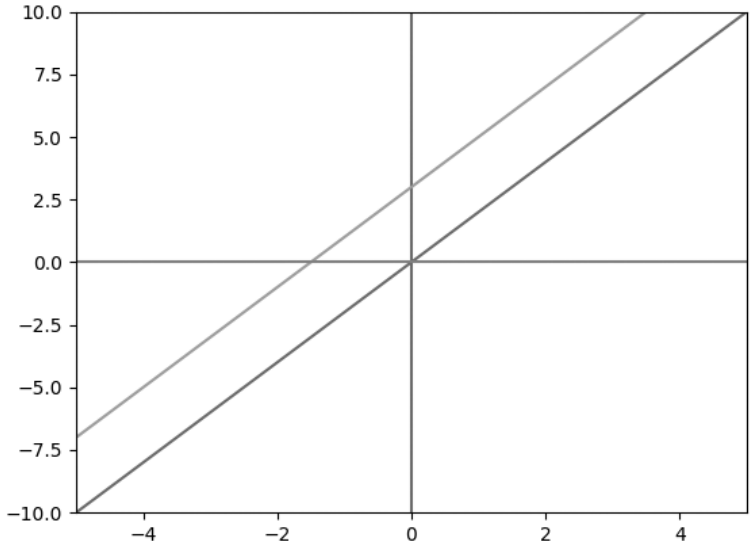

FIGURE 3.3 A graph of two slanted parallel line segments.

Figure 3.4 displays a piecewise linear graph consisting of connected line segments.

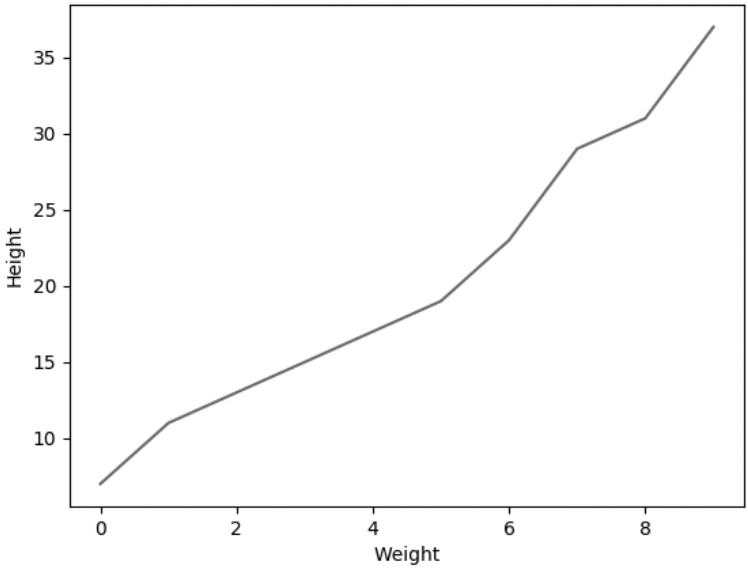

FIGURE 3.4 A piecewise linear graph of line segments.

Now that you have seen some basic examples of lines in the Euclidean plane, let's look at some code samples that use `NumPy` and `Matplotlib` to display scatterplots of points in the plane.

PLOTTING RANDOMIZED POINTS WITH `NUMPY` AND `MATPLOTLIB`

The previous section contains simple examples of line segments, but the code is deferred until Chapter 7. This section and the next section contain code samples with `Matplotlib` APIs that are not discussed; however, the code is straightforward, so you can infer its purpose. In addition, you can learn more about `Matplotlib` in Chapter 7 (which focuses on data visualization) or read a short online tutorial for more details.

Listing 3.20 displays the contents of `np_plot.py` that illustrates how to plot multiple points on a line in the plane.

LISTING 3.20: np_plot.py

```
import numpy as np
import matplotlib.pyplot as plt

x = np.random.randn(15,1)
y = 2.5*x + 5 + 0.2*np.random.randn(15,1)

plt.scatter(x,y)
plt.show()
```

Listing 3.20 starts with two `import` statements, followed by the initialization of `x` as a set of random values via the `NumPy randn()` API. Next, `y` is assigned a range of values that consist of two parts: a linear equation with input values from the `x` values, which is combined with a randomization factor. Figure 3.5 displays the output generated by the code in Listing 3.20.

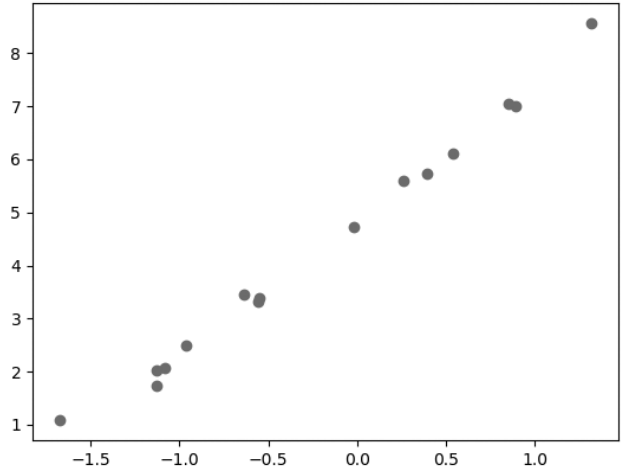

FIGURE 3.5 Datasets with potential linear regression.

PLOTTING A QUADRATIC WITH NUMPY AND MATPLOTLIB

Listing 3.21 displays the contents of np_plot_quadratic.py that illustrates how to plot a quadratic function in the plane.

LISTING 3.21: np_plot_quadratic.py

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-5,5,num=100)[:,None]
y = -0.5 + 2.2*x +0.3*x**3+ 2*np.random.randn(100,1)

plt.plot(x,y)
plt.show()
```

Listing 3.21 starts with two `import` statements, followed by the initialization of `x` as a range of values via the `NumPy linspace()` API. Next, `y` is assigned a range of values that fit a quadratic equation, which are based on the values for the variable `x`. Figure 3.6 displays the output generated by the code in Listing 3.21.

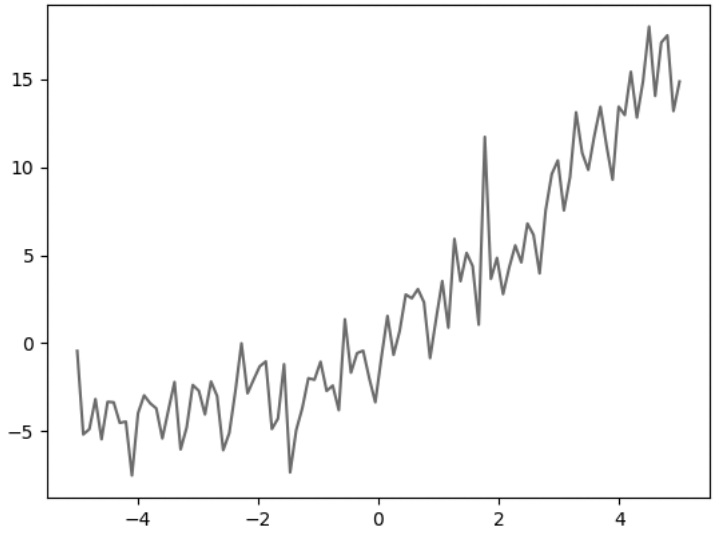

FIGURE 3.6 Datasets with potential linear regression.

Now that you have seen an assortment of line graphs and scatterplots, let's delve into linear regression, which is the topic of the next section.

WHAT IS LINEAR REGRESSION?

Linear regression was created in 1805 (more than two hundred years ago), and it's a very important algorithm in statistical analysis and in machine

learning. Any decent statistical package supports linear regression and invariably supports polynomial regression. Just to make sure: linear regression involves lines, which are polynomials with degree one, whereas polynomial regression involves fitting polynomials of degree greater than one to a dataset.

In general terms, linear regression finds the equation of the best-fitting hyperplane that approximates a dataset, where a hyperplane has degree one less than the dimensionality of the dataset. In particular, if the dataset is in the Euclidean plane, the hyperplane is simply a line; if the dataset is in 3D, the hyperplane is a "regular" plane.

Linear regression is suitable when the points in a dataset are distributed in such a way that they can reasonably be approximated by a hyperplane. If not, then you can try to fit other types of multivariable polynomial surfaces to the points in the dataset.

Keep in mind two other details. First, the best-fitting hyperplane does not necessarily intersect all (or even most of) the points in the dataset. In fact, the best-fitting hyperplane might not intersect *any* points in the dataset. The purpose of a best-fitting hyperplane is to *approximate* the points in the dataset as closely as possible. Second, linear regression is *not* the same as curve fitting, which attempts to find a polynomial that passes through a set of points.

Some details about curve fitting: given n points in the plane (no two of which have the same `x` value), there is a polynomial of degree less than or equal to n–1 that passes through those points. Thus, a line (which has degree one) will pass through any pair of nonvertical points in the plane. For any triple of points in the plane, there is a quadratic equation or a line that passes through those points.

In some cases, a lower degree polynomial is available. For instance, consider the set of 100 points in which the `x` value equals the `y` value: clearly the line `y = x` (a polynomial of degree one) passes through all of those points.

However, keep in mind that the extent to which a line "represents" a set of points in the plane depends on how closely those points can be approximated by a line.

What Is Multivariate Analysis?

Multivariate analysis generalizes the equation of a line in the Euclidean plane, and it has the following form:

```
y = w1*x1 + w2*x2 + . . . + wn*xn + b
```

As you can see, the preceding equation contains a linear combination of the variables `x1, x2, . . . , xn`. In this book, we will usually work with datasets that involve lines in the Euclidean plane.

What about Nonlinear Datasets?

Simple linear regression finds the best-fitting line that fits a dataset, but what happens if the dataset does not fit a line in the plane? This is an excellent

point! In such a scenario, we look for other curves to approximate the dataset, such a quadratic, cubic, or higher-degree polynomials. However, these alternatives involve trade-offs, as we'll discuss later.

Another possibility is to use a continuous piecewise linear function, which is a function that comprises a set of line segments, where adjacent line segments are connected. If one or more pairs of adjacent line segments are not connected, then it's a piecewise linear function (i.e., the function is discontinuous). In either case, line segments have degree one, which involves lower computational complexity than higher order polynomials.

Thus, given a set of points in the plane, try to find the "best-fitting" line that fits those points, after addressing the following questions:

1. How do we know that a line "fits" the data?
2. What if a different type of curve is a better fit?
3. What does "best fit" mean?

One way to check if a line fits the data well is through a simple visual check: display the data in a graph, and if the data conforms to the shape of a line reasonably well, then a line might be a good fit. However, this is a subjective decision, and a sample dataset that does not fit a line is displayed in Figure 3.7.

Figure 3.7 displays a dataset containing four points that do not fit a line.

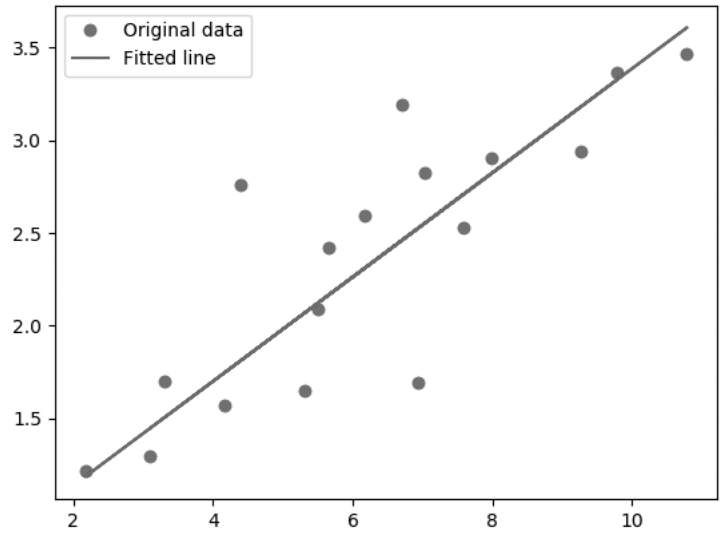

FIGURE 3.7 A nonlinear dataset.

On the other hand, if a line does not appear to be a good fit for the data, then perhaps a quadratic or cubic (or even higher degree) polynomial has the potential of being a better fit. Let's defer the nonlinear scenario and let's make the assumption that a line would be a good fit for the data. There is a

well-known technique for finding the "best-fitting" line for such a dataset, and it's called MSE.

THE MSE FORMULA

Figure 3.8 displays the formula for the MSE. Translated into English: the MSE is the sum of the squares of the difference between an *actual* `y` value and the *predicted* `y` value, divided by the number of points. Note that the predicted `y` value is the `y` value that each data point would have if that data point were actually on the best-fitting line.

In general, the goal is to minimize the error, which determines the best-fitting line in the case of linear regression. However, you might be satisfied with a "good enough" value when the time and/or loss for any additional reduction in the error is deemed prohibitive, which means that this decision is not a purely programmatic decision.

Figure 3.8 displays the formula for MSE for calculating the best-fitting line for a set of points in the plane.

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

FIGURE 3.8 The MSE formula.

Other Error Types

Although we will only discuss MSE for linear regression in this book, there are other types of formulas for errors that you can use for linear regression, some of which are listed here:

- MSE
- RMSE
- RMSPROP
- MAE

The MSE is the basis for the preceding error types. For example, RMSE is "Root Mean Squared Error", which is the square root of MSE.

On the other hand, MAE is "Mean Absolute Error", which is *the sum of the absolute value of the differences of the y terms* (not the square of the differences of the y terms).

The RMSProp optimizer utilizes the magnitude of recent gradients to normalize the gradients. Maintain a moving average over the RMS ("root mean squared", which is the square root of the MSE) gradients, and then divide that term by the current gradient.

Although it's easier to compute the derivative of MSE (because it's a differentiable function), it's also true that MSE is more susceptible to outliers, more

so than MAE. The reason is simple: a squared term can be significantly larger than adding the absolute value of a term. For example, if a difference term is 10, then the squared term 100 is added to MSE, whereas only 10 is added to MAE. Similarly, if a difference term is –20, then the squared term 400 is added to MSE, whereas only 20 (which is the absolute value of –20) is added to MAE.

Nonlinear Least Squares

When predicting housing prices, where the dataset contains a wide range of values, techniques such as linear regression or random forests can cause the model to "overfit" the samples with the highest values in order to reduce quantities such as MAE.

In this scenario, you probably want an error metric, such as relative error, that reduces the importance of fitting the samples with the largest values. This technique is called nonlinear least squares, which may use a log-based transformation of labels and predicted values.

CALCULATING MSE MANUALLY

Let's look at two simple graphs, each of which contains a line that approximates a set of points in a scatterplot. Notice that the line segment is the same for both sets of points, but the datasets are slightly different. We will manually calculate the MSE for both datasets and determine which value of MSE is smaller.

Figure 3.9 displays a set of points and a line that is a potential candidate for best-fitting line for the data.

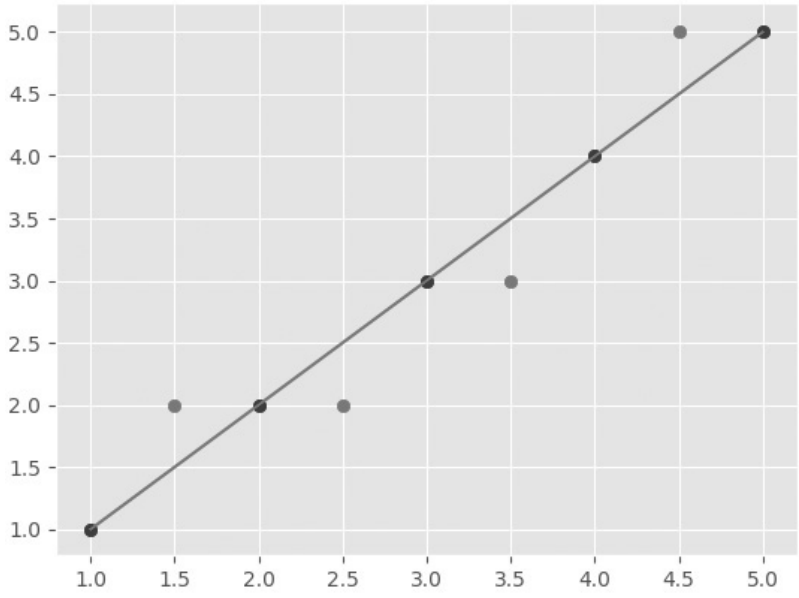

FIGURE 3.9 A line graph that approximates points of a scatterplot.

The MSE for the line in Figure 3.9 is computed as follows:

```
MSE = 1*1 + (-1)*(-1) + (-1)*(-1) + 1*1 = 4
```

Now look at Figure 3.10, which also displays a set of points and a line that is a potential candidate for best-fitting line for the data.

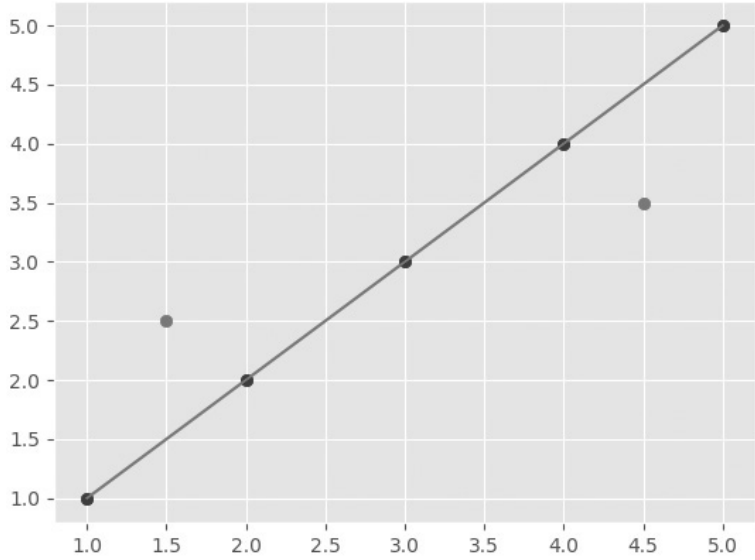

FIGURE 3.10 A line graph that approximates points of a scatterplot.

The MSE for the line in Figure 3.10 is computed as follows:

```
MSE = (-2)*(-2) + 2*2 = 8
```

Thus, the line in Figure 3.10 has a smaller MSE than the line in Figure 3.9, which might have surprised you (or did you guess correctly?).

In these two figures we calculated the MSE easily and quickly, but in general it's significantly more difficult. For instance, if we plot ten points in the Euclidean plane that do not closely fit a line, with individual terms that involve non-integer values, we will probably need a calculator. A better solution involves `NumPy` functions, as discussed in the next section.

FIND THE BEST-FITTING LINE IN `NUMPY`

Earlier in this chapter you saw examples of lines in the plane, including horizontal, slanted, and parallel lines. Most of those lines have a positive slope and a nonzero value for their y-intercept. Although there are scatterplots of data points in the plane where the best-fitting line has a negative slope, the examples in this book involve scatterplots whose best-fitting line has a positive slope.

Listing 3.22 displays the contents of find_best_fit.py that illustrates how to determine the best-fitting line for a set of points in the Euclidean plane. The solution is based on so-called "closed form" formulas that are available from statistics.

LISTING 3.22: find_best_fit.py

```
import numpy as np

xs = np.array([1,2,3,4,5], dtype=np.float64)
ys = np.array([1,2,3,4,5], dtype=np.float64)

def best_fit_slope(xs,ys):
  m = (((np.mean(xs)*np.mean(ys))-np.mean(xs*ys)) /
       ((np.mean(xs)**2) - np.mean(xs**2)))
  b = np.mean(ys) - m * np.mean(xs)

  return m, b

m,b = best_fit_slope(xs,ys)
print('m:',m,'b:',b)
```

Listing 3.22 starts with two NumPy arrays xs and ys that are initialized with the first five positive integers. The Python function best_fit_slope() calculates the optimal values of m (the slope) and b (the y-intercept) of a set of numbers. The output from Listing 3.22 is here:

```
m: 1.0 b: 0.0
```

Notice that the NumPy arrays xs and ys are identical, which means that these points lie on the identity function whose slope is 1. By simple extrapolation, the point (0,0) is also a point on the same line. Hence, the y-intercept of this line must equal 0.

If you are interested, you can search online to find the derivation for the values of m and b. In this chapter, we're going to skip the derivation and proceed with examples of calculating the MSE. The first example involves calculating the MSE manually, followed by an example that uses NumPy formulas to perform the calculations.

CALCULATING MSE BY SUCCESSIVE APPROXIMATION (1)

This section contains a code sample that uses a simple technique for successively determining better approximations for the slope and y-intercept of a best-fitting line. Recall that an approximation of a derivative is the ratio of "delta y" divided by "delta x". The "delta" values calculate the difference of the y values and the difference of the x values, respectively, of two nearby points (x1,y1) and (x2,y2) on a function. Hence, the delta-based approximation ratio is (y2-y1)/(x2-x1).

The technique in this section involves a simplified approximation for the "delta" values: we assume that the denominators are equal to 1. As a result, we need only calculate the numerators of the "delta" values: in this code sample, those numerators are the variables `dw` and `db`.

Listing 3.23 displays the contents of `plain_linreg1.py` that illustrates how to compute the MSE with simulated data.

LISTING 3.23: plain_linreg1.py

```
import numpy as np
import matplotlib.pyplot as plt

X = [0,0.12,0.25,0.27,0.38,0.42,0.44,0.55,0.92,1.0]
Y = [0,0.15,0.54,0.51,0.34,0.1, 0.19,0.53,1.0,0.58]

losses = []
#Step 1: Parameter initialization
W = 0.45 # the initial slope
b = 0.75 # the initial y-intercept

for i in range(1, 100):
  #Step 2: Calculate loss
  Y_pred = np.multiply(W, X) + b
  loss_error = 0.5 * (Y_pred - Y)**2
  loss = np.sum(loss_error)/10

  #Step 3: Calculate dw and db
  db = np.sum((Y_pred - Y))
  dw = np.dot((Y_pred - Y), X)
  losses.append(loss)

  #Step 4: Update parameters:
  W = W - 0.01*dw
  b = b - 0.01*db

  if i%10 == 0:
    print("Loss at", i,"iteration = ", loss)

#Step 5: Repeat via a for loop with 1000 iterations

#Plot loss versus # of iterations
print("W = ", W,"& b = ",  b)
plt.plot(losses)
plt.ylabel('loss')
plt.xlabel('iterations (per tens)')
plt.show()
```

Listing 3.23 defines the variables `X` and `Y` that are simple arrays of numbers (this is our dataset). Next, the `losses` array is initialized as an empty array, and we will append successive loss approximations to this array. The variables `W` and `b` correspond to the slope and y-intercept, and they are initialized with the values 0.45 and 0.75, respectively (feel free to experiment with these values).

The next portion of Listing 3.23 is a for loop that executes 100 times. During each iteration, the variables `Y_pred`, `loss_error`, and `loss` are computed, and they correspond to the predicted value, the error, and the loss, respectively (remember, we are performing linear regression). The value of `loss` (which is the error for the current iteration) is then appended to the `losses` array.

Next, the variables `dw` and `db` are calculated: these correspond to "delta w" and "delta b" that we'll use to update the values of `W` and `b`, respectively. The code is reproduced here:

```
#Step 4: Update parameters:
W = W - 0.01*dw
b = b - 0.01*db
```

Notice that `dw` and `db` are both multiplied by the value 0.01, which is the value of our "learning rate" (experiment with this value as well).

The next code snippet displays the current loss, which is performed every tenth iteration through the loop. When the loop finishes execution, the values of `W` and `b` are displayed, and a plot is displayed that shows the loss values on the vertical axis and the loop iterations on the horizontal axis. The output from Listing 3.23 is here:

```
Loss at 10 iteration =  0.04114630674619491
Loss at 20 iteration =  0.026706242729839395
Loss at 30 iteration =  0.024738889446900423
Loss at 40 iteration =  0.023850565034634254
Loss at 50 iteration =  0.0231499048706651
Loss at 60 iteration =  0.02255361434242207
Loss at 70 iteration =  0.0220425055291673
Loss at 80 iteration =  0.021604128492245713
Loss at 90 iteration =  0.021228111750568435
W =  0.47256473531193927 & b =  0.19578262688662174
```

Figure 3.11 displays the plot of loss-versus-iterations for Listing 3.23.

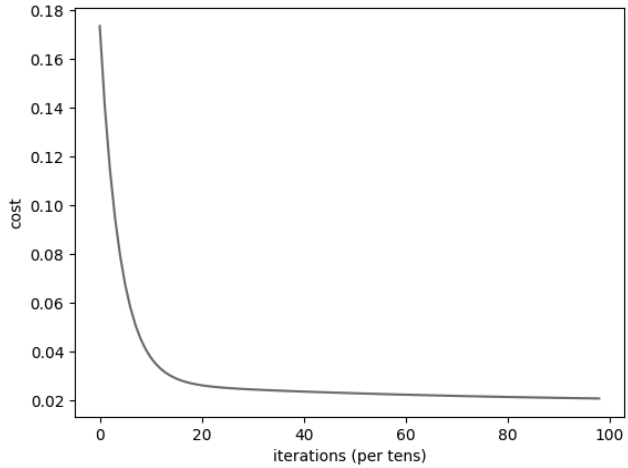

FIGURE 3.11 A plot of loss-versus-iterations.

CALCULATING MSE BY SUCCESSIVE APPROXIMATION (2)

In the previous section you saw how to calculate "delta" approximations in order to determine the equation of a best-fitting line for a set of points in a 2D plane. The example in this section generalizes the code in the previous section by adding an outer loop that represents the number of epochs. In case you don't already know, the number of epochs specifies the number of times that an inner loop is executed.

Listing 3.24 displays the contents of `plain_linreg2.py` that illustrates how to compute the MSE with simulated data.

LISTING 3.24: plain_linreg2.py

```
import numpy as np
import matplotlib.pyplot as plt

# %matplotlib inline
X = [0,0.12,0.25,0.27,0.38,0.42,0.44,0.55,0.92,1.0]
Y = [0,0.15,0.54,0.51, 0.34,0.1,0.19,0.53,1.0,0.58]

#uncomment to see a plot of X versus Y values
#plt.plot(X,Y)
#plt.show()

losses = []
#Step 1: Parameter initialization
W = 0.45
b = 0.75

epochs = 100
lr = 0.001

for j in range(1, epochs):
  for i in range(1, 100):
    #Step 2: Calculate loss
    Y_pred = np.multiply(W, X) + b
    Loss_error = 0.5 * (Y_pred - Y)**2
    loss = np.sum(Loss_error)/10

    #Step 3: Calculate dW and db
    db = np.sum((Y_pred - Y))
    dw = np.dot((Y_pred - Y), X)
    losses.append(loss)

    #Step 4: Update parameters:
    W = W - lr*dw
    b = b - lr*db

    if i%50 == 0:
      print("Loss at epoch", j,"= ", loss)

#Plot loss versus # of iterations
print("W = ", W,"& b = ",  b)
```

```
plt.plot(losses)
plt.ylabel('loss')
plt.xlabel('iterations (per tens)')
plt.show()
```

Compare the new contents of Listing 3.24 (shown in bold) with the contents of Listing 3.23: the changes are minimal, and the main difference is to execute the inner loop 100 times for each iteration of the outer loop, which also executes 100 times. The output from Listing 3.24 is here:

```
('Loss at epoch', 1, '= ', 0.07161762489862147)
('Loss at epoch', 2, '= ', 0.030073922512586938)
('Loss at epoch', 3, '= ', 0.025415528992988472)
('Loss at epoch', 4, '= ', 0.024227826373677794)
('Loss at epoch', 5, '= ', 0.02346241967071181)
('Loss at epoch', 6, '= ', 0.022827707922883803)
('Loss at epoch', 7, '= ', 0.022284262669854064)
('Loss at epoch', 8, '= ', 0.02181735173716673)
('Loss at epoch', 9, '= ', 0.021416050179776294)
('Loss at epoch', 10, '= ', 0.02107112540934384)
// details omitted for brevity
('Loss at epoch', 90, '= ', 0.018960749188638278)
('Loss at epoch', 91, '= ', 0.01896074755776306)
('Loss at epoch', 92, '= ', 0.018960746155994725)
('Loss at epoch', 93, '= ', 0.018960744951148113)
('Loss at epoch', 94, '= ', 0.018960743915559485)
('Loss at epoch', 95, '= ', 0.018960743025451313)
('Loss at epoch', 96, '= ', 0.018960742260386375)
('Loss at epoch', 97, '= ', 0.018960741602798474)
('Loss at epoch', 98, '= ', 0.018960741037589136)
('Loss at epoch', 99, '= ', 0.018960740551780944)
('W = ', 0.6764145874436108, '& b = ', 0.09976839618922698)
```

Figure 3.12 displays the plot of loss-versus-iterations for Listing 3.24.

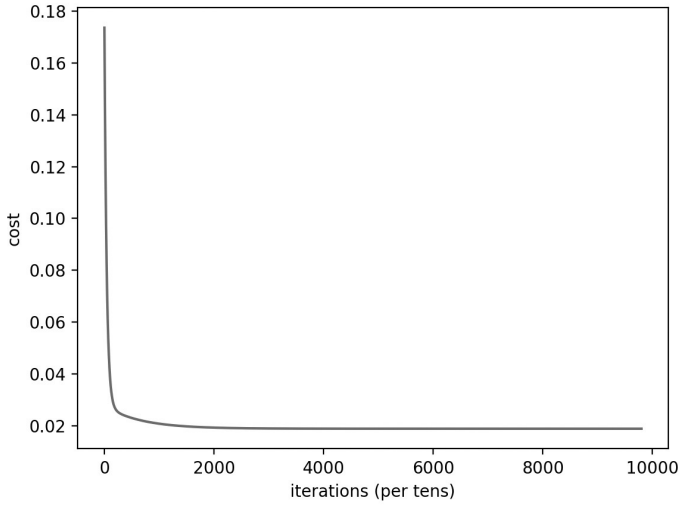

FIGURE 3.12 A plot of loss-versus-iterations.

Notice that Figure 3.12 has 10,000 iterations on the horizontal axis, whereas Figure 3.11 has only 100 iterations on the horizontal axis.

GOOGLE COLABORATORY

Depending on the hardware, GPU-based TF 2 code is typically at least fifteen times faster than CPU-based TF 2 code. However, the loss of a good GPU can be a significant factor. Although NVIDIA provides GPUs, those consumer-based GPUs are not optimized for multi-GPU support (which *is* supported by TF 2).

Fortunately, Google Colaboratory is an affordable alternative that provides free GPU support, and also runs as a `Jupyter` notebook environment. In addition, Google Colaboratory executes your code in the cloud and involves zero configuration, and it's available here:

https://colab.research.google.com/notebooks/welcome.ipynb.

This `Jupyter` notebook is suitable for training simple models and testing ideas quickly. Google Colaboratory makes it easy to upload local files, install software in `Jupyter` notebooks, and even connect Google Colaboratory to a `Jupyter` runtime on your local machine.

Some of the supported features of Colaboratory include TF 2 execution with GPUs, visualization using `Matplotlib`, and the ability to save a copy of your Google Colaboratory notebook to Github by using `File > Save a copy to GitHub`.

Moreover, you can load any `.ipynb` on GitHub by just adding the path to the URL `colab.research.google.com/github/` (see the Colaboratory website for details).

Google Colaboratory has support for other technologies such as HTML and SVG, enabling you to render SVG-based graphics in notebooks that are in Google Colaboratory. One point to keep in mind: any software that you install in a Google Colaboratory notebook is only available on a per-session basis: if you log out and log in again, you need to perform the same installation steps that you performed during your earlier Google Colaboratory session.

As mentioned earlier, there is one other *very* nice feature of Google Colaboratory: you can execute code on a GPU for up to twelve hours per day for free. This free GPU support is extremely useful for people who don't have a suitable GPU on their local machine (which is probably the majority of users), and now they launch TF 2 code to train neural networks in less than twenty or thirty minutes that would otherwise require multiple hours of CPU-based execution time.

In case you're interested, you can launch Tensorboard inside a Google Colaboratory notebook with the following command (replace the specified directory with your own location):

```
%tensorboard --logdir /logs/images
```

Keep in mind the following details about Google Colaboratory. First, whenever you connect to a server in Google Colaboratory, you start what's known as a *session*. You can execute the code in a session with a CPU (the default), a GPU, or a TPU (which is available for free), and you can execute your code without any time limit for your session. However, if you select the GPU option for your session, *only the first 12 hours of GPU execution time are free*. Any additional GPU time during that same session incurs a small charge (see the website for those details).

The other point to keep in mind is that any software that you install in a Jupyter notebook during a given session will *not* be saved when you exit that session. For example, the following code snippet installs `TFLearn` in a Jupyter notebook:

```
!pip install tflearn
```

When you exit the current session and at some point later you start a new session, you need to install `TFLearn` again, as well as any other software (such as Github repositories) that you also installed in any previous session.

Incidentally, you can also run TF 2 code and TensorBoard in Google Colaboratory. Navigate to this link for more information:

https://www.tensorflow.org/tensorboard/r2/tensorboard_in_notebooks.

Uploading CSV Files in Google Colaboratory

Listing 3.25 displays the contents `upload_csv_file.ipynb` that illustrates how to upload a CSV file in a Google Colaboratory notebook.

LISTING 3.25: upload_csv_file.ipynb

```
import pandas as pd

from google.colab import files
uploaded = files.upload()

df = pd.read_csv("weather_data.csv")
print("dataframe df:")
df
```

Listing 3.25 uploads the CSV file `weather_data.csv` whose contents are not shown because they are not important for this example. The code shown in bold is the Colaboratory-specific code that is required to upload the CSV file. When you launch this code, you will see a small button labeled "Browse", which you must click and then select the CSV file that is listed in the code snippet. After doing so, the rest of the code is executed, and you will see the contents of the CSV file displayed in your browser session.

NOTE *You must supply the CSV file* `weather_data.csv` *if you want to launch this Jupyter notebook successfully in Google Colaboratory.*

SUMMARY

This chapter introduced you to the `NumPy` library for Python. You learned how to write Python scripts containing loops, arrays, and lists. You also saw how to work with dot products, the `reshape()` method, plotting with `Matplotlib` (discussed in more detail in Chapter 7), and examples of linear regression.

Then you learned how to work with subranges of arrays, and also negative subranges of vectors and arrays, both of which are very useful for extracting portions of datasets in machine learning tasks. You also saw various other `NumPy` operations, such as the `reshape()` method that is extremely useful (and very common) when working with images files.

Next, you learned how to use `NumPy` for linear regression, the MSE, and how to calculate MSE with the `NumPy linspace()` method. Finally, you got an introduction to Google Colaboratory, where you can take advantage of the free GPU time when you launch Jupyter notebooks.

CHAPTER 4

INTRODUCTION TO PANDAS

This chapter introduces the Pandas library and contains various code samples that illustrate some useful Pandas features. As you will see, the title of each section clearly indicates its contents, so you can easily scan this chapter for those sections that contain material that is new to you. This approach will help you make efficient use of your time when you read the contents of this chapter.

The first part of this chapter contains a brief introduction to Pandas, followed by code samples that illustrate how to define Pandas DataFrames and also display their attributes. Please keep in mind that this chapter is devoted to Pandas DataFrames. There is one code block that illustrates how to define a Pandas Series, and if you want to learn more about this Pandas Series, you can search online for more information.

The second part of this chapter discusses various types of dataframes that you can create, such as numeric and Boolean dataframes. In addition, you will see examples of creating dataframes with NumPy functions and random numbers. You will also see examples of converting between Python dictionaries and JSON-based data, and also how to create a Pandas DataFrame from JSON-based data.

WHAT IS PANDAS?

Pandas is a Python package that is compatible with other Python packages, such as NumPy, Matplotlib, and so forth. Install Pandas by opening a command shell and invoking this command for Python 3.x:

```
pip3 install pandas
```

In many ways the Pandas package has the semantics of a spreadsheet, and it also works with various file types, such as xsl, xml, html, and csv files.

Pandas provides a data type called a dataframe (similar to a Python dictionary) with extremely powerful functionality (similar to the functionality of a spreadsheet).

Pandas DataFrames

In simplified terms, a Pandas DataFrame is a two-dimensional data structure, and it's convenient to think of the data structure in terms of rows and columns. Dataframes can be labeled (rows as well as columns), and the columns can contain different data types. The source of the dataset can be a data file, database tables, a web service, and so forth. Pandas DataFrame features include:

- Dataframe Methods
- Dataframe Statistics
- Grouping, Pivoting, and Reshaping
- Handling Missing Data
- Joining Dataframes

Dataframes and Data Cleaning Tasks

The specific tasks that you need to perform depend on the structure and contents of a dataset. In general, you will perform a workflow with the following steps (not necessarily always in this order), all of which can be performed with a Pandas DataFrame:

- Read data into a dataframe
- Display top of dataframe
- Display column data types
- Display non-missing values
- Replace NA with a value
- Iterate through the columns
- Statistics for each column
- Find Missing Values
- Total missing values
- Percentage of missing values
- Sort table values
- Print summary information
- Columns with > 50% missing values
- Rename columns

A PANDAS DATAFRAME EXAMPLE

Listing 4.1 displays the contents of pandas_df.py that illustrates how to define several Pandas DataFrames and display their contents.

LISTING 4.1: pandas_df.py

```
import pandas as pd
import numpy as np

myvector1 = np.array([1,2,3,4,5])
print("myvector1:")
print(myvector1)
print()

mydf1 = pd.DataFrame(myvector1)
print("mydf1:")
print(mydf1)
print()

myvector2 = np.array([i for i in range(1,6)])
print("myvector2:")
print(myvector2)
print()

mydf2 = pd.DataFrame(myvector2)
print("mydf2:")
print(mydf2)
print()

myarray = np.array([[10,30,20], [50,40,60],[1000,2000,3000]])
print("myarray:")
print(myarray)
print()

mydf3 = pd.DataFrame(myarray)
print("mydf3:")
print(mydf3)
print()
```

Listing 4.1 starts with a standard `import` statement for `Pandas` and `NumPy`, followed by the definition of two one-dimensional `NumPy` arrays and a two-dimensional `NumPy` array. The `NumPy` syntax ought to be familiar to you (many basic tutorials are available online). Each `NumPy` variable is followed by a corresponding `Pandas` dataframe `mydf1`, `mydf2`, and `mydf3`. Now launch the code in Listing 4.1 and you will see the following output, and you can compare the `NumPy` arrays with the `Pandas DataFrames`:

```
myvector1:
[1 2 3 4 5]

mydf1:
   0
0  1
1  2
2  3
3  4
4  5
```

```
myvector2:
[1 2 3 4 5]

mydf2:
   0
0  1
1  2
2  3
3  4
4  5

myarray:
[[  10   30   20]
 [  50   40   60]
 [1000 2000 3000]]

mydf3:
      0     1     2
0    10    30    20
1    50    40    60
2  1000  2000  3000
```

By contrast, the following code block illustrates how to define a `Pandas Series`:

```
names = pd.Series(['SF', 'San Jose', 'Sacramento'])
sizes = pd.Series([852469, 1015785, 485199])
df = pd.DataFrame({ 'Cities': names, 'Size': sizes })
print(df)
```

Create a Python file with the preceding code (along with the required `import` statement), and when you launch that code you will see the following output:

```
    City name    sizes
0          SF   852469
1    San Jose  1015785
2  Sacramento   485199
```

DESCRIBING A PANDAS DATAFRAME

Listing 4.2 displays the contents of `pandas_df_describe.py` that illustrates how to define a `Pandas DataFrame` that contains a 3x3 `NumPy` array of integer values, where the rows and columns of the dataframe are labeled. Various other aspects of the dataframe are also displayed.

LISTING 4.2: pandas_df_describe.py

```
import numpy as np
import pandas as pd
```

```
myarray = np.array([[10,30,20], [50,40,60],[1000,2000,3000]])

rownames = ['apples', 'oranges', 'beer']
colnames = ['January', 'February', 'March']

mydf = pd.DataFrame(myarray, index=rownames, columns=colnames)
print("contents of df:")
print(mydf)
print()

print("contents of January:")
print(mydf['January'])
print()

print("Number of Rows:")
print(mydf.shape[0])
print()

print("Number of Columns:")
print(mydf.shape[1])
print()

print("Number of Rows and Columns:")
print(mydf.shape)
print()

print("Column Names:")
print(mydf.columns)
print()

print("Column types:")
print(mydf.dtypes)
print()

print("Description:")
print(mydf.describe())
print()
```

Listing 4.2 starts with two standard `import` statements followed by the variable `myarray`, which is a 3x3 `NumPy` array of numbers. The variables `rownames` and `colnames` provide names for the rows and columns, respectively, of the `Pandas DataFrame` `mydf`, which is initialized as a `Pandas DataFrame` with the specified datasource (i.e., `myarray`).

The first portion of the following output requires a single `print` statement (which simply displays the contents of `mydf`). The second portion of the output is generated by invoking the `describe()` method that is available for any `NumPy DataFrame`. The `describe()` method is very useful; you will see various statistical quantities such as the mean, standard deviation minimum, and maximum performed column-wise (not row-wise), along with values for the 25th, 50th, and 75th percentiles. The output of Listing 4.2 is here:

```
contents of df:
         January  February  March
apples        10        30     20
oranges       50        40     60
beer        1000      2000   3000

contents of January:
apples       10
oranges      50
beer       1000
Name: January, dtype: int64

Number of Rows:
3

Number of Columns:
3

Number of Rows and Columns:
(3, 3)

Column Names:
Index(['January', 'February', 'March'], dtype='object')

Column types:
January     int64
February    int64
March       int64
dtype: object

Description:
           January     February        March
count     3.000000     3.000000     3.000000
mean    353.333333   690.000000  1026.666667
std     560.386771  1134.504297  1709.073823
min      10.000000    30.000000    20.000000
25%      30.000000    35.000000    40.000000
50%      50.000000    40.000000    60.000000
75%     525.000000  1020.000000  1530.000000
max    1000.000000  2000.000000  3000.000000
```

PANDAS BOOLEAN DATAFRAMES

Pandas supports Boolean operations on dataframes, such as the logical or, the logical and, and the logical negation of a pair of dataframes. Listing 4.3 displays the contents of pandas_boolean_df.py that illustrates how to define a Pandas DataFrame whose rows and columns are Boolean values.

LISTING 4.3: pandas_boolean_df.py

```
import pandas as pd

df1 = pd.DataFrame({'a': [1, 0, 1], 'b': [0, 1, 1] },
dtype=bool)
```

```
df2 = pd.DataFrame({'a': [0, 1, 1], 'b': [1, 1, 0] },
dtype=bool)

print("df1 & df2:")
print(df1 & df2)

print("df1 | df2:")
print(df1 | df2)

print("df1 ^ df2:")
print(df1 ^ df2)
```

Listing 4.3 initializes the `DataFrames df1` and `df2`, and then computes `df1 & df2`, df1 `| df2`, `df1 ^ df2`, which represent the logical `AND`, the logical `OR`, and the logical negation, respectively, of `df1` and `df2`. The output from launching the code in Listing 4.3 is here:

```
df1 & df2:
       a      b
0  False  False
1  False   True
2   True  False
df1 | df2:
      a     b
0  True  True
1  True  True
2  True  True
df1 ^ df2:
       a      b
0   True   True
1   True  False
2  False   True
```

Transposing a Pandas DataFrame

The `T` attribute (as well as the transpose function) enables you to generate the transpose of a `Pandas DataFrame`, similar to a `NumPy ndarray`.

For example, the following code snippet defines a `Pandas DataFrame df1` and then displays the transpose of `df1`:

```
df1 = pd.DataFrame({'a': [1, 0, 1], 'b': [0, 1, 1] },
dtype=int)

print("df1.T:")
print(df1.T)
```

The output is here:

```
df1.T:
   0  1  2
a  1  0  1
b  0  1  1
```

The following code snippet defines the Pandas DataFrames df1 and df2 and then displays their sum:

```
df1 = pd.DataFrame({'a' : [1, 0, 1], 'b' : [0, 1, 1] },
dtype=int)
df2 = pd.DataFrame({'a' : [3, 3, 3], 'b' : [5, 5, 5] },
dtype=int)

print("df1 + df2:")
print(df1 + df2)
```

The output is here:

```
df1 + df2:
   a  b
0  4  5
1  3  6
2  4  6
```

PANDAS DATAFRAMES AND RANDOM NUMBERS

Listing 4.4 displays the contents of pandas_random_df.py that illustrates how to create a Pandas DataFrame with random numbers.

LISTING 4.4: pandas_random_df.py

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randint(1, 5, size=(5, 2)),
columns=['a','b'])
df = df.append(df.agg(['sum', 'mean']))

print("Contents of dataframe:")
print(df)
```

Listing 4.4 defines the Pandas DataFrame df that consists of 5 rows and 2 columns of random integers between 1 and 5. Notice that the columns of df are labeled "a" and "b". In addition, the next code snippet appends two rows consisting of the sum and the mean of the numbers in both columns. The output of Listing 4.4 is here:

```
a     b
0      1.0  2.0
1      1.0  1.0
2      4.0  3.0
3      3.0  1.0
4      1.0  2.0
sum   10.0  9.0
mean   2.0  1.8
```

Listing 4.5 displays the contents of pandas_combine_df.py that illustrates how to define a Pandas DataFrame that is based on two NumPy arrays of numbers.

LISTING 4.5: pandas_combine_df.py

```
import pandas as pd
import numpy as np

df = pd.DataFrame({'foo1' : np.random.randn(5),
                   'foo2' : np.random.randn(5)})

print("contents of df:")
print(df)

print("contents of foo1:")
print(df.foo1)

print("contents of foo2:")
print(df.foo2)
```

Listing 4.5 defines the Pandas DataFrame df that consists of 5 rows and 2 columns (labeled "foo1" and "foo2") of random real numbers between 0 and 5. The next portion of Listing 4.5 displays the contents of df and foo1. The output of Listing 4.5 is here:

```
contents of df:
       foo1      foo2
0  0.274680 _0.848669
1 _0.399771 _0.814679
2  0.454443 _0.363392
3  0.473753  0.550849
4 _0.211783 _0.015014
contents of foo1:
0    0.256773
1    1.204322
2    1.040515
3   _0.518414
4    0.634141
Name: foo1, dtype: float64
contents of foo2:
0   _2.506550
1   _0.896516
2   _0.222923
3    0.934574
4    0.527033
Name: foo2, dtype: float64
```

CONVERTING CATEGORICAL DATA TO NUMERIC DATA

One common task in machine learning involves converting a feature containing character data into a feature that contains numeric data.

Listing 4.6 displays the contents of `sometext.tsv` that contains labeled data (spam or ham), which is used in the code sample displayed in Listing 4.7.

LISTING 4.6: sometext.tsv

```
type    text
ham     Available only for today
ham     I'm joking with you
spam    Free entry in 2 a wkly comp
ham     U dun say so early hor
ham     I don't think he goes to usf
spam    FreeMsg Hey there
ham     my brother is not sick
ham     As per your request Melle
spam    WINNER!! As a valued customer
```

Listing 4.7 displays the contents of `cat2numeric.py` that illustrates how to replace a text field with a corresponding numeric field.

LISTING 4.7: cat2numeric.py

```
import pandas as pd
import numpy as np

df = pd.read_csv('sometext.tsv', delimiter='\t')

print("=> First five rows (before):")
print(df.head(5))
print("-------------------------")

# map ham/spam to 0/1 values:
df['type'] = df['type'].map( {'ham':0 , 'spam':1} )

print("=> First five rows (after):")
print(df.head(5))
print("-------------------------")
```

Listing 4.7 initializes the `DataFrame df` with the contents of the CSV file `sometext.tsv`, and then displays the contents of the first five rows by invoking `df.head(5)`, which is also the default number of rows to display. The next code snippet in Listing 4.7 invokes the `map()` method to replace occurrences of `ham` with 0 and replace occurrences of `spam` with 1 in the column labeled type, as shown here:

```
df['type'] = df['type'].map( {'ham':0 , 'spam':1} )
```

The last portion of Listing 4.7 invokes the `head()` method again to display the first five rows of the dataset after having renamed the contents of the column type. Launch the code in Listing 4.7 and you will see the following output:

```
=> First five rows (before):
   type                          text
0   ham     Available only for today
```

```
1   ham           I'm joking with you
2  spam  Free entry in 2 a wkly comp
3   ham        U dun say so early hor
4   ham  I don't think he goes to usf
------------------------
=> First five rows (after):
   type                          text
0     0     Available only for today
1     0           I'm joking with you
2     1  Free entry in 2 a wkly comp
3     0        U dun say so early hor
4     0  I don't think he goes to usf
------------------------
```

As another example, Listing 4.8 displays the contents of `shirts.csv` and Listing 4.9 displays the contents of `shirts.py` that illustrate four techniques for converting categorical data to numeric data.

LISTING 4.8: shirts.csv

```
type,ssize
shirt,xxlarge
shirt,xxlarge
shirt,xlarge
shirt,xlarge
shirt,xlarge
shirt,large
shirt,medium
shirt,small
shirt,small
shirt,xsmall
shirt,xsmall
shirt,xsmall
```

LISTING 4.9: shirts.py

```
import pandas as pd

shirts = pd.read_csv("shirts.csv")
print("shirts before:")
print(shirts)
print()

# TECHNIQUE #1:
#shirts.loc[shirts['ssize']=='xxlarge','size'] = 4
#shirts.loc[shirts['ssize']=='xlarge', 'size'] = 4
#shirts.loc[shirts['ssize']=='large',  'size'] = 3
#shirts.loc[shirts['ssize']=='medium', 'size'] = 2
#shirts.loc[shirts['ssize']=='small',  'size'] = 1
#shirts.loc[shirts['ssize']=='xsmall', 'size'] = 1

# TECHNIQUE #2:
#shirts['ssize'].replace('xxlarge', 4, inplace=True)
#shirts['ssize'].replace('xlarge',  4, inplace=True)
```

```
#shirts['ssize'].replace('large',   3, inplace=True)
#shirts['ssize'].replace('medium',  2, inplace=True)
#shirts['ssize'].replace('small',   1, inplace=True)
#shirts['ssize'].replace('xsmall',  1, inplace=True)

# TECHNIQUE #3:
#shirts['ssize'] = shirts['ssize'].apply({'xxlarge':4,
'xlarge':4, 'large':3, 'medium':2, 'small':1, 'xsmall':1}.
get)

# TECHNIQUE #4:
shirts['ssize'] = shirts['ssize'].replace(regex='xlarge',
value=4)
shirts['ssize'] = shirts['ssize'].replace(regex='large',
value=3)
shirts['ssize'] = shirts['ssize'].replace(regex='medium',
value=2)
shirts['ssize'] = shirts['ssize'].replace(regex='small',
value=1)

print("shirts after:")
print(shirts)
```

Listing 4.9 starts with a code block of six statements that uses direct comparison with strings to make numeric replacements. For example, the following code snippet replaces all occurrences of the string `xxlarge` with the value 4:

```
shirts.loc[shirts['ssize']=='xxlarge','size'] = 4
```

The second code block consists of six statements that use the `replace()` method to perform the same updates, an example of which is shown here:

```
shirts['ssize'].replace('xxlarge', 4, inplace=True)
```

The third code block consists of a single statement that uses the `apply()` method to perform the same updates, as shown here:

```
shirts['ssize'] = shirts['ssize'].apply({'xxlarge':4,
'xlarge':4, 'large':3, 'medium':2, 'small':1, 'xsmall':1}.
get)
```

The fourth code block consists of four statements that use regular expressions to perform the same updates, an example of which is shown here:

```
shirts['ssize'] = shirts['ssize'].replace(regex='xlarge',
value=4)
```

Since the preceding code snippet matches `xxlarge` as well as `xlarge`, we only need four statements instead of six statements. If you are unfamiliar with regular expressions, you can read the appendix that contains an assortment of regular expressions. Now launch the code in Listing 4.9 and you will see the following output:

```
shirts before
     type     size
0   shirt  xxlarge
1   shirt  xxlarge
2   shirt   xlarge
3   shirt   xlarge
4   shirt   xlarge
5   shirt    large
6   shirt   medium
7   shirt    small
8   shirt    small
9   shirt   xsmall
10  shirt   xsmall
11  shirt   xsmall

shirts after:
     type  size
0   shirt     4
1   shirt     4
2   shirt     4
3   shirt     4
4   shirt     4
5   shirt     3
6   shirt     2
7   shirt     1
8   shirt     1
9   shirt     1
10  shirt     1
11  shirt     1
```

MATCHING AND SPLITTING STRINGS IN PANDAS

Listing 4.10 displays the contents of `shirts_str.py` that illustrates how to match a column value with an initial string and also how to split a column value based on a letter.

LISTING 4.10: shirts_str.py

```
import pandas as pd

shirts = pd.read_csv("shirts.csv")
print("shirts:")
print(shirts)
print()

print("shirts starting with xl:")
print(shirts[shirts.ssize.str.startswith('xl')])
print()

print("Exclude 'xlarge' shirts:")
print(shirts[shirts['ssize'] != 'xlarge'])
print()
```

```
print("first three letters:")
shirts['sub1'] = shirts['ssize'].str[:3]
print(shirts)
print()

print("split ssize on letter 'a':")
shirts['sub2'] = shirts['ssize'].str.split('a')
print(shirts)
print()

print("Rows 3 through 5 and column 2:")
print(shirts.iloc[2:5, 2])
print()
```

Listing 4.10 initializes the `DataFrame df` with the contents of the CSV file `shirts.csv`, and then displays the contents of the dataframe. The next code snippet in Listing 4.10 uses the `startswith()` method to match the shirt types that start with the letters `xl`, followed by a code snippet that displays the shorts whose size does not equal the string `xlarge`.

The next code snippet uses the construct `str[:3]` to display the first three letters of the shirt types, followed by a code snippet that uses the `split()` method to split the shirt types based on the letter "a". The final code snippet invokes `iloc[2:5,2]` to display the contents of rows 3 through 5 inclusive, and only the second column. The output of Listing 4.10 is here:

```
shirts:
     type    ssize
0   shirt  xxlarge
1   shirt  xxlarge
2   shirt   xlarge
3   shirt   xlarge
4   shirt   xlarge
5   shirt    large
6   shirt   medium
7   shirt    small
8   shirt    small
9   shirt   xsmall
10  shirt   xsmall
11  shirt   xsmall

shirts starting with xl:
    type   ssize
2  shirt  xlarge
3  shirt  xlarge
4  shirt  xlarge

Exclude 'xlarge' shirts:
     type    ssize
0   shirt  xxlarge
1   shirt  xxlarge
5   shirt    large
6   shirt   medium
7   shirt    small
```

```
8   shirt    small
9   shirt   xsmall
10  shirt   xsmall
11  shirt   xsmall

first three letters:
     type    ssize sub1
0   shirt  xxlarge  xxl
1   shirt  xxlarge  xxl
2   shirt   xlarge  xla
3   shirt   xlarge  xla
4   shirt   xlarge  xla
5   shirt    large  lar
6   shirt   medium  med
7   shirt    small  sma
8   shirt    small  sma
9   shirt   xsmall  xsm
10  shirt   xsmall  xsm
11  shirt   xsmall  xsm

split ssize on letter 'a':
     type    ssize sub1        sub2
0   shirt  xxlarge  xxl  [xxl, rge]
1   shirt  xxlarge  xxl  [xxl, rge]
2   shirt   xlarge  xla   [xl, rge]
3   shirt   xlarge  xla   [xl, rge]
4   shirt   xlarge  xla   [xl, rge]
5   shirt    large  lar    [l, rge]
6   shirt   medium  med    [medium]
7   shirt    small  sma    [sm, ll]
8   shirt    small  sma    [sm, ll]
9   shirt   xsmall  xsm   [xsm, ll]
10  shirt   xsmall  xsm   [xsm, ll]
11  shirt   xsmall  xsm   [xsm, ll]

Rows 3 through 5 and column 2:
2    xlarge
3    xlarge
4    xlarge
Name: ssize, dtype: object
```

MERGING AND SPLITTING COLUMNS IN PANDAS

Listing 4.11 displays the contents of `employees.csv` and Listing 4.12 displays the contents of `emp_merge_split.py` that illustrate how to merge columns and split columns of a CSV file.

LISTING 4.11: employees.csv

```
name,year,month
Jane-Smith,2015,Aug
Dave-Smith,2020,Jan
Jane-Jones,2018,Dec
```

```
Jane-Stone,2017,Feb
Dave-Stone,2014,Apr
Mark-Aster,,Oct
Jane-Jones,NaN,Jun
```

LISTING 4.12: emp_merge_split.py

```
import pandas as pd

emps = pd.read_csv("employees.csv")
print("emps:")
print(emps)
print()

emps['year']  = emps['year'].astype(str)
emps['month'] = emps['month'].astype(str)

# separate column for first name and for last name:
emps['fname'],emps['lname'] = emps['name'].str.
split("-",1).str

# concatenate year and month with a "#" symbol:
emps['hdate1'] = emps['year'].
astype(str)+"#"+emps['month'].astype(str)

# concatenate year and month with a "-" symbol:
emps['hdate2'] = emps[['year','month']].agg('-'.join, axis=1)

print(emps)
print()
```

Listing 4.12 initializes the dataframe `df` with the contents of the CSV file `employees.csv`, and then displays the contents of the dataframe. The next pair of code snippets invoke the `astype()` method to convert the contents of the `year` and `month` columns to strings.

The next code snippet in Listing 4.12 uses the `split()` method to split the name column into the columns `fname` and `lname` that contain the first name and last name, respectively, of each employee's name:

```
emps['fname'],emps['lname'] = emps['name'].str.
split("-",1).str
```

The next code snippet concatenates the contents of the year and month string with a "#" character to create a new column called `hdate1`, as shown here:

```
emps['hdate1'] = emps['year'].
astype(str)+"#"+emps['month'].astype(str)
```

The final code snippet concatenates the contents of the `year` and `month` string with a "-" to create a new column called `hdate2`, as shown here:

```
emps['hdate2'] = emps[['year','month']].agg('-'.join, axis=1)
```

Now launch the code in Listing 4.12 and you will see the following output:

```
emps:
        name    year month
0  Jane-Smith  2015.0   Aug
1  Dave-Smith  2020.0   Jan
2  Jane-Jones  2018.0   Dec
3  Jane-Stone  2017.0   Feb
4  Dave-Stone  2014.0   Apr
5  Mark-Aster     NaN   Oct
6  Jane-Jones     NaN   Jun

        name   year month fname lname      hdate1      hdate2
0 Jane-Smith 2015.0   Aug  Jane Smith  2015.0#Aug  2015.0-Aug
1 Dave-Smith 2020.0   Jan  Dave Smith  2020.0#Jan  2020.0-Jan
2 Jane-Jones 2018.0   Dec  Jane Jones  2018.0#Dec  2018.0-Dec
3 Jane-Stone 2017.0   Feb  Jane Stone  2017.0#Feb  2017.0-Feb
4 Dave-Stone 2014.0   Apr  Dave Stone  2014.0#Apr  2014.0-Apr
5 Mark-Aster    nan   Oct  Mark Aster     nan#Oct     nan-Oct
6 Jane-Jones    nan   Jun  Jane Jones     nan#Jun     nan-Jun
```

One other detail regarding the following code snippet:

```
#emps['fname'],emps['lname'] = emps['name'].str.
split("-",1).str
```

The following deprecation message is displayed:

```
#FutureWarning: Columnar iteration over characters
#will be deprecated in future releases.
```

COMBINING PANDAS DATAFRAMES

Pandas supports the "concat" method in dataframes in order to concatenate dataframes. Listing 4.13 displays the contents of concat_frames.py that illustrates how to combine two Pandas DataFrames.

LISTING 4.13: concat_frames.py

```
import pandas as pd

can_weather = pd.DataFrame({
    "city": ["Vancouver","Toronto","Montreal"],
    "temperature": [72,65,50],
    "humidity": [40, 20, 25]
})

us_weather = pd.DataFrame({
    "city": ["SF","Chicago","LA"],
    "temperature": [60,40,85],
    "humidity": [30, 15, 55]
})

df = pd.concat([can_weather, us_weather])
print(df)
```

The first line in Listing 4.13 is an `import` statement, followed by the definition of the `Pandas DataFrames` `can_weather` and `us_weather` that contain weather-related information for cities in Canada and the United States, respectively. The `Pandas DataFrame` `df` is the concatenation of `can_weather` and `us_weather`. The output from Listing 4.13 is here:

```
0  Vancouver        40           72
1    Toronto        20           65
2   Montreal        25           50
0         SF        30           60
1    Chicago        15           40
2         LA        55           85
```

DATA MANIPULATION WITH PANDAS DATAFRAMES

As a simple example, suppose that we have a two-person company that keeps track of income and expenses on a quarterly basis, and we want to calculate the profit/loss for each quarter as well as the overall profit/loss.

Listing 4.14 displays the contents of `pandas_quarterly_df1.py` that illustrates how to define a `Pandas DataFrame` consisting of income-related values.

LISTING 4.14: pandas_quarterly_df1.py

```
import pandas as pd

summary = {
    'Quarter': ['Q1', 'Q2', 'Q3', 'Q4'],
    'Cost':    [23500, 34000, 57000, 32000],
    'Revenue': [40000, 40000, 40000, 40000]
}

df = pd.DataFrame(summary)

print("Entire Dataset:\n",df)
print("Quarter:\n",df.Quarter)
print("Cost:\n",df.Cost)
print("Revenue:\n",df.Revenue)
```

Listing 4.14 defines the variable `summary` that contains hard-coded quarterly information about cost and revenue for our two-person company. In general, these hard-coded values would be replaced by data from another source (such as a CSV file), so think of this code sample as a simple way to illustrate some of the functionality that is available in `Pandas DataFrame`.

The variable `df` is a `Pandas DataFrame` based on the data in the `summary` variable. The three `print()` statements display the quarters, the cost per quarter, and the revenue per quarter. The output from Listing 4.14 is here:

```
Entire Dataset:
     Cost Quarter  Revenue
0  23500      Q1    40000
1  34000      Q2    60000
2  57000      Q3    50000
3  32000      Q4    30000
Quarter:
 0    Q1
1    Q2
2    Q3
3    Q4
Name: Quarter, dtype: object
Cost:
 0    23500
1    34000
2    57000
3    32000
Name: Cost, dtype: int64
Revenue:
 0    40000
1    60000
2    50000
3    30000
Name: Revenue, dtype: int64
```

DATA MANIPULATION WITH PANDAS DATAFRAMES (2)

In this section, let's suppose that we have a two-person company that keeps track of income and expenses on a quarterly basis, and we want to calculate the profit/loss for each quarter as well as the overall profit/loss.

Listing 4.15 displays the contents of `pandas_quarterly_df2.py` that illustrates how to define a `Pandas DataFrame` consisting of income-related values.

LISTING 4.15: pandas_quarterly_df2.py

```
import pandas as pd

summary = {
    'Quarter': ['Q1', 'Q2', 'Q3', 'Q4'],
    'Cost':    [-23500, -34000, -57000, -32000],
    'Revenue': [40000, 40000, 40000, 40000]
}

df = pd.DataFrame(summary)
print("First Dataset:\n",df)

df['Total'] = df.sum(axis=1)
print("Second Dataset:\n",df)
```

Listing 4.15 defines the variable `summary` that contains quarterly information about cost and revenue for our two-person company. The variable `df` is

a Pandas DataFrame based on the data in the summary variable. The three print statements display the quarters, the cost per quarter, and the revenue per quarter. The output from Listing 4.15 is here:

```
First Dataset:
     Cost Quarter  Revenue
0 -23500      Q1    40000
1 -34000      Q2    60000
2 -57000      Q3    50000
3 -32000      Q4    30000
Second Dataset:
     Cost Quarter  Revenue  Total
0 -23500      Q1    40000  16500
1 -34000      Q2    60000  26000
2 -57000      Q3    50000  -7000
3 -32000      Q4    30000  -2000
```

DATA MANIPULATION WITH PANDAS DATAFRAMES (3)

Let's start with the same assumption as the previous section: we have a two-person company that keeps track of income and expenses on a quarterly basis, and we want to calculate the profit/loss for each quarter, and also the overall profit/loss. In addition, we want to compute column totals and row totals.

Listing 4.16 displays the contents of pandas_quarterly_df3.py that illustrates how to define a Pandas DataFrame consisting of income-related values.

LISTING 4.16: pandas_quarterly_df3.py

```
import pandas as pd

summary = {
    'Quarter': ['Q1', 'Q2', 'Q3', 'Q4'],
    'Cost':    [_23500, _34000, _57000, _32000],
    'Revenue': [40000, 40000, 40000, 40000]
}

df = pd.DataFrame(summary)
print("First Dataset:\n",df)

df['Total'] = df.sum(axis=1)
df.loc['Sum'] = df.sum()
print("Second Dataset:\n",df)

# or df.loc['avg'] / 3
#df.loc['avg'] = df[:3].mean()
#print("Third Dataset:\n",df)
```

Listing 4.16 defines the variable summary that contains quarterly information about cost and revenue for our two-person company. The variable df is a Pandas DataFrame based on the data in the summary variable. The three

print statements display the quarters, the cost per quarter, and the revenue per quarter. The output from Listing 4.16 is here:

```
First Dataset:
     Cost Quarter  Revenue
0 -23500      Q1    40000
1 -34000      Q2    60000
2 -57000      Q3    50000
3 -32000      Q4    30000
Second Dataset:
        Cost   Quarter  Revenue  Total
0    -23500        Q1    40000  16500
1    -34000        Q2    60000  26000
2    -57000        Q3    50000  -7000
3    -32000        Q4    30000  -2000
Sum -146500  Q1Q2Q3Q4   180000  33500
```

PANDAS DATAFRAMES AND CSV FILES

The code samples in several earlier sections contain hard-coded data inside the Python scripts. However, it's also very common to read data from a CSV file. You can use the Python csv.reader() function, the NumPy loadtxt() function, or the Pandas read_csv() function (shown in this section) to read the contents of CSV files.

Listing 4.17 displays the contents of the CSV file weather_data.csv, and Listing 4.18 displays the contents of weather_data.py that illustrates how to read the CSV weather_data.csv.

LISTING 4.17: weather_data.py

```
day,temperature,windspeed,event
7/1/2018,42,16,Rain
7/2/2018,45,3,Sunny
7/3/2018,78,12,Snow
7/4/2018,74,9,Snow
7/5/2018,42,24,Rain
7/6/2018,51,32,Sunny
```

LISTING 4.18: weather_data.py

```
import pandas as pd

df = pd.read_csv("weather_data.csv")

print(df)
print(df.shape)  # rows, columns
print(df.head()) # df.head(3)
print(df.tail())
print(df[1:3])
print(df.columns)
print(type(df['day']))
```

```
print(df[['day','temperature']])
print(df['temperature'].max())
```

Listing 4.18 invokes the `Pandas read_csv()` function to read the contents of the CSV file `weather_data.csv`, followed by a set of Python `print()` statements that display various portions of the CSV file. The output from Listing 4.18 is here:

```
        day  temperature  windspeed     event
0  7/1/2018           42         16      Rain
1  7/2/2018           45          3     Sunny
2  7/3/2018           78         12      Snow
3  7/4/2018           74          9      Snow
4  7/5/2018           42         24      Rain
5  7/6/2018           51         32     Sunny
(6, 4)
        day  temperature  windspeed     event
0  7/1/2018           42         16      Rain
1  7/2/2018           45          3     Sunny
2  7/3/2018           78         12      Snow
3  7/4/2018           74          9      Snow
4  7/5/2018           42         24      Rain
        day  temperature  windspeed     event
1  7/2/2018           45          3     Sunny
2  7/3/2018           78         12      Snow
3  7/4/2018           74          9      Snow
4  7/5/2018           42         24      Rain
5  7/6/2018           51         32     Sunny
        day  temperature  windspeed     event
1  7/2/2018           45          3     Sunny
2  7/3/2018           78         12      Snow
Index(['day', 'temperature', 'windspeed', 'event'],
dtype='object')
<class 'pandas.core.series.Series'>
        day  temperature
0  7/1/2018           42
1  7/2/2018           45
2  7/3/2018           78
3  7/4/2018           74
4  7/5/2018           42
5  7/6/2018           51
78
```

In some situations, you might need to apply Boolean conditional logic to "filter out" some rows of data, based on a conditional condition that's applied to a column value.

Listing 4.19 displays the contents of the CSV file `people.csv`, and Listing 4.20 displays the contents of `people_pandas.py` that illustrates how to define a `Pandas DataFrame` that reads the CSV file and manipulates the data.

LISTING 4.19: people.csv

```
fname,lname,age,gender,country
john,smith,30,m,usa
```

```
jane,smith,31,f,france
jack,jones,32,m,france
dave,stone,33,m,italy
sara,stein,34,f,germany
eddy,bower,35,m,spain
```

LISTING 4.20: people_pandas.py

```
import pandas as pd

df = pd.read_csv('people.csv')
df.info()
print('fname:')
print(df['fname'])
print('_____________')
print('age over 33:')
print(df['age'] > 33)
print('_____________')
print('age over 33:')
myfilter = df['age'] >  33
print(df[myfilter])
```

Listing 4.20 populates the `Pandas DataFrame df` with the contents of the CSV file `people.csv`. The next portion of Listing 4.20 displays the structure of `df`, followed by the first names of all the people. The next portion of Listing 4.20 displays a tabular list of six rows containing either True or False, depending on whether a person is over 33 or at most 33, respectively. The final portion of Listing 4.20 displays a tabular list of two rows containing all the details of the people who are over 33. The output from Listing 4.20 is here:

```
myfilter = df['age'] >  33
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6 entries, 0 to 5
Data columns (total 5 columns):
fname      6 non_null object
lname      6 non_null object
age        6 non_null int64
gender     6 non_null object
country    6 non_null object
dtypes: int64(1), object(4)
memory usage: 320.0+ bytes
fname:
0    john
1    jane
2    jack
3    dave
4    sara
5    eddy
Name: fname, dtype: object

_____________
age over 33:
0    False
1    False
2    False
```

```
3    False
4     True
5     True
Name: age, dtype: bool
_____________
age over 33:
  fname  lname  age gender country
4  sara  stein   34      f  france
5  eddy  bower   35      m  france
```

PANDAS DATAFRAMES AND EXCEL SPREADSHEETS

Listing 4.21 displays the contents of `write_people_xlsx.py` that illustrates how to read data from a CSV file and then create an Excel spreadsheet with that data.

LISTING 4.21: write_people_xlsx.py

```
import pandas as pd

df1 = pd.read_csv("people.csv")
df1.to_excel("people.xlsx")

#optionally specify the sheet name:
#df1.to_excel("people.xlsx", sheet_name='Sheet_name_1')
```

Listing 4.21 initializes the `Pandas DataFrame df1` with the contents of the CSV file `people.csv`, and then invokes the `to_excel()` method in order to save the contents of the dataframe to the Excel spreadsheet `people.xlsx`.

Listing 4.22 displays the contents of `read_people_xlsx.py` that illustrates how to read data from the Excel spreadsheet `people.xlsx` and create a `Pandas DataFrame` with that data.

LISTING 4.22: read_people_xlsx.py

```
import pandas as pd

df = pd.read_excel("people.xlsx")
print("Contents of Excel spreadsheet:")
print(df)
```

Listing 4.22 is straightforward: the `Pandas DataFrame df` is initialized with the contents of the spreadsheet `people.xlsx` (whose contents are the same as `people.csv`) via the `Pandas` function `read_excel()`. The output from Listing 4.22 is here:

```
df1:
   Unnamed: 0 fname  lname  age gender  country
0           0  john  smith   30      m      usa
1           1  jane  smith   31      f   france
2           2  jack  jones   32      m   france
```

```
3           3  dave  stone   33      m    italy
4           4  sara  stein   34      f  germany
5           5  eddy  bower   35      m    spain
```

SELECT, ADD, AND DELETE COLUMNS IN DATAFRAMES

This section contains short code blocks that illustrate how to perform operations on a dataframe that resemble the operations on a Python dictionary. For example, getting, setting, and deleting columns works with the same syntax as the analogous Python `dict` operations, as shown here:

```
df = pd.DataFrame.from_dict(dict([('A',[1,2,3]),('B',[4,5,6])]),
             orient='index', columns=['one', 'two', 'three'])

print(df)
```

The output from the preceding code snippet is here:

```
   one  two  three
A    1    2      3
B    4    5      6
```

Now look at the following operation that appends a new column to the contents of the `DataFrame df`:

```
df['four'] = df['one'] * df['two']
print(df)
```

The output from the preceding code block is here:

```
   one  two  three  four
A    1    2      3     2
B    4    5      6    20
```

The following operation squares the contents of a column in the `DataFrame df`:

```
df['three'] = df['two'] * df['two']
print(df)
```

The output from the preceding code block is here:

```
   one  two  three  four
A    1    2      4     2
B    4    5     25    20
```

The following operation inserts a column of random numbers in index position 1 (which is the second column) in the `DataFrame df`:

```
import numpy as np
rand = np.random.randn(2)
df.insert(1, 'random', rand)
print(df)
```

The output from the preceding code block is here:

```
   one    random  two  three  four
A    1 -1.703111    2      4     2
B    4  1.139189    5     25    20
```

The following operation appends a new column called flag that contains True or False, based on whether or not the numeric value in the "one" column is greater than 2:

```
import numpy as np
rand = np.random.randn(2)
df.insert(1, 'random', rand)
print(df)
```

The output from the preceding code block is here:

```
   one    random  two  three  four   flag
A    1 -1.703111    2      4     2  False
B    4  1.139189    5     25    20   True
```

Columns can be deleted, as shown in following code snippet that deletes the "two" column:

```
del df['two']
print(df)
```

The output from the preceding code block is here:

```
   one    random  three  four   flag
A    1 -0.460401      4     2  False
B    4  1.211468     25    20   True
```

Columns can be deleted via the `pop()` method, as shown in following code snippet that deletes the "three" column:

```
three = df.pop('three')
print(df)
```

```
   one    random  four    flag
A    1 -0.544829     2  False
B    4  0.581476    20   True
```

When inserting a scalar value, it will naturally be propagated to fill the column:

```
df['foo'] = 'bar'
print(df)
```

The output from the preceding code snippet is here:

```
   one    random  four   flag  foo
A    1 -0.187331     2  False  bar
B    4 -0.169672    20   True  bar
```

HANDLING OUTLIERS IN PANDAS

If you are unfamiliar with outliers and anomalies, please read the sections in Chapter 2 that discuss these two concepts, because this section uses Pandas to find outliers in a dataset. The key idea involves finding the "z score" of the values in the dataset, which involves calculating the mean sigma and standard deviation std, and then mapping each value x in the dataset to the value (x-sigma)/std.

Next, you specify a value of z (such as 3) and find the rows whose z score is greater than 3. These are the rows that contain values that are considered outliers. *Note that a suitable value for the z score is your decision (not some other external factor).*

Listing 4.23 displays the contents of outliers_zscores.py that illustrates how to find rows of a dataset whose z-score is greater than (or less than) a specified value.

LISTING 4.23: outliers_zscores.py

```
import numpy as np
import pandas as pd
from scipy import stats
from sklearn import datasets

df = datasets.load_iris()
columns = df.feature_names
iris_df = pd.DataFrame(df.data)
iris_df.columns = columns

print("=> iris_df.shape:",iris_df.shape)
print(iris_df.head())
print()

z = np.abs(stats.zscore(iris_df))
print("z scores for iris:")
print("z.shape:",z.shape)

upper = 2.5
lower = 0.01
print("=> upper outliers:")
print(z[np.where(z > upper)])
print()

outliers = iris_df[z < lower]
print("=> lower outliers:")
print(outliers)
print()
```

Listing 4.23 initializes the variable df with the contents of the built-in Iris dataset. Next, the variable columns is initialized with the column names, and the DataFrame iris_df is initialized from the contents of df.data that contains the actual data for the Iris dataset. In addition, iris_df.columns is initialized with the contents of the variable columns.

The next portion of Listing 4.23 displays the shape of the `DataFrame` `iris_df`, followed by the `zscore` of the `iris_df` `DataFrame`, which is computed by subtracting the mean and then dividing by the standard deviation (performed for each row).

The last two portions of Listing 4.23 display the outliers (if any) whose `zscore` is outside the interval [0.01, 2.5]. Launch the code in Listing 4.23 and you will see the following output:

```
=> iris_df.shape: (150, 4)
   sepal length (cm)  sepal width (cm)  petal length (cm)
petal width  (cm)
0                5.1               3.5               1.4
0.2
1                4.9               3.0               1.4
0.2
2                4.7               3.2               1.3
0.2
3                4.6               3.1               1.5
0.2
4                5.0               3.6               1.4
0.2

z scores for iris:
z.shape: (150, 4)

=> upper outliers:
[3.09077525 2.63038172]

=> lower outliers:
    sepal length (cm)  sepal width (cm)  petal length (cm)
petal width (cm)
73                6.1               2.8                4.7
1.2
82                5.8               2.7                3.9
1.2
90                5.5               2.6                4.4
1.2
92                5.8               2.6                4.0
1.2
95                5.7               3.0                4.2
1.2
```

PANDAS DATAFRAMES AND SCATTERPLOTS

Listing 4.24 displays the contents of `pandas_scatter_df.py` that illustrates how to generate a scatterplot from a `Pandas DataFrame`.

LISTING 4.24: pandas_scatter_df.py

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

```
from pandas import read_csv
from pandas.plotting import scatter_matrix

myarray = np.array([[10,30,20],
[50,40,60],[1000,2000,3000]])

rownames = ['apples', 'oranges', 'beer']
colnames = ['January', 'February', 'March']

mydf = pd.DataFrame(myarray, index=rownames, columns=colnames)

print(mydf)
print(mydf.describe())

scatter_matrix(mydf)
plt.show()
```

Listing 4.24 starts with various `import` statements, followed by the definition of the `NumPy` array `myarray`. Next, the variables `myarray` and `colnames` are initialized with values for the rows and columns, respectively. The next portion of Listing 4.24 initializes the `Pandas DataFrame mydf` so that the rows and columns are labeled in the output, as shown here:

```
           January     February        March
apples          10           30           20
oranges         50           40           60
beer          1000         2000         3000
           January     February        March
count     3.000000     3.000000     3.000000
mean    353.333333   690.000000  1026.666667
std     560.386771  1134.504297  1709.073823
min      10.000000    30.000000    20.000000
25%      30.000000    35.000000    40.000000
50%      50.000000    40.000000    60.000000
75%     525.000000  1020.000000  1530.000000
max    1000.000000  2000.000000  3000.0000000
```

PANDAS DATAFRAMES AND SIMPLE STATISTICS

Listing 4.25 displays the contents of `housing_stats.py` that illustrates how to gather basic statistics from data in a `Pandas DataFrame`.

LISTING 4.25: housing_stats.py

```
import pandas as pd

df = pd.read_csv("Housing.csv")

minimum_bdrms = df["bedrooms"].min()
median_bdrms  = df["bedrooms"].median()
maximum_bdrms = df["bedrooms"].max()
```

```
print("minimum # of bedrooms:",minimum_bdrms)
print("median  # of bedrooms:",median_bdrms)
print("maximum # of bedrooms:",maximum_bdrms)
print("")

print("median values:",df.median().values)
print("")

prices = df["price"]
print("first 5 prices:")
print(prices.head())
print("")

median_price = df["price"].median()
print("median price:",median_price)
print("")

corr_matrix = df.corr()
print("correlation matrix:")
print(corr_matrix["price"].sort_values(ascending=False))
```

Listing 4.25 initializes the `Pandas DataFrame df` with the contents of the CSV file `housing.csv`. The next three variables are initialized with the minimum, median, and maximum number of bedrooms, respectively, and then these values are displayed.

The next portion of Listing 4.25 initializes the variable `prices` with the contents of the `prices` column of the `Pandas DataFrame df`. Next, the first five rows are printed via the `prices.head()` statement, followed by the median value of the prices.

The final portion of Listing 4.25 initializes the variable `corr_matrix` with the contents of the correlation matrix for the `Pandas DataFrame df`, and then displays its contents. The output from Listing 4.25 is here:

```
Apples
10
```

FINDING DUPLICATE ROWS IN PANDAS

Listing 4.26 displays the contents of `duplicates.csv`, and Listing 4.27 displays the contents of `duplicates.py` that illustrates how to find duplicate rows in a `Pandas DataFrame`.

LISTING 4.26: duplicates.csv

```
fname,lname,level,dept,state
Jane,Smith,Senior,Sales,California
Dave,Smith,Senior,Devel,California
Jane,Jones,Year1,Mrktg,Illinois
Jane,Jones,Year1,Mrktg,Illinois
Jane,Stone,Senior,Mrktg,Arizona
Dave,Stone,Year2,Devel,Arizona
```

```
Mark,Aster,Year3,BizDev,Florida
Jane,Jones,Year1,Mrktg,Illinois
```

LISTING 4.27: duplicates.py

```
import pandas as pd

df = pd.read_csv("duplicates.csv")
print("Contents of dataframe:")
print(df)
print()

print("Duplicate rows:")
#df2 = df.duplicated(subset=None)
df2 = df.duplicated(subset=None, keep='first')
print(df2)
print()

print("Duplicate first names:")
df3 = df[df.duplicated(['fname'])]
print(df3)
print()

print("Duplicate first name and level:")
df3 = df[df.duplicated(['fname','level'])]
print(df3)
print()
```

Listing 4.27 initializes the `DataFrame df` with the contents of the CSV file `duplicates.csv`, and then displays the contents of the dataframe. The next portion of Listing 4.27 displays the duplicate rows by invoking the `duplicated()` method, whereas the following portion of Listing 4.27 displays only the first name `fname` of the duplicate rows. The final portion of Listing 4.27 displays the first name `fname` as well as the level of the duplicate rows. Launch the code in Listing 4.27 and you will see the following output:

```
Contents of dataframe:
  fname  lname   level    dept       state
0  Jane  Smith  Senior   Sales  California
1  Dave  Smith  Senior   Devel  California
2  Jane  Jones   Year1   Mrktg    Illinois
3  Jane  Jones   Year1   Mrktg    Illinois
4  Jane  Stone  Senior   Mrktg     Arizona
5  Dave  Stone   Year2   Devel     Arizona
6  Mark  Aster   Year3  BizDev     Florida
7  Jane  Jones   Year1   Mrktg    Illinois

Duplicate rows:
0    False
1    False
2    False
3     True
4    False
5    False
```

```
6    False
7     True
dtype: bool

Duplicate first names:
  fname  lname   level   dept     state
2  Jane  Jones   Year1  Mrktg  Illinois
3  Jane  Jones   Year1  Mrktg  Illinois
4  Jane  Stone  Senior  Mrktg   Arizona
5  Dave  Stone   Year2  Devel   Arizona
7  Jane  Jones   Year1  Mrktg  Illinois

Duplicate first name and level:
  fname  lname   level   dept     state
3  Jane  Jones   Year1  Mrktg  Illinois
4  Jane  Stone  Senior  Mrktg   Arizona
7  Jane  Jones   Year1  Mrktg  Illinois
```

Listing 4.28 displays the contents of `drop_duplicates.py` that illustrates how to drop duplicate rows in a `Pandas DataFrame`.

LISTING 4.28: drop_duplicates.py

```
import pandas as pd

df = pd.read_csv("duplicates.csv")
print("Contents of dataframe:")
print(df)
print()

fname_filtered = df.drop_duplicates(['fname'])
print("Drop duplicate first names:")
print(fname_filtered)
print()

fname_lname_filtered = df.drop_duplicates(['fname','lname'])
print("Drop duplicate first and last names:")
print(fname_lname_filtered)
print()
```

Listing 4.28 initializes the `DataFrame df` with the contents of the CSV file `duplicates.csv`, and then displays the contents of the dataframe. The next portion of Listing 4.28 deletes the rows that have duplicate `fname` values, followed by a code block that drops rows with duplicate `fname` and `lname` values. Launch the code in Listing 4.28 and you will see the following output:

```
Contents of dataframe:
  fname  lname   level    dept       state
0  Jane  Smith  Senior   Sales  California
1  Dave  Smith  Senior   Devel  California
2  Jane  Jones   Year1   Mrktg    Illinois
3  Jane  Jones   Year1   Mrktg    Illinois
4  Jane  Stone  Senior   Mrktg     Arizona
5  Dave  Stone   Year2   Devel     Arizona
```

```
6  Mark  Aster   Year3  BizDev     Florida
7  Jane  Jones   Year1   Mrktg    Illinois

Drop duplicate first names:
  fname  lname   level    dept       state
0  Jane  Smith  Senior   Sales  California
1  Dave  Smith  Senior   Devel  California
6  Mark  Aster   Year3  BizDev     Florida

Drop duplicate first and last names:
  fname  lname   level    dept       state
0  Jane  Smith  Senior   Sales  California
1  Dave  Smith  Senior   Devel  California
2  Jane  Jones   Year1   Mrktg    Illinois
4  Jane  Stone  Senior   Mrktg     Arizona
5  Dave  Stone   Year2   Devel     Arizona
6  Mark  Aster   Year3  BizDev     Florida
```

FINDING MISSING VALUES IN PANDAS

Listing 4.29 displays the contents of `employees2.csv`, and Listing 4.30 displays the contents of `missing_values.py` that illustrates how to display rows of a dataframe that have missing values in a `Pandas DataFrame`.

LISTING 4.29: employees2.csv

```
name,year,month
Jane-Smith,2015,Aug
Jane-Smith,2015,Aug
Dave-Smith,2020,
Dave-Stone,,Apr
Jane-Jones,2018,Dec
Jane-Stone,2017,Feb
Jane-Stone,2017,Feb
Mark-Aster,,Oct
Jane-Jones,NaN,Jun
```

LISTING 4.30: missing_values.py

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

df = pd.read_csv("employees2.csv")

print("=> contents of CSV file:")
print(df)
print()

#NA:  Not Available (Pandas)
#NaN: Not a Number (Pandas)
#NB:  NumPy uses np.nan() to check for NaN values
```

```
# the preceding comment block describes
# the different meanings of NA and NaN,
# and the use of np.nan() in NumPy

print("=> any NULL values per column?")
print(df.isnull().any())
print()

print("=> count of NAN/MISSING values in each column:")
print(df.isnull().sum())
print()

print("=> count of NAN/MISSING values in each column:")
print(pd.isna(df).sum())
print()

print("=> count of NAN/MISSING values in each column (sorted):")
print(df.isnull().sum().sort_values(ascending=False))
print()

nan_null = df.isnull().sum().sum()
miss_values = df.isnull().any().sum()

print("=> count of NaN/MISSING values:",nan_null)
print("=> count of MISSING values:",miss_values)
print("=> count of NaN values:",nan_null-miss_values)
```

Listing 4.30 initializes the dataframe `df` with the contents of the CSV file `employees2.csv`, and then displays the contents of the dataframe. The next portion of Listing 4.30 displays the number of null values that appear in any row or column. The following portion of Listing 4.30 displays the fields and the names of the fields that have null values.

The next portion of Listing 4.30 displays the number of duplicate rows, followed by the row numbers that are duplicates. Launch the code in Listing 4.30 and you will see the following output:

```
=> contents of CSV file:
         name    year month
0  Jane-Smith  2015.0   Aug
1  Jane-Smith  2015.0   Aug
2  Dave-Smith  2020.0   NaN
3  Dave-Stone     NaN   Apr
4  Jane-Jones  2018.0   Dec
5  Jane-Stone  2017.0   Feb
6  Jane-Stone  2017.0   Feb
7  Mark-Aster     NaN   Oct
8  Jane-Jones     NaN   Jun

=> any NULL values per column?
name     False
year      True
month     True
dtype: bool
```

```
=> count of NAN/MISSING values in each column:
name     0
year     3
month    1
dtype: int64

=> count of NAN/MISSING values in each column:
name     0
year     3
month    1
dtype: int64

=> count of NAN/MISSING values in each column (sorted):
year     3
month    1
name     0
dtype: int64

=> count of NaN/MISSING values: 4
=> count of MISSING values: 2
=> count of NaN values: 2
```

SORTING DATAFRAMES IN PANDAS

Listing 4.31 displays the contents of `sort_df.py` that illustrates how to sort the rows in a `Pandas DataFrame`.

LISTING 4.31: sort_df.py

```
import pandas as pd

df = pd.read_csv("duplicates.csv")
print("Contents of dataframe:")
print(df)
print()

df.sort_values(by=['fname'], inplace=True)
print("Sorted (ascending) by first name:")
print(df)
print()

df.sort_values(by=['fname'], inplace=True,ascending=False)
print("Sorted (descending) by first name:")
print(df)
print()

df.sort_values(by=['fname','lname'], inplace=True)
print("Sorted (ascending) by first name and last name:")
print(df)
print()
```

Listing 4.31 initializes the `DataFrame df` with the contents of the CSV file `duplicates.csv`, and then displays the contents of the dataframe. The next portion of Listing 4.31 displays the rows in *ascending* order based on the

first name, and the following code block displays the rows in *descending* order based on the first name. The final code block in Listing 4.31 displays the rows in ascending order based on the first name as well as the last name. Launch the code in Listing 4.31 and you will see the following output:

```
Contents of dataframe:
  fname  lname   level    dept       state
0  Jane  Smith  Senior   Sales  California
1  Dave  Smith  Senior   Devel  California
2  Jane  Jones   Year1   Mrktg    Illinois
3  Jane  Jones   Year1   Mrktg    Illinois
4  Jane  Stone  Senior   Mrktg     Arizona
5  Dave  Stone   Year2   Devel     Arizona
6  Mark  Aster   Year3  BizDev     Florida
7  Jane  Jones   Year1   Mrktg    Illinois

Sorted (ascending) by first name:
  fname  lname   level    dept       state
1  Dave  Smith  Senior   Devel  California
5  Dave  Stone   Year2   Devel     Arizona
0  Jane  Smith  Senior   Sales  California
2  Jane  Jones   Year1   Mrktg    Illinois
3  Jane  Jones   Year1   Mrktg    Illinois
4  Jane  Stone  Senior   Mrktg     Arizona
7  Jane  Jones   Year1   Mrktg    Illinois
6  Mark  Aster   Year3  BizDev     Florida

Sorted (descending) by first name:
  fname  lname   level    dept       state
6  Mark  Aster   Year3  BizDev     Florida
0  Jane  Smith  Senior   Sales  California
2  Jane  Jones   Year1   Mrktg    Illinois
3  Jane  Jones   Year1   Mrktg    Illinois
4  Jane  Stone  Senior   Mrktg     Arizona
7  Jane  Jones   Year1   Mrktg    Illinois
1  Dave  Smith  Senior   Devel  California
5  Dave  Stone   Year2   Devel     Arizona

Sorted (ascending) by first name and last name:
  fname  lname   level    dept       state
1  Dave  Smith  Senior   Devel  California
5  Dave  Stone   Year2   Devel     Arizona
2  Jane  Jones   Year1   Mrktg    Illinois
3  Jane  Jones   Year1   Mrktg    Illinois
7  Jane  Jones   Year1   Mrktg    Illinois
0  Jane  Smith  Senior   Sales  California
4  Jane  Stone  Senior   Mrktg     Arizona
6  Mark  Aster   Year3  BizDev     Florida
```

WORKING WITH GROUPBY() IN PANDAS

Listing 4.32 displays the contents of `groupby1.py` that illustrates how to invoke the `Pandas groupby()` method in order to compute subtotals of feature values.

LISTING 4.32: groupby1.py

```
import pandas as pd

# colors and weights of balls:
data = {'color':['red','blue','blue','red','blue'],
        'weight':[40,50,20,30,90]}
df1 = pd.DataFrame(data)
print("df1:")
print(df1)
print()
print(df1.groupby('color').mean())
print()

red_filter = df1['color']=='red'
print(df1[red_filter])
print()
blue_filter = df1['color']=='blue'
print(df1[blue_filter])
print()

red_avg = df1[red_filter]['weight'].mean()
blue_avg = df1[blue_filter]['weight'].mean()
print("red_avg,blue_avg:")
print(red_avg,blue_avg)
print()

df2 = pd.DataFrame({'color':['blue','red'],'weight':[red_
avg,blue_avg]})
print("df2:")
print(df2)
print()
```

Listing 4.32 defines the variable `data` containing `color` and `weight` values, and then initializes the dataframe `df` with the contents of the variable `data`. The next two code blocks define `red_filter` and `blue_filter` that match the rows whose colors are red and blue, respectively, and then prints the matching rows.

The next portion of Listing 4.32 defines the two filters `red_avg` and `blue_avg` that calculate the average weight of the red values and the blue values, respectively. The last code block in Listing 4.32 defines the `DataFrame df2` with a `color` column and a `weight` column, where the latter contains the average weight of the red values and the blue values. Launch the code in Listing 4.32 and you will see the following output:

```
initial dataframe:
df1:
  color  weight
0   red      40
1  blue      50
2  blue      20
3   red      30
4  blue      90
```

```
          weight
color
blue   53.333333
red    35.000000

  color  weight
0   red      40
3   red      30

  color  weight
1  blue      50
2  blue      20
4  blue      90

red_avg,blue_avg:
35.0 53.333333333333336

df2:
  color     weight
0  blue  35.000000
1   red  53.333333
```

AGGREGATE OPERATIONS WITH THE TITANIC.CSV DATASET

Listing 4.33 displays the contents of `aggregate2.py` that illustrates how to perform aggregate operations with columns in the CSV file `titanic.csv`.

LISTING 4.33: aggregate2.py

```
import pandas as pd

#Loading titanic.csv in Seaborn:
#df = sns.load_dataset('titanic')
df = pd.read_csv("titanic.csv")

# convert floating point values to integers:
df['survived'] = df['survived'].astype(int)

# specify column and aggregate functions:
aggregates1 = {'embark_town': ['count', 'nunique', 'size']}

# group by 'deck' value and apply aggregate functions:
result = df.groupby(['deck']).agg(aggregates1)
print("=> Grouped by deck:")
print(result)
print()

# some details regarding count() and nunique():
# count() excludes NaN values whereas size() includes them
# nunique() excludes NaN values in the unique counts

# group by 'age' value and apply aggregate functions:
result2 = df.groupby(['age']).agg(aggregates1)
print("=> Grouped by age (before):")
```

```
print(result2)
print()

# some "age" values are missing (so drop them):
df = df.dropna()

# convert floating point values to integers:
df['age'] = df['age'].astype(int)

# group by 'age' value and apply aggregate functions:
result3 = df.groupby(['age']).agg(aggregates1)
print("=> Grouped by age (after):")
print(result3)
print()
```

Listing 4.33 initializes the `DataFrame` `df` with the contents of the CSV file `titanic.csv`. The next code snippet converts floating point values to integer, followed by defining the variable `aggregates1` that specifies the functions `count()`, `nunique()`, and `size()` that will be invoked on the `embark_town` field.

The next code snippet initializes the variable `result` after invoking the `groupby()` method on the `deck` field, followed by invoking the `agg()` method.

The next code block performs the same computation to initialize the variable `result2`, except that the `groupby()` function is invoked on the `age` field instead of the `embark_town` field. Notice the comment section regarding the `count()` and `nunique()` functions; let's drop the rows with missing values via `df.dropna()` and investigate how that affects the calculations.

After dropping the rows with missing values, the final code block initializes the variable `result3` in exactly the same way that `result2` was initialized. Now launch the code in Listing 4.33 and the output is shown here:

```
=> Grouped by deck:
     embark_town
           count nunique size
deck
A             15       2   15
B             45       2   47
C             59       3   59
D             33       2   33
E             32       3   32
F             13       3   13
G              4       1    4

=> Grouped by age (before):
        age
      count nunique size
age
0.42      1       1    1
0.67      1       1    1
0.75      2       1    2
0.83      2       1    2
0.92      1       1    1
...     ...     ...  ...
```

```
70.00     2        1    2
70.50     1        1    1
71.00     2        1    2
74.00     1        1    1
80.00     1        1    1

[88 rows x 3 columns]

=> Grouped by age (after):
      age
    count nunique size
age
0       1       1    1
1       1       1    1
2       3       1    3
3       1       1    1
4       3       1    3
6       1       1    1
11      1       1    1
14      1       1    1
15      1       1    1
// details omitted for brevity
60      2       1    2
61      2       1    2
62      1       1    1
63      1       1    1
64      1       1    1
65      2       1    2
70      1       1    1
71      1       1    1
80      1       1    1
```

WORKING WITH APPLY() AND MAPAPPLY() IN PANDAS

Earlier in this chapter you saw an example of the `Pandas apply()` method for modifying the categorical values of a feature in the CSV file `shirts.csv`. This section contains more examples of the `apply()` method, along with examples of the `mapappy()` method.

Listing 4.34 displays the contents of `apply1.py` that illustrates how to invoke the `Pandas apply()` method in order to compute the sum of a set of values.

LISTING 4.34: apply1.py

```
import pandas as pd

df = pd.DataFrame({'X1': [1,2,3], 'X2': [10,20,30]})

def cube(x):
  return x * x * x

df1 = df.apply(cube)
# same result:
# df1 = df.apply(lambda x: x * x * x)
```

```
print("initial dataframe:")
print(df)
print("cubed values:")
print(df1)
```

Listing 4.34 initializes the `DataFrame` `df` with columns `X1` and `X2`, where the values for `X2` are 10 times the corresponding values in `X1`. Next, the Python function `cube()` returns the cube of its argument. Listing 4.34 then defines the variable `df1` by invoking the `apply()` function, which specifies the user-defined Python function `cube()`, and then prints the values of `df` as well as `df1`. Launch the code in Listing 4.34 and you will see the following output:

```
initial dataframe:
   X1  X2
0   1  10
1   2  20
2   3  30
cubed values:
   X1     X2
0   1   1000
1   8   8000
2  27  27000
```

Listing 4.35 displays the contents of `apply2.py` that illustrates how to invoke the `Pandas apply()` method in order to compute the sum of a set of values.

LISTING 4.35: apply2.py

```
import pandas as pd
import numpy as np

df = pd.DataFrame({'X1': [10,20,30], 'X2': [50,60,70]})

df1 = df.apply(np.sum, axis=0)
df2 = df.apply(np.sum, axis=1)

print("initial dataframe:")
print(df)
print("add values (axis=0):")
print(df1)
print("add values (axis=1):")
print(df2)
```

Listing 4.35 is a variation of Listing 4.34: the variables `df1` and `df2` contain the column-wise sum and the row-wise sum, respectively, of the `DataFrame` `df`. Launch the code in Listing 4.35 and you will see the following output:

```
   X1  X2
0  10  50
1  20  60
2  30  70
```

```
add values (axis=0):
X1     60
X2    180
dtype: int64
add values (axis=1):
0     60
1     80
2    100
dtype: int64
```

Listing 4.36 displays the contents of `mapapply1.py` that illustrates how to invoke the `Pandas mapapply()` method in order to compute the sum of a set of values.

LISTING 4.36: mapapply1.py

```
import pandas as pd
import math

df = pd.DataFrame({'X1': [1,2,3], 'X2': [10,20,30]})
df1 = df.applymap(math.sqrt)

print("initial dataframe:")
print(df)
print("square root values:")
print(df1)
```

Listing 4.36 is yet another variant of Listing 4.34: in this case, the variable `df1` is defined by invoking the `applymap()` function on the variable `df`, which in turn references (but does not execute) the `math.sqrt()` function. Next, a print statement displays the contents of `df`, followed by a `print()` statement that displays the contents of `df1`: it is at this point that the built-in `math.sqrt()` function is invoked in order to calculate the square root of the values in `df`. Launch the code in Listing 4.36 and you will see the following output:

```
initial dataframe:
   X1  X2
0   1  10
1   2  20
2   3  30

square root values:
         X1        X2
0  1.000000  3.162278
1  1.414214  4.472136
2  1.732051  5.477226
```

Listing 4.37 displays the contents of `mapapply2.py` that illustrates how to invoke the `Pandas mapapply()` method in order to convert strings to lowercase and uppercase.

LISTING 4.37: mapapply2.py

```
import pandas as pd

df = pd.DataFrame({'fname': ['Jane'], 'lname': ['Smith']},
                  {'fname': ['Dave'], 'lname': ['Jones']})

df1 = df.applymap(str.lower)
df2 = df.applymap(str.upper)

print("initial dataframe:")
print(df)
print()
print("lowercase:")
print(df1)
print()
print("uppercase:")
print(df2)
print()
```

Listing 4.37 initializes the variable `df` with two first and last name pairs, and then defines the variables `df1` and `df2` by invoking the `applymap()` method to the variable `df`. The variable `df1` converts its input values to lowercase, whereas the variable `df2` converts its input values to uppercase. Launch the code in Listing 4.37 and you will see the following output:

```
initial dataframe:
      fname  lname
fname  Jane  Smith
lname  Jane  Smith

lowercase:
      fname  lname
fname  jane  smith
lname  jane  smith

uppercase:
      fname  lname
fname  JANE  SMITH
lname  JANE  SMITH
```

USEFUL ONE-LINE COMMANDS IN PANDAS

This section contains an eclectic mix of one-line commands in `Pandas` (some of which you have already seen in this chapter) that are useful to know:

Save a dataframe to a CSV file (comma separated and without indices):

```
df.to_csv("data.csv", sep=",", index=False)
```

List the column names of a dataframe:

```
df.columns
```

Drop missing data from a dataframe:

```
df.dropna(axis=0, how='any')
```

Replace missing data in a dataframe:

```
df.replace(to_replace=None, value=None)
```

Check for NANs in a dataframe:

```
pd.isnull(object)
```

Drop a feature in a dataframe:

```
df.drop('feature_variable_name', axis=1)
```

Convert object type to float in a dataframe:

```
pd.to_numeric(df["feature_name"], errors='coerce')
```

Convert data in a dataframe to `NumPy` array:

```
df.as_matrix()
```

Display the first n rows of a dataframe:

```
df.head(n)
```

Get data by feature name in a dataframe:

```
df.loc[feature_name]
```

Apply a function to a dataframe, such as multiplying all values in the "height" column of the dataframe by 3:

```
df["height"].apply(lambda height: 3 * height)
```

OR:

```
def multiply(x):
    return x * 3
df["height"].apply(multiply)
```

Rename the fourth column of the dataframe as "height":

```
df.rename(columns = {df.columns[3]:'height'}, inplace=True)
```

Get the unique entries of the column "first" in a dataframe:

```
df["first"].unique()
```

Create a dataframe with columns `first` and `last` from an existing dataframe:

```
new_df = df[["first", "last"]]
```

Sort the data in a dataframe:

```
df.sort_values(ascending = False)
```

Filter the data column named "size" to display only values equal to 7:

```
df[df["size"] == 7]
```

Select the first row of the "height" column in a dataframe:

```
df.loc([0], ['height'])
```

WORKING WITH JSON-BASED DATA

A JSON object consists of data represented as colon-separated name/value pairs, and data objects are separated by commas. An object is specified inside curly braces {}, and an array of objects is indicated by square brackets []. Note that character-valued data elements are inside a pair of double quotes "" (but no quotes for numeric data).

Here is a simple example of a JSON object:

```
{ "fname":"Jane", "lname":"Smith", "age":33, "city":"SF" }
```

Here is a simple example of an array of JSON objects (note the outer enclosing square brackets):

```
[
{ "fname":"Jane", "lname":"Smith", "age":33, "city":"SF" },
{ "fname":"John", "lname":"Jones", "age":34, "city":"LA" },
{ "fname":"Dave", "lname":"Stone", "age":35, "city":"NY" },
]
```

Python Dictionary and JSON

The Python json library enables you to work with JSON-based data in Python.

Listing 4.38 displays the contents of dict2json.py that illustrates how to convert a Python dictionary to a JSON string.

LISTING 4.38: dict2json.py

```
import json

dict1 = {}
dict1["fname"] = "Jane"
dict1["lname"] = "Smith"
dict1["age"]   = 33
dict1["city"]  = "SF"

print("Python dictionary to JSON data:")
print("dict1:",dict1)
```

```
json1 = json.dumps(dict1, ensure_ascii=False)
print("json1:",json1)
print("")

# convert JSON string to Python dictionary:
json2 = '{"fname":"Dave", "lname":"Stone", "age":35,
"city":"NY"}'
dict2 = json.loads(json2)
print("JSON data to Python dictionary:")
print("json2:",json2)
print("dict2:",dict2)
```

Listing 4.38 invokes the `json.dumps()` function to perform the conversion from a Python dictionary to a `JSON` string. Launch the code in Listing 4.38 and you will see the following output:

```
Python dictionary to JSON data:
dict1: {'fname': 'Jane', 'lname': 'Smith', 'age': 33,
'city': 'SF'}
json1: {"fname": "Jane", "lname": "Smith", "age": 33,
"city": "SF"}

JSON data to Python dictionary:
json2: {"fname":"Dave", "lname":"Stone", "age":35,
"city":"NY"}
dict2: {'fname': 'Dave', 'lname': 'Stone', 'age': 35,
'city': 'NY'}
```

Python, Pandas, and JSON

Listing 4.39 displays the contents of `pd_python_json.py` that illustrates how to convert a Python dictionary to a `Pandas DataFrame` and then convert the dataframe to a `JSON` string.

LISTING 4.39: pd_python_json.py

```
import json
import pandas as pd

dict1 = {}
dict1["fname"] = "Jane"
dict1["lname"] = "Smith"
dict1["age"]   = 33
dict1["city"]  = "SF"

df1 = pd.DataFrame.from_dict(dict1, orient='index')
print("Pandas df1:")
print(df1)
print()

json1 = json.dumps(dict1, ensure_ascii=False)
print("Serialized to JSON1:")
print(json1)
print()
```

```
print("Dataframe to JSON2:")
json2 = df1.to_json(orient='split')
print(json2)
```

Listing 4.39 initializes a Python dictionary `dict1` with multiple attributes for a user (first name, last name, and so forth). Next, the `DataFrame df1` is created from the Python dictionary `dict1`, and its contents are displayed.

The next portion of Listing 4.39 initializes the variable `json1` by serializing the contents of `dict1`, and its contents are displayed. The last code block in Listing 4.39 initializes the variable `json2` to the result of converting the `DataFrame df1` to a `JSON` string. Launch the code in Listing 4.39 and you will see the following output:

```
dict1: {'fname': 'Jane', 'lname': 'Smith', 'age': 33,
'city': 'SF'}
Pandas df1:
           0
fname   Jane
lname  Smith
age       33
city      SF

Serialized to JSON1:
{"fname": "Jane", "lname": "Smith", "age": 33, "city":
"SF"}

Dataframe to JSON2:
{"columns":[0],"index":["fname","lname","age","city"],
"data":[["Jane"],["Smith"],[33],["SF"]]}
json1: {"fname": "Jane", "lname": "Smith", "age": 33,
"city": "SF"}
```

PANDAS AND REGULAR EXPRESSIONS (OPTIONAL)

This section is marked "optional" because the code snippets require an understanding of regular expressions. If you are not ready to learn about regular expressions, you can skip this section with no loss of continuity. Alternatively, you can read the appropriate appendix (if necessary) regarding regular expressions and then you can return to this portion of the chapter.

Listing 4.40 displays the contents `pandas_regexs.py` that illustrates how to extract data from a `Pandas DataFrame` using regular expressions.

LISTING 4.40: pandas_regexs.py

```
import pandas as pd

schedule = ["Monday: Prepare lunch at 12:30pm for VIPs",
            "Tuesday: Yoga class from 10:00am to 11:00am",
            "Wednesday: PTA meeting at library at 3pm",
            "Thursday: Happy hour at 5:45 at Julie's house.",
            "Friday: Prepare pizza dough for lunch at 12:30pm.",
```

```
            "Saturday: Early shopping for the week at 8:30am.",
            "Sunday: Neighborhood bbq block party at 2:00pm."]

# create a Pandas dataframe:
df = pd.DataFrame(schedule, columns = ['dow_of_week'])

# convert to lowercase:
df = df.applymap(lambda s:s.lower() if type(s) == str else s)
print("df:")
print(df)
print()

# character count for each string in df['dow_of_week']:
print("string lengths:")
print(df['dow_of_week'].str.len())
print()

# the number of tokens for each string in df['dow_of_week']
print("number of tokens in each string in df['dow_of_week']:")
print(df['dow_of_week'].str.split().str.len())
print()

# the number of occurrences of digits:
print("number of digits:")
print(df['dow_of_week'].str.count(r'\d'))
print()

# display all occurrences of digits:
print("show all digits:")
print(df['dow_of_week'].str.findall(r'\d'))
print()

# display hour and minute values:
print("display (hour, minute) pairs:")
print(df['dow_of_week'].str.findall(r'(\d?\d):(\d\d)'))
print()

# create new columns from hour:minute value:
print("hour and minute columns:")
print(df['dow_of_week'].str.extract(r'(\d?\d):(\d\d)'))
print()
```

Listing 4.40 initializes the variable `schedule` with a set of strings, each of which specifies a daily to-do item for an entire week. The format for each to-do item is of the form `day:task`, where day is a day of the week and task is a string that specifies what needs to be done on that particular day. Next, the `DataFrame df1` is initialized with the contents of `schedule`, followed by an example of defining a lambda expression that converts string-based values to lower case, as shown here:

```
df = df.applymap(lambda s:s.lower() if type(s) == str else s)
```

The preceding code snippet is useful because you do not need to specify individual columns of a dataframe: the code ignores any non-string values (such as integers and floating point values).

The next pair of code blocks involve various operations using the methods `applymap()`, `split()`, and `len()` that you have seen in previous examples. The next code block displays the number of digits in each to-do item by means of the regular expression in the following code snippet:

```
print(df['dow_of_week'].str.count(r'\d'))
```

The next code block displays the actual digits (instead of the number of digits) in each to-do item by means of the regular expression in the following code snippet:

```
print(df['dow_of_week'].str.findall(r'\d'))
```

The final code block displays the strings of the form `hour:minutes` by means of the regular expression in the following code snippet:

```
print(df['dow_of_week'].str.findall(r'(\d?\d):(\d\d)'))
```

As mentioned in the beginning of this section, you can learn more about regular expressions by reading the appendix of this book. Launch the code in Listing 4.40 and you will see the following output:

```
=> df:
                                          dow_of_week
0          monday: prepare lunch at 12:30pm for vips
1        tuesday: yoga class from 10:00am to 11:00am
2          wednesday: pta meeting at library at 3pm
3     thursday: happy hour at 5:45 at julie's house.
4  friday: prepare pizza dough for lunch at 12:30pm.
5   saturday: early shopping for the week at 8:30am.
6    sunday: neighborhood bbq block party at 2:00pm.

=> string lengths:
0    41
1    43
2    40
3    46
4    49
5    48
6    47
Name: dow_of_week, dtype: int64

=> number of tokens in each string in df['dow_of_week']:
0    7
1    7
2    7
3    8
4    8
```

```
5    8
6    7
Name: dow_of_week, dtype: int64

=> number of digits:
0    4
1    8
2    1
3    3
4    4
5    3
6    3
Name: dow_of_week, dtype: int64

=> show all digits:
0                [1, 2, 3, 0]
1    [1, 0, 0, 0, 1, 1, 0, 0]
2                         [3]
3                   [5, 4, 5]
4                [1, 2, 3, 0]
5                   [8, 3, 0]
6                   [2, 0, 0]
Name: dow_of_week, dtype: object

=> display (hour, minute) pairs:
0              [(12, 30)]
1    [(10, 00), (11, 00)]
2                      []
3               [(5, 45)]
4              [(12, 30)]
5               [(8, 30)]
6               [(2, 00)]
Name: dow_of_week, dtype: object

=> hour and minute columns:

     0    1
0   12   30
1   10   00
2  NaN  NaN
3    5   45
4   12   30
5    8   30
6    2   00
```

WHAT IS TEXTHERO?

`Texthero` is a Python-based open-source toolkit that functions as a layer of abstraction over `Pandas`, and its home page is here:

https://github.com/jbesomi/texthero.

`Texthero` leverages very useful Python libraries for NLP, such as Gensim, NLTK, SpaCy, and Sklearn. Moreover, `texthero` supports the following functionality:

- NER and topic modeling (for NLP)
- TF-IDF, term frequency, and word-embeddings (for NLP)
- DBSCAN, Hierarchical, k-Means, and Meanshift algorithms
- Various types of text visualization

Open a command shell and install `texthero` with the following command:

```
pip3 install texthero
```

`Texthero` supports various other algorithms, including dimensionality reduction algorithms for machine learning. Navigate to the following link for documentation and other information about `texthero`:

https://texthero.org/docs/getting-started.

SUMMARY

This chapter introduced you to `Pandas` for creating labeled dataframes and displaying metadata of `Pandas Dataframe`s. Then you learned how to create `Pandas Dataframe`s from various sources of data, such as random numbers and hard-coded data values.

You also learned how to read Excel spreadsheets and perform numeric calculations on that data, such as the minimum, mean, and maximum values in numeric columns. Then you saw how to create `Pandas Dataframe`s from data stored in CSV files. In addition, you learned how to generate a scatterplot from data in a `Pandas Dataframe`.

Then you got a brief introduction to `JSON`, along with an example of converting a Python dictionary to `JSON`-based data (and vice versa). Finally, you learned about `texthero`, which is an open-source, Python-based toolkit that is a layer of abstraction over `Pandas`.

CHAPTER 5

INTRODUCTION TO PROBABILITY AND STATISTICS

This chapter introduces you to concepts in probability as well as a wide assortment of statistical terms and algorithms.

The first section of this chapter starts with a discussion of probability, how to calculate the expected value of a set of numbers (with associated probabilities), the concept of a random variable (discrete and continuous), and a short list of some well-known probability distributions.

The second section of this chapter introduces basic statistical concepts, such as *mean*, *median*, *mode*, *variance*, and *standard deviation*, along with simple examples that illustrate how to calculate these terms. You will also learn about the terms `RSS`, `TSS`, `R^2`, and `F1` score.

The third section of this chapter introduces Gini impurity, entropy, perplexity, cross-entropy, and KL divergence. You will also learn about skewness and kurtosis.

The fourth section explains covariance and correlation matrices and how to calculate eigenvalues and eigenvectors.

The fifth section explains PCA (Principal Component Analysis), which is a well-known dimensionality reduction technique. The final section introduces you to Bayes's Theorem.

WHAT IS A PROBABILITY?

If you have ever performed a science experiment in one of your classes, you might remember that measurements have some uncertainty. In general, we assume that there is a correct value, and we endeavor to find the best estimate of that value.

When we work with an event that can have multiple outcomes, we try to define the probability of an outcome as the chance that it will occur, which is calculated as follows:

```
p(outcome)  = # of times outcome occurs/(total number of outcomes)
```

For example, in the case of a single balanced coin, the probability of tossing a head H equals the probability of tossing a tail T:

```
p(H) = 1/2 = p(T)
```

The set of probabilities associated with the outcomes {H, T} is shown in the set `P`:

```
P = {1/2, 1/2}
```

Some experiments involve replacement while others involve non-replacement. For example, suppose that an urn contains 10 red balls and 10 green balls. What is the probability that a randomly selected ball is red? The answer is 10/(10 + 10) = 1/2. What is the probability that the second ball is also red?

There are two scenarios with two different answers. If each ball is selected with replacement, then each ball is returned to the urn after selection, which means that the urn always contains 10 red balls and 10 green balls. In this case, the answer is 1/2 * 1/2 = 1/4. In fact, the probability of any event is independent of all previous events.

On the other hand, if balls are selected without replacement, then the answer is 10/20 * 9/19. As you undoubtedly know, card games are also examples of selecting cards without replacement.

One other concept is called *conditional probability*, which refers to the likelihood of the occurrence of event `E1` given that event `E2` has occurred. A simple example is the following statement:

```
"If it rains (E2), then I will carry an umbrella (E1)."
```

Calculating the Expected Value

Consider the following scenario involving a well-balanced coin: whenever a head appears, you earn $1 and whenever a tail appears, you earn $1 dollar. If you toss the coin 100 times, how much money do you expect to earn? Since you will earn $1 regardless of the outcome, the expected value (in fact, the guaranteed value) is $100.

Now consider this scenario: whenever a head appears, you earn $1 and whenever a tail appears, you earn 0 dollars. If you toss the coin 100 times, how much money do you expect to earn? You probably determined the value 50 (which is the correct answer) by making a quick mental calculation. The more formal derivation of the value of E (the expected earning) is here:

```
E = 100 *[1 * 0.5 + 0 * 0.5] = 100 * 0.5 = 50
```

The quantity **1** * 0.5 + **0** * 0.5 is the amount of money you expected to earn during each coin toss (half the time you earn $1 and half the time you earn 0 dollars), and multiplying this number by 100 is the expected earning after 100 coin tosses. Also note that you might never earn $50: the actual amount that you earn can be any integer between 1 and 100 inclusive.

As another example, suppose that you earn $3 whenever a head appears, and you *lose* $1.50 dollars whenever a tail appears. Then the expected earning `E` after 100 coin tosses is shown here:

```
E = 100 *[3 * 0.5 - 1.5 * 0.5] = 100 * 1.5 = 150
```

We can generalize the preceding calculations as follows. Let P = `{p1,…,pn}` be a probability distribution, which means that the values in P are nonnegative and their sum equals 1. In addition, let R = `{R1,…,Rn}` be a set of rewards, where reward Ri is received with probability `pi`. Then the expected value E after N trials is shown here:

```
E = N * [SUM pi*Ri]
```

In the case of a single balanced die, we have the following probabilities:

```
p(1) = 1/6
p(2) = 1/6
p(3) = 1/6
p(4) = 1/6
p(5) = 1/6
p(6) = 1/6
P = { 1/6, 1/6, 1/6, 1/6, 1/6, 1/6}
```

As a simple example, suppose that the earnings are {3, 0, –1, 2, 4, –1} when the values 1,2,3,4,5,6, respectively, appear when tossing the single die. Then after 100 trials our expected earnings are calculated as follows:

```
E = 100 * [3 + 0 + -1 + 2 + 4 + -1]/6 = 100 * 3/6 = 50
```

In the case of two balanced dice, we have the following probabilities of rolling 2,3, … , or 12:

```
p(2) = 1/36
p(3) = 2/36
...
p(12) = 1/36
P = {1/36,2/36,3/36,4/36,5/36,6/36,5/36,4/36,3/36,2/36,1/36}
```

RANDOM VARIABLES

A random variable is a variable that can have multiple values and where each value has an associated probability of occurrence. For example, if we let `X` be a random variable whose values are the outcomes of tossing a well-balanced

die, then the values of X are the numbers in the set {1,2,3,4,5,6}. Moreover, each of those values can occur with equal probability (which is 1/6).

In the case of two well-balanced dice, let X be a random variable whose values can be any of the numbers in the set {2,3,4, . . . , 12}. Then the associated probabilities for the different values for X are listed in the previous section.

Discrete versus Continuous Random Variables

The preceding section contains examples of *discrete* random variables because the list of possible values is either finite or countably infinite (such as the set of integers). As an aside, the set of rational numbers is also countably infinite, but the set of irrational numbers and also the set of real numbers are both uncountably infinite (proofs are available online). As pointed out earlier, the associated set of probabilities must form a probability distribution, which means that the probability values are nonnegative and their sum equals 1.

A *continuous* random variable is a variable whose values can be *any* number in an interval, which can be an uncountably infinite number of values. For example, the amount of time required to perform a task is represented by a continuous random variable.

A continuous random variable also has a probability distribution that is represented as a continuous function. The constraint for such a variable is that the area under the curve (which is sometimes calculated via a mathematical integral) equals 1.

Well-Known Probability Distributions

There are many probability distributions, and some of the well-known probability distributions are listed here:

- Gaussian distribution
- Poisson distribution
- Chi-squared distribution
- Binomial distribution

The Gaussian distribution is named after Karl F. Gauss, and it is sometimes called the normal distribution or the Bell curve. The Gaussian distribution is symmetric: the shape of the curve on the left of the mean is identical to the shape of the curve on the right side of the mean. As an example, the distribution of IQ scores follows a curve that is similar to a Gaussian distribution.

On the other hand, the frequency of traffic at a given point in a road follows a Poisson distribution (which is not symmetric). Interestingly, if you count the number of people who go to a public pool based on five-degree (Fahrenheit) increments of the temperature, followed by five-degree decrements in temperature, that set of numbers follows a Poisson distribution.

Perform an Internet search for each of the bullet items in the preceding list and you will find numerous articles that contain images and technical details about these (and other) probability distributions.

This concludes the brief introduction to probability, and the next section delves into the concepts of mean, median, mode, and standard deviation.

FUNDAMENTAL CONCEPTS IN STATISTICS

This section contains several subsections that discuss the mean, median, mode, variance, and standard deviation. Feel free to skim (or skip) this section if you are already familiar with these concepts. As a start point, let's suppose that we have a set of numbers X = {x1, ..., xn} that can be positive, negative, integer-valued, or decimal values.

The Mean

The *mean* of the numbers in the set X is the average of the values. For example, if the set X consists of {-10,35,75,100}, then the mean equals (–10 + 35 + 75 + 100)/4 = 50. If the set X consists of {2,2,2,2}, then the mean equals (2 + 2 + 2 + 2)/4 = 2. As you can see, the mean value is not necessarily one of the values in the set.

Keep in mind that the mean is sensitive to outliers. For example, the mean of the set of numbers {1,2,3,4} is 2.5, whereas the mean of the set of number {1,2,3,4,1000} is 202. Since the formulas for the variance and standard deviation involve the mean of a set of numbers, both of these terms are also more sensitive to outliers.

The Median

The *median* of the numbers (sorted in increasing or decreasing order) in the set X is the middle value in the set of values, which means that half the numbers in the set are less than the median and half the numbers in the set are greater than the median. For example, if the set X consists of {-10,35,75,100}, then the *median* equals 55 because 55 is the average of the two numbers 35 and 75. As you can see, half the numbers are less than 55 and half the numbers are greater than 55. If the set X consists of {2,2,2,2}, then the *median* equals 2.

By contrast, the median is much less sensitive to outliers than the mean. For example, the median of the set of numbers {1,2,3,4} is 2.5, and the median of the set of numbers {1,2,3,4,1000} is 3.

The Mode

The *mode* of the numbers (sorted in increasing or decreasing order) in the set X is the most frequently occurring value, which means that there can be more than one such value. If the set X consists of {2,2,2,2}, then the *mode* equals 2.

If X is the set of numbers {2,4,5,5,6,8}, then the number 5 occurs twice and the other numbers occur only once, so the *mode* equals 5.

If X is the set of numbers {2,2,4,5,5,6,8}, then the numbers 2 and 5 occur twice and the other numbers occur only once, so the *mode* equals 2 and

5. A set that has two modes is called bimodal, and a set that has more than two modes is called multimodal.

One other scenario involves sets that have numbers with the same frequency and they are all different. In this case, the mode does not provide meaningful information, and one alternative is to partition the numbers into subsets and then select the largest subset. For example, if set X has the values {1,2,15,16,17,25,35,50}, we can partition the set into subsets whose elements are in ranges that are multiples of ten, which results in the subsets {1,2}, {15,16,17}, {25}, {35}, and {50}. The largest subset is {15,16,17}, so we could select the number 16 as the mode.

As another example, if set X has the values {-10,35,75,100}, then partitioning this set does not provide any additional information, so it's probably better to work with either the mean or the median.

The Variance and Standard Deviation

The *variance* is the sum of the squares of the difference between the numbers in X and the mean mu of the set X, divided by the number of value in X, as shown here:

```
variance = [SUM (xi - mu)**2 ] / n
```

For example, if the set X consists of {-10,35,75,100}, then the *mean* equals (–10 + 35 + 75 + 100)/4 = 50, and the variance is computed as follows:

```
variance = [(-10-50)**2 + (35-50)**2 + (75-50)**2 + (100-
50)**2]/4
         = [60**2 + 15**2 + 25**2 + 50**2]/4
         = [3600 + 225 + 625 + 2500]/4
         = 6950/4 = 1,737
```

The standard deviation std is the square root of the variance:

```
std = sqrt(1737) = 41.677
```

If the set X consists of {2,2,2,2}, then the *mean* equals (2 + 2 + 2 + 2)/4 = 2, and the variance is computed as follows:

```
variance = [(2-2)**2 + (2-2)**2 + (2-2)**2 + (2-2)**2]/4
         = [0**2 + 0**2 + 0**2 + 0**2]/4
         = 0
```

The standard deviation std is the square root of the variance:

```
std = sqrt(0) = 0
```

Population, Sample, and Population Variance

The population specifically refers to the entire set of entities in a given group, such as the population of a country, the people over 65 in the United States, or the number of first-year students in a university.

However, in many cases statistical quantities are calculated on samples instead of an entire population. Thus, a sample is a (much smaller) subset of the given population. See the central limit theorem regarding the distribution of the mean of a sample of a population (which need not be a population with a Gaussian distribution).

If you want to learn about techniques for sampling data, here is a list of three different techniques that you can investigate:

- Stratified sampling
- Cluster sampling
- Quota sampling

One other important point: the population variance is calculated by multiplying the sample variance by `n/(n-1)`, as shown here:

```
population variance = [n/(n-1)]*variance
```

Chebyshev's Inequality

Chebyshev's inequality provides a very simple way to determine the minimum percentage of data that lies within k standard deviations. Specifically, this inequality states that for any positive integer k greater than 1, the amount of data in a sample that lies within k standard deviations is at least 1 – 1/k**2. For example, if k = 2, then at least 1 – 1/2**2 = 3/4 of the data must lie within 2 standard deviations.

The interesting part of this inequality is that it's been mathematically proven to be true; that is, it's not an empirical or heuristic-based result. An extensive description regarding Chebyshev's inequality (including some advanced mathematical explanations) is here:

https://en.wikipedia.org/wiki/Chebyshev%27s_inequality.

What Is a p-value?

The null hypothesis states that there is no correlation between a dependent variable (such as `y`) and an independent variable (such as `x`). The `p-value` is used to reject the null hypothesis if the `p-value` is small enough (< 0.005), which indicates a higher significance. The threshold value for `p` is typically 1% or 5%.

There is no straightforward formula for calculating p-values, which are values that are always between 0 and 1. In fact, `p-values` are statistical quantities to evaluate the null hypothesis, and they are calculated by means of `p-value` tables or via spreadsheet/statistical software.

THE MOMENTS OF A FUNCTION (OPTIONAL)

The previous sections describe several statistical terms that are sufficient for the material in this book. However, several of those terms can be viewed from the perspective of different moments of a function.

In brief, the moments of a function are measures that provide information regarding the shape of the graph of a function. In the case of a probability distribution, the first four moments are defined as follows:

- The mean is the first central moment
- The variance is the second central moment
- The skewness (discussed later) is the third central moment
- The kurtosis (discussed later) is the fourth central moment

More detailed information (including the relevant integrals) regarding moments of a function is available here:

https://en.wikipedia.org/wiki/Moment_(mathematics)#Variance.

What Is Skewness?

Skewness is a measure of the asymmetry of a probability distribution. A Gaussian distribution is symmetric, which means that its skew value is zero (it's not exactly zero, but close enough for our purposes). In addition, the skewness of a distribution is the *third* moment of the distribution.

A distribution can be skewed on the left side or on the right side. A *left-sided* skew means that the long tail is on the left side of the curve, with the following relationships:

```
mean < median < mode
```

A *right-sided* skew means that the long tail is on the right side of the curve, with the following relationships (compare with the left-sided skew):

```
mode < median < mean
```

If need be, you can transform skewed data to a normally distributed dataset using one of the following techniques (which depends on the specific use-case):

Exponential transform
Log transform
Power transform

Perform an online search for more information regarding the preceding transforms and when to use each of these transforms.

What Is Kurtosis?

Kurtosis is related to the skewness of a probability distribution, in the sense that both of them assess the asymmetry of a probability distribution. The kurtosis of a distribution is a scaled version of the *fourth* moment of the distribution, whereas its skewness is the *third* moment of the distribution. Note that the kurtosis of a univariate distribution equals 3.

If you are interested in learning about additional kurtosis-related concepts, you can perform an online search for information regarding mesokurtic, leptokurtic, and platykurtic types of so-called "excess kurtosis".

DATA AND STATISTICS

This section contains various subsections that briefly discuss some of the challenges and obstacles that you might encounter when working with datasets. This section and subsequent sections introduce you to the following concepts:

- Correlation versus causation
- The bias-variance tradeoff
- Types of bias
- The central limit theorem
- Statistical inferences

First, keep in mind that statistics typically involves data *samples*, which are subsets of observations of a population. The goal is to find well-balanced samples that provide a good representation of the entire population.

Although this goal can be very difficult to achieve, it's also possible to achieve highly accurate results with a very small sample size. For example, the Harris poll in the United States has been used for decades to analyze political trends. This poll computes percentages that indicate the favorability rating of political candidates, and it's usually within 3.5% of the correct percentage values. What's remarkable about the Harris poll is that its sample size is a mere 4,000 people that are from the U.S. population that is greater than 325,000,000 people.

Another aspect to consider is that each sample has a mean and variance, which do not necessarily equal the mean and variance of the actual population. However, the expected value of the sample mean and variance equal the mean and variance, respectively, of the population.

The Central Limit Theorem

Samples of a population have an interesting property. Suppose that you take a set of samples `{S1, S3, …, Sn}` of a population and you calculate the mean of those samples, which is `{m1, m2, …, mn}`. The Central Limit Theorem is a remarkable result: given a set of samples of a population and the mean value of those samples, the distribution of the mean values can be approximated by a Gaussian distribution. Moreover, as the number of samples increases, the approximation becomes more accurate.

Correlation versus Causation

In general, datasets have some features (columns) that are more significant in terms of their set of values, and some features only provide additional information that does not contribute to potential trends in the dataset. For example,

the passenger names in the list of passengers on the Titanic are unlikely to affect the survival rate of those passengers, whereas the gender of the passengers is likely to be an important factor.

In addition, a pair of significant features may also be "closely coupled" in terms of their values. For example, a real estate dataset for a set of houses will contain the number of bedrooms and the number of bathrooms for each house in the dataset. As you know, these values tend to increase together and also decrease together. Have you ever seen a house that has ten bedrooms and one bathroom, or a house that has ten bathrooms and one bedroom? If you did find such a house, would you purchase that house as your primary residence?

The extent to which the values of two features change is called their correlation, which is a number between –1 and 1. Two "perfectly" correlated features have a correlation of 1, and two features that are not correlated have a correlation of 0. In addition, if the values of one feature decrease when the values of another feature increase, and vice versa, then their correlation is closer to –1 (and might also equal –1).

On the other hand, causation between two features means that the values of one feature can be used to calculate the values of the second feature (within some margin of error).

Keep in mind this fundamental point about machine learning models: they can provide correlation, but they cannot provide causation.

Statistical Inferences

Statistical thinking relates to processes and statistics, whereas statistical inference refers to the process by which inferences are made regarding a population. Those inferences are based on statistics that are derived from samples of the population. The validity and reliability of those inferences depend on random sampling in order to reduce bias. There are various metrics that you can calculate to help you assess the validity of a model that has been trained on a particular dataset.

STATISTICAL TERMS – RSS, TSS, R^2, AND F1 SCORE

Statistics is extremely important in machine learning, so it's not surprising that many concepts are common to both fields. Machine learning relies on a number of statistical quantities in order to assess the validity of a model, some of which are listed here:

- `RSS`
- `TSS`
- `R^2`

The term RSS is the "residual sum of squares" and the term TSS is the "total sum of squares". Moreover, these terms are used in regression models.

As a starting point so we can simplify the explanation of the preceding terms, suppose that we have a set of points `{(x1,y1), . . . , (xn,yn)}` in the Euclidean plane. In addition, let's define the following quantities:

- `(x,y)` is any point in the dataset
- `y` is the y-coordinate of a point in the dataset
- `y_` is the mean of the y-values of the points in the dataset
- `y_hat` is the y-coordinate of a point on a best-fitting line

Just to be clear, `(x,y)` is a point in the *dataset*, whereas `(x,y_hat)` is the corresponding point that lies on the *best fitting line*. With these definitions in mind, the definitions of `RSS`, `TSS`, and `R^2` are listed here (n equals the number of points in the dataset):

```
RSS = Iy - y_hat)**2/n
TSS = (y - y_bar)**2/n
R^2 = 1 - RSS/TSS
```

We also have the following inequalities involving `RSS`, `TSS`, and `R^2`:

```
0 <= RSS
RSS <= TSS
0 <= RSS/TSS <= 1
0 <= 1 - RSS/TSS <= 1
0 <= R^2 <= 1
```

When RSS is close to 0, then RSS/TSS is also close to zero, which means that R^2 is close to 1. Conversely, when RSS is close to TSS, then RSS/TSS is close to 1, and R^2 is close to 0. In general, a larger R^2 is preferred (i.e., the model is closer to the data points), but a lower value of R^2 is not necessarily a bad score.

What Is an F1 Score?

In machine learning, an F1 score is for models that are evaluated on a feature that contains categorical data, and the p-value is useful for machine learning in general. An F1 score is a measure of the accuracy of a test, and it's defined as the harmonic mean of precision and recall. Here are the relevant formulas, where `p` is the precision and `r` is the recall:

```
p = (# of correct positive results)/(# of all positive results)
r = (# of correct positive results)/(# of all relevant samples)

F1-score  = 1/[((1/r) + (1/p))/2]
          = 2*[p*r]/[p+r]
```

The best value of an F1 score is 1 and the worst value is 0. Keep in mind that an F1 score is for categorical classification problems, whereas the `R^2` value is typically for regression tasks (such as linear regression).

GINI IMPURITY, ENTROPY, AND PERPLEXITY

These concepts are useful for assessing the quality of a machine learning model, and the latter pair are useful for dimensionality reduction algorithms.

Before we discuss the details of Gini impurity, suppose that P is a set of nonnegative numbers {p1, p2, …, pn} such that the sum of all the numbers in the set P equals 1. Under these two assumptions, the values in the set P comprise a probability distribution, which we can represent with the letter p.

Now suppose that the set K contains a total of M elements, with k1 elements from class S1, k2 elements from class S2, . . . , and kn elements from class Sn. Compute the fractional representation for each class as follows:

```
p1 = k1/M, p2 = k2/M, . . ., pn = kn/M
```

As you can surmise, the values in the set {p1, p2, …, pn} form a probability distribution. We're going to use the preceding values in the following subsections.

What Is Gini Impurity?

The Gini impurity is defined as follows, where {p1,p2,…,pn} is a probability distribution:

```
Gini = 1 - [p1*p1 + p2*p2 + . . . + pn*pn]
     = 1 - SUM pi*pi (for all i, where 1<=i<=n)
```

Since each pi is between 0 and 1, then pi*pi <= pi, which means that:

```
1 = p1 + p2 + . . . + pn
  >= p1*p1 + p2*p2 + . . . + pn*pn
   = Gini impurity
```

Since the Gini impurity is the sum of the squared values of a set of probabilities, the Gini impurity cannot be negative. Hence, we have derived the following result:

```
0 <= Gini impurity <= 1
```

What Is Entropy?

A formal definition: *entropy* is a measure of the expected ("average") number of bits required to encode the outcome of a random variable. The calculation for the entropy H (the letter E is reserved for Einstein's formula) is defined via the following formula:

```
H = (-1)*[p1*log p1 + p2 * log p2 + . . . + pn * log pn]
  = (-1)* SUM [pi * log(pi)] (for all i, where 1<=i<=n)
```

Calculating Gini Impurity and Entropy Values

For our first example, suppose that we have two classes A and B and a cluster of 10 elements with 8 elements from class A and 2 elements from class B. Therefore, `p1` and `p2` are 8/10 and 2/10, respectively. We can compute the Gini score as follows:

```
Gini = 1 – [p1*p1 + p2*p2]
     = 1 – [64/100 + 04/100]
     = 1 - 68/100
     = 32/100
     = 0.32
```

We can also calculate the entropy for this example as follows:

```
Entropy = (-1)*[p1 * log p1 + p2 * log p2]
        = (-1)*[0.8 * log 0.8 + 0.2 * log 0.2]
        = (-1)*[0.8 * (-0.322) + 0.2 * (-2.322)]
        = 0.8 * 0.322 + 0.2 * 2.322
        = 0.7220
```

For our second example, suppose that we have three classes A, B, C and a cluster of 10 elements with 5 elements from class A, 3 elements from class B, and 2 elements from class C. Therefore `p1`, `p2`, and `p3` are 5/10, 3/10, and 2/10, respectively. We can compute the Gini score as follows:

```
Gini = 1 – [p1*p1 + p2*p2 + p3*p3]
     = 1 – [25/100 + 9/100 + 04/100]
     = 1 - 38/100
     = 62/100
     = 0.62
```

We can also calculate the entropy for this example as follows:

```
Entropy = (-1)*[p1 * log p1 + p2 * log p2]
        = (-1)*[0.5*log0.5 + 0.3*log0.3 + 0.2*log0.2]
        = (-1)*[-0.5 + 0.3*(-1.737) + 0.2*(-2.322)]
        = 0.5 + 0.3*1.737 + 0.2*2.322
        = 1.4855
```

In both examples the Gini impurity is between 0 and 1. However, while the entropy is between 0 and 1 in the first example, it's greater than 1 in the second example (which was the rationale for showing you two examples).

Keep in mind that a set whose elements belong to the same class has Gini impurity equal to 0 and also its entropy equal to 0. For example, if a set has 10 elements and all of them belong to class S1, then:

```
Gini = 1 – SUM pi*pi
     = 1 - p1*p1
     = 1 – (10/10)*(10/10)
     = 1 – 1 = 0
```

```
Entropy = (-1)*SUM pi*log pi
        = (-1) * p1*log p1
        = (-1) * (10/10) * log(10/10)
        = (-1)*1*0  = 0
```

Multidimensional Gini Index

The Gini index is a one-dimensional index that works well because the value is uniquely defined. However, when working with multiple factors, we need a multidimensional index. Unfortunately, the multidimensional Gini index (MGI) is not uniquely defined. While there have been various attempts to define an MGI that has unique values, they tend to be nonintuitive and mathematically much more complex. More information about MGI is here:

https://link.springer.com/chapter/10.1007/978-981-13-1727-9_5.

What Is Perplexity?

Suppose that `q` and `p` are two probability distributions, and `{x1, x2, …, xN}` is a set of sample values that is drawn from a model whose probability distribution is `p`. In addition, suppose that `b` is a positive integer (it's usually equal to 2). Now define the variable `S` as the following sum (logarithms are in base `b` not 10):

```
S = (-1/N) * [log q(x1) + log q(x2) + . . . + log q(xN)]
  = (-1/N) * SUM log q(xi)
```

The formula for the perplexity PERP of the model `q` is `b` raised to the power `S`, as shown here:

```
PERP = b^S
```

If you compare the formula for entropy with the formula for `S`, you can see that the formulas are similar, so the perplexity of a model is somewhat related to the entropy of a model.

CROSS-ENTROPY AND KL DIVERGENCE

Cross-entropy is useful for understanding machine learning algorithms and frameworks such as TensorFlow, which supports multiple APIs that involve cross-entropy. KL divergence is relevant in machine learning, deep learning, and reinforcement learning.

As an interesting example, consider the credit assignment problem, which involves assigning credit to different elements or steps in a sequence. For example, suppose that users arrive at a Web page by clicking on a previous page, which was also reached by clicking on yet another Web page. Then on the final Web page users click on an ad. How much credit is given to the first and second Web pages for the selected ad? You might be surprised to discover that one solution to this problem involves KL Divergence.

What Is Cross-Entropy?

The following formulas for logarithms are presented here because they are useful for the derivation of cross entropy in this section:

- `log (a * b) = log a + log b`
- `log (a / b) = log a - log b`
- `log (1 / b) = (-1) * log b`

In a previous section, you learned that for a probability distribution `P` with values `{p1,p2,… pn}`, its entropy is `H` defined as follows:

```
H(P) = (-1)*SUM pi*log(pi)
```

Now let's introduce another probability distribution `Q` whose values are `{q1,q2,…, qn}`, which means that the entropy `H` of `Q` is defined as follows:

```
H(Q) = (-1)*SUM qi*log(qi)
```

Now we can define the cross-entropy CE of Q and P as follows (notice the `log qi` and `log pi` terms and recall the formulas for logarithms in the previous section):

```
CE(Q,P) = SUM (pi*log qi) - SUM (pi*log pi)
        = SUM (pi*log qi - pi*log pi)
        = SUM pi*(log qi - log pi)
        = SUM pi*(log qi/pi)
```

What Is KL Divergence?

Now that entropy and cross-entropy have been discussed, we can easily define the KL divergence of the probability distributions Q and P as follows:

```
KL(P||Q) = CE(P,Q) - H(P)
```

The definitions of entropy `H`, cross-entropy CE, and KL divergence in this chapter involve discrete probability distributions `P` and `Q`. However, these concepts have counterparts in continuous probability density functions. The mathematics involves the concept of a Lebesgue measure on Borel sets (which is beyond the scope of this book) that are described here:

https://en.wikipedia.org/wiki/Lebesgue_measure
https://en.wikipedia.org/wiki/Borel_set

In addition to KL divergence, there is also JS divergence, also called Jenson-Shannon divergence, which was developed by Johan Jensen and Claude Shannon (who defined the formula for entropy). JS divergence is based on KL divergence, and it has some differences: JS divergence is symmetric and

a true metric, whereas KL divergence is neither (as noted in Chapter 4). More information regarding JS divergence is available here:

https://en.wikipedia.org/wiki/Jensen–Shannon_divergence.

What's Their Purpose?

The Gini impurity is often used to obtain a measure of the homogeneity of a set of elements in a decision tree. The entropy of that set is an alternative to its Gini impurity, and you will see both of these quantities used in machine learning models.

The perplexity value in NLP is one way to evaluate language models, which are probability distributions over sentences or texts. This value provides an estimate for the encoding size of a set of sentences.

Cross-entropy is used in various methods in the TensorFlow framework, and the KL divergence is used in various algorithms, such as the dimensionality reduction algorithm t-SNE. For more information about any of these terms, perform an online search and you will find numerous online tutorials that provide more detailed information.

COVARIANCE AND CORRELATION MATRICES

This section explains two important matrices: the covariance matrix and the correlation matrix. Although these are relevant for PCA (principal component analysis) that is discussed later in this chapter, these matrices are not specific to PCA, which is the rationale for discussing them in a separate section. If you are familiar with these matrices, feel free to skim through this section.

The Covariance Matrix

As a reminder, the statistical quantity called the variance of a random variable `X` is defined as follows:

```
variance(x) = [SUM (x - xbar)*(x-xbar)]/n
```

A covariance matrix `C` is an nxn matrix whose values on the main diagonal are the variance of the variables `X1`, `X2`, . . . , `Xn`. The other values of `C` are the covariance values of each pair of variables `Xi` and `Xj`.

The formula for the covariance of the variables `X` and `Y` is a generalization of the variance of a variable, and the formula is shown here:

```
covariance(X, Y) = [SUM (x - xbar)*(y-ybar)]/n
```

Notice that you can reverse the order of the product of terms (multiplication is commutative), and therefore the covariance matrix `C` is a symmetric matrix:

```
covariance(X, Y) = covariance(Y,X)
```

Suppose that a CSV file contains four numeric features, all of which have been scaled appropriately, and let's call them `x1`, `x2`, `x3`, and `x4`. Then the covariance matrix C is a 4x4 square matrix that is defined with the following entries (pretend that there are outer brackets on the left side and the right side to indicate a matrix):

```
cov(x1,x1) cov(x1,x2) cov(x1,x3) cov(x1,x4)
cov(x2,x1) cov(x2,x2) cov(x2,x3) cov(x2,x4)
cov(x3,x1) cov(x3,x2) cov(x3,x3) cov(x3,x4)
cov(x4,x1) cov(x4,x2) cov(x4,x3) cov(x4,x4)
```

Note that the following is true for the diagonal entries in the preceding covariance matrix `C`:

```
var(x1,x1) = cov(x1,x1)
var(x2,x2) = cov(x2,x2)
var(x3,x3) = cov(x3,x3)
var(x4,x4) = cov(x4,x4)
```

In addition, `C` is a symmetric matrix, which is to say that the transpose of matrix `C` (rows become columns and columns become rows) is identical to the matrix `C`. The latter is true because (as you saw in the previous section) `cov(x,y) = cov(y,x)` for any feature `x` and any feature `y`.

Covariance Matrix: An Example

Suppose we have the two-column matrix A defined as follows:

```
      x  y
A = | 1  1 | <= 6x2 matrix
    | 2  1 |
    | 3  2 |
    | 4  2 |
    | 5  3 |
    | 6  3 |
```

The mean `x_bar` of column `x` is `(1+2+3+4+5+6)/6 = 3.5`, and the mean `y_bar` of column y is `(1+1+2+2+3+3)/6 = 2`. Now subtract `x_bar` from column `x` and subtract `y_bar` from column `y` and we get matrix `B`, as shown here:

```
B = | -2.5 -1 | <= 6x2 matrix
    | -1.5 -1 |
    | -0.5  0 |
    |  0.5  0 |
    |  1.5  1 |
    |  2.5  1 |
```

Let `Bt` indicate the transpose of the matrix `B` (i.e., switch columns with rows and rows with columns), which means that `Bt` is a 2 x 6 matrix, as shown here:

```
Bt = |-2.5 -1.5 -0.5 0.5, 1.5, 2.5|
     |-1   -1    0   0    1    1  |
```

The covariance matrix C is the product of Bt and B, as shown here:

```
C = Bt * B = | 15.25 4 |
             |  4    8 |
```

Note that if the units of measure of features x and y do not have a similar scale, then the covariance matrix is adversely affected. In this case, the solution is simple: use the correlation matrix, which defined in the next section.

The Correlation Matrix

As you learned in the preceding section, if the units of measure of features x and y do not have a similar scale, then the covariance matrix is adversely affected. The solution involves the correlation matrix, which equals the covariance values cov(x,y) divided by the standard deviation stdx and stdy of x and y, respectively, as shown here:

```
corr(x,y) = cov(x,y)/[stdx * stdy]
```

The correlation matrix no longer has units of measure, and we can use this matrix to find the eigenvalues and eigenvectors.

Now that you understand how to calculate the covariance matrix and the correlation matrix, you are ready for an example of calculating eigenvalues and eigenvectors, which are the topic of the next section.

Eigenvalues and Eigenvectors

According to a well-known theorem in mathematics (whose proof you can find online), the eigenvalues of a symmetric real matrix are real numbers. Consequently, the eigenvectors of C are vectors in a Euclidean vector space (not a complex vector space).

Before we continue, a nonzero vector x' is an eigenvector of the matrix C if there is a nonzero scalar lambda such that C*x' = lambda * x'.

Now suppose that the eigenvalues of C are b1, b2, b3, and b4, in decreasing numeric order from left to right, and that the corresponding eigenvectors of C are the vectors w1, w2, w3, and w4. Then the matrix M that consists of the column vectors w1, w2, w3, and w4 represents the principal components.

CALCULATING EIGENVECTORS: A SIMPLE EXAMPLE

As a simple illustration of calculating eigenvalues and eigenvectors, suppose that the square matrix C is defined as follows:

```
C = | 1  3 |
    | 3  1 |
```

Let I denote the 2x2 identity matrix, and let b be an eigenvalue of C, which means that there is an eigenvector x' such that:

```
C* x' = b * x', or
(C-b*I)*x' = 0 (the right side is a 2x1 vector)
```

Since `x'` is nonzero, that means the following is true (where `det` refers to the *determinant* of a matrix):

```
det(C-b*I) = det |1-b 3   | = (1-b)*(1-b)-9 = 0
                 |3   1-b|
```

We can expand the quadratic equation in the preceding line to get:

```
det(C-b*I) = (1-b)*(1-b) - 9
           = 1 - 2*b + b*b - 9
           = -8 - 2*b + b*b
           = b*b - 2*b - 8
```

Use the quadratic formula (or perform factorization by visual inspection) to determine that the solution for `det(C-b*I) = 0` is `b = -2` or `b = 4`. Next, substitute `b = –2` into `(C-b*I)x' = 0` and we get the following result:

```
|1-(-2) 3      | |x1| = |0|
|3      1-(-2)| |x2|   |0|
```

The preceding reduces to the following identical equations:

```
3*x1 + 3*x2 = 0
3*x1 + 3*x2 = 0
```

The general solution is `x1 = -x2`, and we can choose any nonzero value for `x2`, so let's set `x2 = 1` (any nonzero value will do just fine), which yields `x1 = –1`. Therefore, the eigenvector [–1, 1] is associated with the eigenvalue –2. In a similar fashion, if `x'` is an eigenvector whose eigenvalue is 4, then [1,1] is an eigenvector.

Notice that the eigenvectors [–1, 1] and [1,1] are orthogonal because their inner product is zero, as shown here:

```
[-1,1] * [1,1] = (-1)*1 + (1)*1 = 0
```

In fact, the set of eigenvectors of a square matrix (whose eigenvalues are real) are always orthogonal, regardless of the dimensionality of the matrix.

Gauss Jordan Elimination (Optional)

This simple technique enables you to find the solution to systems of linear equations "in place", which involves a sequence of arithmetic operations to transform a given matrix into an identity matrix.

The following example combines the Gauss-Jordan elimination technique (which finds the solution to a set of linear equations) with the "bookkeeper's method", which determines the inverse of an invertible matrix (its determinant is nonzero).

This technique involves two adjacent matrices: the left-side matrix is the initial matrix, and the right-side matrix is an identity matrix. Next, perform various linear operations on the left-side matrix to reduce it to an identity matrix:

the matrix on the right side equals its inverse. For example, consider the following pair of linear equations whose solution is x = 1 and y = 2:

```
2*x + 2*y = 6
4*x - 1*y = 2
```

Step 1: create a 2x2 matrix with the coefficients of x in column 1 and the coefficients of y in column two, followed by the 2x2 identity matrix, and finally a column from the numbers on the right of the equals sign:

```
| 2  2 | 1 0 | 6|
| 4 -1 | 0 1 | 2|
```

Step 2: add (–2) times the first row to the second row:

```
| 2  2  | 1  0 |6  |
| 0  -5 | -2 1 |-10|
```

Step 3: divide the second row by 5:

```
| 2  2  | 1      0 |6    |
| 0  -1 | -2/5 1/5 |-10/5|
```

Step 4: add 2 times the second row to the first row:

```
| 2  0  |  1/5 2/5 |2 |
| 0  -1 | -2/5 1/5 |-2|
```

Step 5: divide the first row by 2:

```
| 1  0  | -2/10 2/10 |1 |
| 0  -1 | -2/5   1/5 |-2|
```

Step 6: multiply the second row by (–1):

```
| 1  0  | -2/10 2/10 |1|
| 0  1  |  2/5  -1/5 |2|
```

As you can see, the left-side matrix is the 2x2 identity matrix, the right-side matrix is the inverse of the original matrix, and the rightmost column is the solution to the original pair of linear equations (x = 1 and y = 2).

PCA (PRINCIPAL COMPONENT ANALYSIS)

PCA is a linear dimensionality reduction technique for determining the most important features in a dataset. This section discusses PCA because it's a very popular technique that you will encounter frequently. Other techniques are more efficient than PCA, so later on it's worthwhile to learn other dimensionality reduction techniques as well.

Keep in mind the following points regarding the PCA technique:

- PCA is a variance-based algorithm
- PCA creates variables that are linear combinations of the original variables
- The new variables are all pair-wise orthogonal
- PCA can be a useful preprocessing step before clustering
- PCA is generally preferred for data reduction

PCA can be useful for variables that are strongly correlated. If most of the coefficients in the correlation matrix are smaller than 0.3, PCA is not helpful. PCA provides some advantages: less computation time for training a model (for example, using only five features instead of 100 features), a simpler model, and the ability to render the data visually when two or three features are selected. Here is a key point about PCA:

PCA calculates the eigenvalues and the eigenvectors of the covariance (or correlation) matrix C.

If you have four or five components, you won't be able to display them visually, but you could select subsets of three components for visualization, and perhaps gain some additional insight into the dataset.

The PCA algorithm involves the following sequence of steps:

1. calculate the correlation matrix (from the covariance matrix) C of a dataset
2. find the eigenvalues of C
3. find the eigenvectors of C
4. construct a new matrix that comprises the eigenvectors

The covariance matrix and correlation matrix were explained in a previous section. You also saw the definition of eigenvalues and eigenvectors, along with an example of calculating eigenvalues and eigenvectors.

The eigenvectors are treated as column vectors that are placed adjacent to each other in decreasing order (from left to right) with respect to their associated eigenvectors.

PCA uses the variance as a measure of information: the higher the variance, the more important the component. In fact, just to jump ahead slightly: PCA determines the eigenvalues and eigenvectors of a covariance matrix (discussed in a previous section), and constructs a new matrix whose columns are eigenvectors, ordered from left to right in a sequence that matches the corresponding sequence of eigenvalues: the leftmost eigenvector has the largest eigenvalue, the next eigenvector has the second-largest eigenvalue, and continuing in this fashion until the rightmost eigenvector (which has the smallest eigenvalue).

Alternatively, there is an interesting theorem in linear algebra: if `C` is a symmetric matrix, then there is a diagonal matrix `D` and an orthogonal matrix `P` (the

columns are pair-wise orthogonal, which means their pair-wise inner product is zero), such that the following holds:

```
C = P * D * Pt (where Pt is the transpose of matrix P)
```

In fact, the diagonal values of `D` are eigenvalues, and the columns of `P` are the corresponding eigenvectors of the matrix `C`.

Fortunately, we can use `NumPy` and `Pandas` to calculate the mean, standard deviation, covariance matrix, correlation matrix, as well as the matrices `D` and `P` in order to determine the eigenvalues and eigenvectors.

As an interesting point: any positive definite square matrix has real-valued eigenvectors, which also applies to the covariance matrix `C` because it is a real-valued symmetric matrix.

The New Matrix of Eigenvectors

The previous section described how the matrices `D` and `P` are determined. The leftmost eigenvector of `D` has the largest eigenvalue, the next eigenvector has the second-largest eigenvalue, and so forth. This fact is very convenient: the eigenvector with the highest eigenvalue is the principal component of the dataset. The eigenvector with the second-highest eigenvalue is the second principal component, and so forth. You specify the number of principal components that you want via the `n_components` hyperparameter in the PCA class of Sklearn (discussed briefly in Chapter 6).

As a simple and minimalistic example, consider the following code block that uses PCA for a (somewhat contrived) dataset:

```
import numpy as np
from sklearn.decomposition import PCA
data = np.array([[-1,-1], [-2,-1], [-3,-2], [1,1], [2,1],
[3,2]])
pca = PCA(n_components=2)
pca.fit(X)
```

Note the trade-off here: we greatly reduce the number of components, which reduces the computation time and the complexity of the model, but we also lose some accuracy. However, if the unselected eigenvalues are small, we lose only a small amount of accuracy.

Now let's use the following notation:

NM denotes the matrix with the new principal components
NMt is the transpose of NM
PC is the matrix of the subset of selected principal components
SD is the matrix of scaled data from the original dataset
SDt is the transpose of SD

Then the matrix NM is calculated via the following formula:

```
NM = PCt * SDt
```

Although PCA is a very nice technique for dimensionality reduction, keep in mind the following limitations of PCA:

less suitable for data with nonlinear relationships
less suitable for special classification problems

A related algorithm is called Kernel PCA, which is an extension of PCA that introduces a nonlinear transformation so you can still use the PCA approach.

WELL-KNOWN DISTANCE METRICS

There are several similarity metrics available, such as item similarity metrics, Jaccard (user-based) similarity, and cosine similarity (which is used to compare vectors of numbers). The following subsections introduce you to these similarity metrics.

Another well-known distance metric is the so-called "taxicab" metric, which is also called the Manhattan distance metric. Given two points A and B in a rectangular grid, the taxicab metric calculates the distance between two points by counting the number of "blocks" that must be traversed in order to reach B from A (the other direction has the same taxicab metric value). For example, if you need to travel two blocks north and then three blocks east in a rectangular grid, then the Manhattan distance is 5.

In fact, there are various other metrics available that you can learn about by searching Wikipedia. In the case of NLP, the most commonly used distance metric is calculated via the cosine similarity of two vectors, and it's derived from the formula for the inner ("dot") product of two vectors.

Pearson Correlation Coefficient

Pearson similarity is the Pearson coefficient between two vectors. Given random variables X and Y, and the following terms:

```
std(X)   = standard deviation of X
std(Y)   = standard deviation of Y
cov(X,Y) = covariance of X and Y
```

Then the Pearson correlation coefficient rho(X,Y) is defined as follows:

$$\text{rho}(X,Y) = \frac{\text{cov}(X,Y)}{\text{std}(X)*\text{std}(Y)}$$

Keep in mind that the Pearson coefficient is limited to items of the same type. More information about the Pearson correlation coefficient is here:

https://en.wikipedia.org/wiki/Pearson_correlation_coefficient.

Jaccard Index (or Similarity)

The Jaccard similarity is based on the number of users which have rated item A and B divided by the number of users who have rated either A or B. Jaccard similarity is based on unique words in a sentence and is unaffected by duplicates, whereas cosine similarity is based on the length of all word vectors (which changes when duplicates are added). The choice between cosine similarity and Jaccard similarity depends on whether or not word duplicates are important.

The following Python method illustrates how to compute the Jaccard similarity of two sentences:

```
def get_jaccard_sim(str1, str2):
  set1 = set(str1.split())
  set2 = set(str2.split())
  set3 = set1.intersection(set2)
  # (size of intersection) / (size of union):
  return float(len(set3)) / (len(set1) + len(set2) -
len(set3))
```

Jaccard similarity can be used in situations involving Boolean values, such as product purchases (true/false), instead of numeric values. More information is available here:

https://en.wikipedia.org/wiki/Jaccard_index.

Local Sensitivity Hashing (Optional)

If you are familiar with hash algorithms, you know that they are algorithms that create a hash table that associates items with a value. The advantage of hash tables is that the lookup time to determine whether or not an item exists in the hash table is constant. Of course, it's possible for two items to "collide", which means that they both occupy the same bucket in the hash table. In this case, a bucket can consist of a list of items that can be searched in more or less constant time. If there are too many items in the same bucket, then a different hashing function can be selected to reduce the number of collisions. The goal of a hash table is to minimize the number of collisions.

The local sensitivity hashing (LSH) algorithm hashes similar input items into the same "buckets". In fact, the goal of LSH is to maximize the number of collisions, whereas traditional hashing algorithms attempt to minimize the number of collisions.

Since similar items end up in the same buckets, LSH is useful for data clustering and nearest neighbor searches. Moreover, LSH is a dimensionality reduction technique that places data points of high dimensionality closer together in a lower-dimensional space, while simultaneously preserving the relative distances between those data points.

More details about LSH are here:

https://en.wikipedia.org/wiki/Locality-sensitive_hashing.

TYPES OF DISTANCE METRICS

Nonlinear dimensionality reduction techniques can also have different distance metrics. For example, linear reduction techniques can use the Euclidean distance metric (based on the Pythagorean theorem). However, you need to use a different distance metric to measure the distance between two points on a sphere (or some other curved surface). In the case of NLP, the cosine similarity metric is used to measure the distance between word embeddings (which are vectors of floating point numbers that represent words or tokens).

Distance metrics are used for measuring physical distances, and some well-known distance metrics are listed here:

Euclidean distance
Manhattan distance
Chebyshev distance

The Euclidean algorithm also obeys the "triangle inequality", which states that for any triangle in the Euclidean plane, the sum of the lengths of any pair of sides must be greater than the length of the third side.

In spherical geometry, you can define the distance between two points as the arc of a great circle that passes through the two points (always selecting the smaller of the two arcs when they are different).

In addition to physical metrics, there are algorithms that implement the concept of "edit distance" (the distance between strings), as listed here:

Hamming distance
Jaro–Winkler distance
Lee distance
Levenshtein distance
Mahalanobis distance metric
Wasserstein metric

The Mahalanobis metric is based on an interesting idea: given a point P and a probability distribution D, this metric measures the number of standard deviations that separate point P from distribution D. More information about Mahalanobis is here:

https://en.wikipedia.org/wiki/Mahalanobis_distance.

In the branch of mathematics called topology, a metric space is a set for which distances between all members of the set are defined. Various metrics are available (such as the Hausdorff metric), depending on the type of topology.

The Wasserstein metric measures the distance between two probability distributions over a metric space X. This metric is also called the "earth mover's metric" for the following reason: given two unit piles of dirt, it's the measure of the minimum cost of moving one pile on top of the other pile.

KL divergence bears some superficial resemblance to the Wasserstein metric. However, there are some important differences between them. Specifically, the Wasserstein metric has the following properties:

1. it is a metric
2. it is symmetric
3. it satisfies the triangle inequality

On the other hand, KL divergence has the following properties:

1. it is not a metric (it's a divergence)
2. it is not symmetric: KL(P,Q) != KL(Q,P)
3. it does not satisfy the triangle inequality

Note that the JS divergence (which is based on the KL Divergence) is a true metric, which would enable a more meaningful comparison with other metrics (such as the Wasserstein metric):

https://stats.stackexchange.com/questions/295617/what-is-the-advantages-of-wasserstein-metric-compared-to-kullback-leibler-diverg.

More information is here:

https://en.wikipedia.org/wiki/Wasserstein_metric.

WHAT IS BAYESIAN INFERENCE?

Bayesian inference is an important technique in statistics that involves statistical inference and Bayes's theorem to update the probability for a hypothesis as more information becomes available. Bayesian inference is often called "Bayesian probability", and it's important in dynamic analysis of sequential data.

Bayes's Theorem

Given two sets A and B, let's define the following numeric values (all of them are between 0 and 1):

```
P(A) = probability of being in set A
P(B) = probability of being in set B
P(Both) = probability of being in A intersect B
P(A|B) = probability of being in A (given you're in B)
P(B|A) = probability of being in B (given you're in A)
```

Then the following formulas are also true:

```
P(A|B) = P(Both)/P(B)  (#1)
P(B|A) = P(Both)/P(A)  (#2)
```

Multiply the preceding pair of equations by the term that appears in the denominator and we get these equations:

```
P(B)*P(A|B) = P(Both) (#3)
P(A)*P(B|A) = P(Both) (#4)
```

Now set the left-side of equations #3 and #4 equal to each another and that gives us this equation:

```
P(B)*P(A|B) = P(A)*P(B|A) (#5)
```

Divide both sides of #5 by P(B) and we get this well-known equation:

```
P(A|B) = P(A)*P(A|B)/P(B) (#6)
```

Some Bayesian Terminology

In the previous section, we derived the following relationship:

```
P(h|d) = (P(d|h) * P(h)) / P(d)
```

There is a name for each of the four terms in the preceding equation, discussed as follows.

First, the *posterior probability* is `P(h|d)`, which is the probability of hypothesis `h` given the data `d`.

Second, `P(d|h)` is the probability of data `d` given that the hypothesis `h` was true.

Third, the *prior probability* of `h` is `P(h)`, which is the probability of hypothesis `h` being true (regardless of the data).

Finally, `P(d)` is the probability of the data (regardless of the hypothesis).

We are interested in calculating the posterior probability of P(h|d) from the prior probability p(h) with P(D) and P(d|h).

What Is MAP?

The maximum a posteriori (MAP) hypothesis is the hypothesis with the highest probability, which is the maximum probable hypothesis. This can be written as follows:

```
MAP(h) = max(P(h|d))
```

or:

```
MAP(h) = max((P(d|h) * P(h)) / P(d))
```

or:

```
MAP(h) = max(P(d|h) * P(h))
```

Why Use Bayes's Theorem?

Bayes's Theorem describes the probability of an event based on the prior knowledge of the conditions that might be related to the event. If we know the conditional probability, we can use Bayes' rule to find out the reverse probabilities. The previous statement is the general representation of the Bayes rule.

SUMMARY

This chapter started with a discussion of probability, expected values, and the concept of a random variable. Then you learned about some basic statistical concepts, such as mean, median, mode, variance, and standard deviation. Next, you learned about the terms RSS, TSS, R^2, and F1 score. In addition, you got an introduction to the concepts of skewness, kurtosis, Gini impurity, entropy, perplexity, cross-entropy, and KL divergence.

Next, you learned about covariance and correlation matrices and how to calculate eigenvalues and eigenvectors. Then you were introduced to the dimensionality reduction technique known as PCA (principal component analysis), after which you learned about Bayes' theorem.

CHAPTER 6

DATA VISUALIZATION

This chapter introduces data visualization, along with a wide-ranging collection of Python-based code samples that use various visualization tools (including `Matplotlib` and `Seaborn`) to render charts and graphs. In addition, this chapter contains Python code samples that combine `Pandas`, `Matplotlib`, and `Sklearn` built-in datasets.

Since the title of the chapter is data visualization, you might be wondering why this chapter contains an introduction to `Sklearn` (also known as `Scikit-learn`). The reason is straightforward: the easy introduction to some `Sklearn` functionality is possible without a more formal learning process. In addition, this knowledge will bode well if you decide to delve into machine learning (and perhaps this section will provide additional motivation to do so). However, a thorough understanding of `Sklearn` involves significantly more time and effort, especially if you plan to learn the details of the `Sklearn` machine learning algorithms. On the other hand, if you are not interested in learning about `Sklearn` at this point in time, you can skip this section and perhaps return to it when you are interested in learning this material.

The first part of this chapter briefly discusses data visualization, with a short list of some data visualization tools, and a list of various types of visualization (bar graphs, pie charts, and so forth).

The second part of this chapter introduces you to `Matplotlib`, which is an open-source Python library that is modeled after MATLAB. This section also provides the Python code samples for the line graphs (horizontal, vertical, and diagonal) in the Euclidean plane that you saw in a previous chapter.

The third part of the chapter introduces you to `Sklearn`, which is a very powerful Python library that supports many machine learning algorithms and also supports visualization. If you are new to machine learning, fear not: *this section does not require a background in machine learning in order to understand the Python code samples.*

The fourth part of the chapter introduces you to Seaborn for data visualization, which is a layer above Matplotlib. Although Seaborn does not have all of the features that are available in Matplotlib, Seaborn provides an easier set of APIs for rendering charts and graphs.

The final portion of this chapter contains a very short introduction to Bokeh, along with a code sample that illustrates how to create a more artistic graphics effect with relative ease in Bokeh.

WHAT IS DATA VISUALIZATION?

Data visualization refers to presenting data in a graphical manner, such as bar charts, line graphs, heat maps, and many other specialized representations. As you probably know, Big Data comprises massive amounts of data, which leverages data visualization tools to assist in making better decisions.

A key role for good data visualization is to tell a meaningful story, which in turn focuses on useful information that resides in datasets that can contain many data points (i.e., billions of rows of data). Another aspect of data visualization is its effectiveness: how well does it convey the trends that might exist in the dataset?

There are many open-source data visualization tools available, some of which are listed here (many others are available):

- Matplotlib
- Seaborn
- Bokeh
- YellowBrick
- Tableau
- D3.js (JavaScript and SVG)

Incidentally, in case you have not already done so, it would be helpful to install the following Python libraries (using pip3) on your computer so that you can launch the code samples in this chapter:

```
pip3 install matplotlib
pip3 install seaborn
pip3 install bokeh
```

Types of Data Visualization

Bar graphs, line graphs, and pie charts are common ways to present data, and yet many other types exist, some of which are listed as follows:

- 2D/3D Area Chart
- Bar Chart
- Gantt Chart

- Heat Map
- Histogram
- Polar Area
- Scatterplot (2D or 3D)
- Timeline

The Python code samples in the next several sections illustrate how to perform visualization via rudimentary APIs from `matplotlib`.

WHAT IS MATPLOTLIB?

`Matplotlib` is a plotting library that supports `NumPy`, `SciPy`, and toolkits such as `wxPython` (among others). `Matplotlib` supports only version 3 of Python: support for version 2 of Python was available only through 2020. `Matplotlib` is a multi-platform library that is built on `NumPy` arrays.

The plotting-related code samples in this chapter use `pyplot`, which is a `Matplotlib` module that provides a MATLAB-like interface. Here is an example of using `pyplot` (copied from *https://www.biorxiv.org/content/10.1101/120378v1.full.pdf*) to plot a smooth curve based on negative powers of Euler's constant e:

```
import matplotlib.pyplot as plt
import numpy as np

a = np.linspace(0, 10, 100)
b = np.exp(-a)
plt.plot(a, b)
plt.show()
```

The Python code samples for visualization in this chapter use primarily `Matplotlib`, along with some code samples that use `Seaborn`. Keep in mind that the code samples that plot line segments assume that you are familiar with the equation of a (nonvertical) line in the plane: `y = m*x + b`, where `m` is the slope and `b` is the y-intercept.

Furthermore, some code samples use `NumPy` APIs such as `np.linspace()`, `np.array()`, `np.random.rand()`, and `np.ones()` that are discussed in Chapter 3, so you can refresh your memory regarding these APIs.

Now let's proceed to the next section that contains a fast-paced set of basic code samples that display various types of line segments.

HORIZONTAL LINES IN MATPLOTLIB

Listing 6.1 displays the contents of `hlines1.py` that illustrates how to plot horizontal lines using `Matplotlib`. Recall that the equation of a nonvertical line in the 2D plane is `y = m*x + b`, where `m` is the slope of the line and `b` is the y-intercept of the line.

LISTING 6.1: hlines1.py

```
import numpy as np
import matplotlib.pyplot as plt

# top line
x1 = np.linspace(-5,5,num=200)
y1 = 4 + 0*x1

# middle line
x2 = np.linspace(-5,5,num=200)
y2 = 0 + 0*x2

# bottom line
x3 = np.linspace(-5,5,num=200)
y3 = -3 + 0*x3

plt.axis([-5, 5, -5, 5])
plt.plot(x1,y1)
plt.plot(x2,y2)
plt.plot(x3,y3)
plt.show()
```

Listing 6.1 uses the `np.linspace()` API in order to generate a list of 200 equally spaced numbers for the horizontal axis, all of which are between –5 and 5. The three lines defined via the variables `y1`, `y2`, and `y3` are defined in terms of the variables `x1`, `x2`, and `x3`, respectively.

Figure 6.1 displays three horizontal line segments whose equations are contained in Listing 6.1.

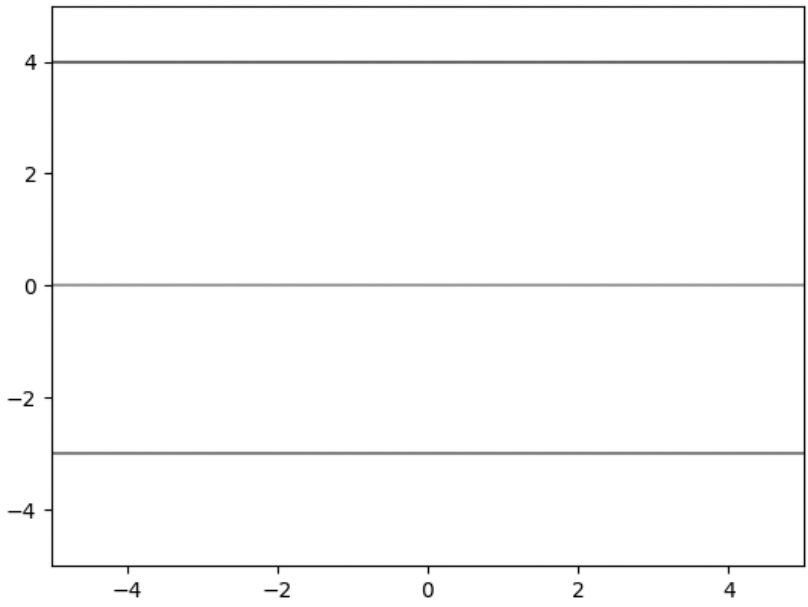

FIGURE 6.1 A graph of three horizontal line segments.

SLANTED LINES IN MATPLOTLIB

Listing 6.2 displays the contents of `diagonallines.py` that illustrates how to plot slanted lines.

LISTING 6.2: diagonallines.py

```
import matplotlib.pyplot as plt
import numpy as np

x1 = np.linspace(-5,5,num=200)
y1 = x1

x2 = np.linspace(-5,5,num=200)
y2 = -x2

plt.axis([-5, 5, -5, 5])
plt.plot(x1,y1)
plt.plot(x2,y2)
plt.show()
```

Listing 6.2 defines two lines using the technique that you saw in Listing 6.1, except that these two lines define `y1 = x1` and `y2 = -x2`, which produces slanted lines instead of horizontal lines.

Figure 6.2 displays two slanted line segments whose equations are defined in Listing 6.2.

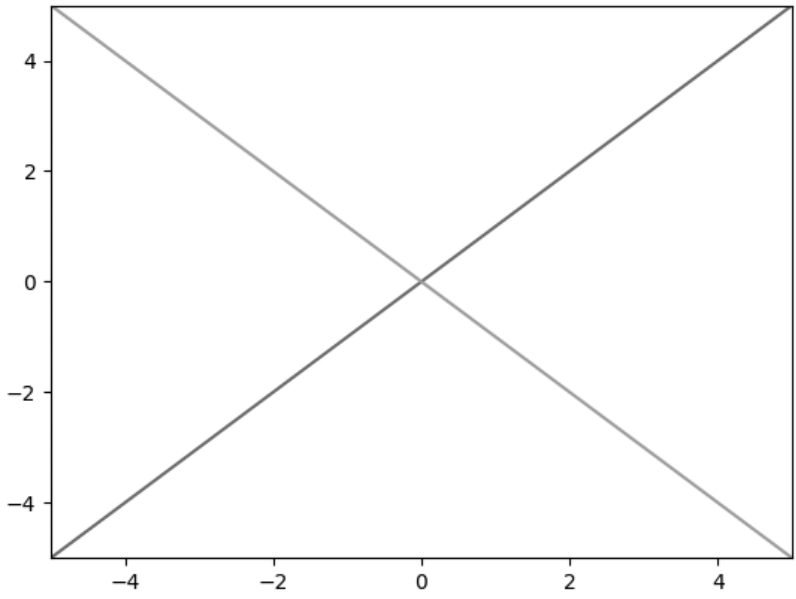

FIGURE 6.2 A graph of two slanted line segments.

PARALLEL SLANTED LINES IN MATPLOTLIB

If two lines in the Euclidean plane have the same slope then they are parallel. Listing 6.3 displays the contents of `parallellines1.py` that illustrates how to plot parallel slanted lines.

LISTING 6.3: parallellines1.py

```
import matplotlib.pyplot as plt
import numpy as np

# lower line
x1 = np.linspace(-5,5,num=200)
y1 = 2*x1

# upper line
x2 = np.linspace(-5,5,num=200)
y2 = 2*x2 + 3

# horizontal axis
x3 = np.linspace(-5,5,num=200)
y3 = 0*x3 + 0

# vertical axis
plt.axvline(x=0.0)

plt.axis([-5, 5, -10, 10])
plt.plot(x1,y1)
plt.plot(x2,y2)
plt.plot(x3,y3)
plt.show()
```

Listing 6.3 defines three lines using the technique that you saw in Listing 6.1, where these three lines are slanted and also parallel to each other.

Figure 6.3 displays two slanted and also parallel line segments whose equations are defined in Listing 6.3.

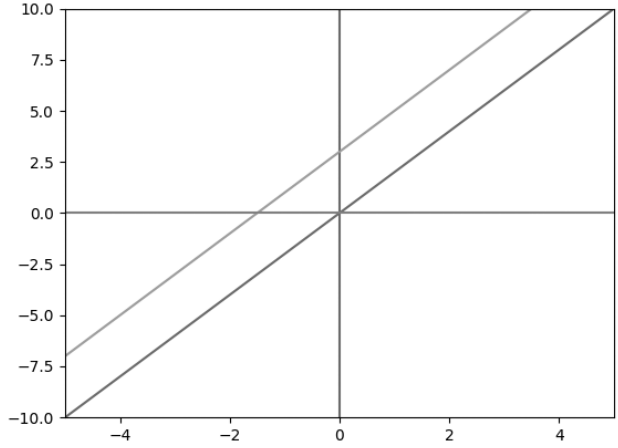

FIGURE 6.3 A graph of two slanted parallel line segments.

A GRID OF POINTS IN MATPLOTLIB

Listing 6.4 displays the contents of `plotgrid.py` that illustrates how to plot a simple grid.

LISTING 6.4: plotgrid.py

```
import numpy as np
from itertools import product
import matplotlib.pyplot as plt

points = np.array(list(product(range(3),range(4))))

plt.plot(points[:,0],points[:,1],'ro')
plt.show()
```

Listing 6.4 defines the `NumPy` variable `points` that defines a 2D list of points with three rows and four columns. The Pyplot API `plot()` uses the `points` variable to display a grid-like pattern. Figure 6.4 displays a grid of points as defined in Listing 6.4.

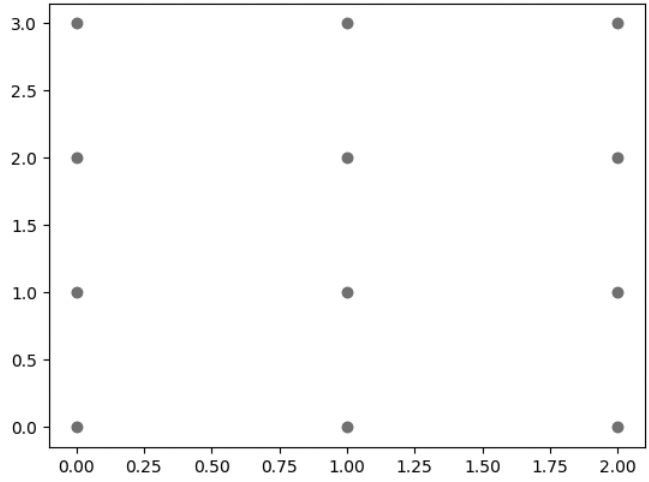

FIGURE 6.4 A grid of points.

A DOTTED GRID IN MATPLOTLIB

Listing 6.5 displays the contents of `plotdottedgrid1.py` that illustrates how to plot a "dotted" grid pattern.

LISTING 6.5: plotdottedgrid1.py

```
import numpy as np
import pylab
```

```
from itertools import product
import matplotlib.pyplot as plt

fig = pylab.figure()
ax = fig.add_subplot(1,1,1)

ax.grid(which='major', axis='both', linestyle='--')

[line.set_zorder(3) for line in ax.lines]
fig.show() # to update

plt.gca().xaxis.grid(True)
plt.show()
```

Listing 6.5 is similar to the code in Listing 6.4 in that both of them plot a grid-like pattern; however, Listing 6.5 renders a "dotted" grid pattern whereas Listing 6.4 renders a rectangular grid of points by specifying the value `'--'` for the `linestyle` parameter.

The next portion of Listing 6.5 invokes the `set_zorder()` method that controls which items are displayed on top of other items, such as dots on top of lines, or vice versa. The final portion of Listing 6.5 invokes the `gca().xaxis.grid(True)` chained methods to display the vertical grid lines.

You can also use the `plt.style` directive to specify a style for figures. The following code snippet specifies the classic style of `Matplotlib`:

```
plt.style.use('classic')
```

Figure 6.5 displays a "dashed" grid pattern based on the code in Listing 6.5.

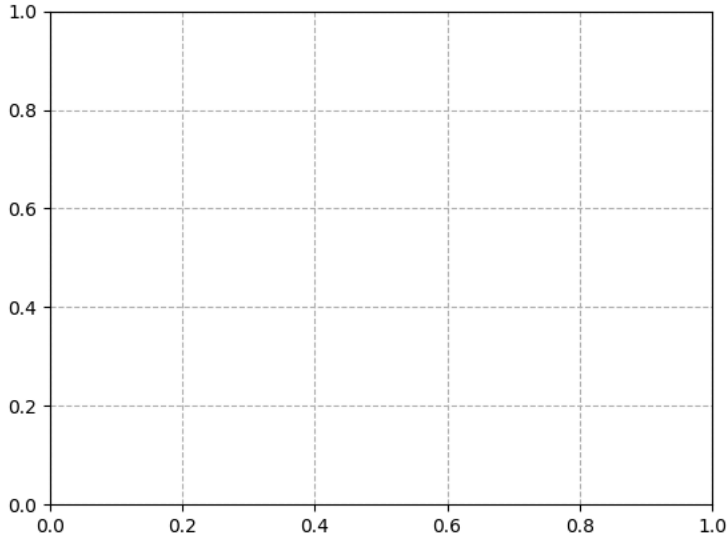

FIGURE 6.5 A "dashed" grid pattern.

LINES IN A GRID IN MATPLOTLIB

Listing 6.6 displays the contents of `plotlinegrid2.py` that illustrates how to plot lines in a grid.

LISTING 6.6: plotlinegrid2.py

```
import numpy as np
import pylab
from itertools import product
import matplotlib.pyplot as plt

fig = plt.figure()
graph = fig.add_subplot(1,1,1)
graph.grid(which='major', linestyle='-', linewidth='0.5',
color='red')

x1 = np.linspace(-5,5,num=200)
y1 = 1*x1
graph.plot(x1,y1, 'r-o')

x2 = np.linspace(-5,5,num=200)
y2 = -x2
graph.plot(x2,y2, 'b-x')

fig.show() # to update
plt.show()
```

Listing 6.6 defines the `NumPy` variable `points` that defines a 2D list of points with three rows and four columns. The Pyplot API `plot()` uses the `points` variable to display a grid-like pattern.

Figure 6.6 displays a set of "dashed" line segments whose equations are contained in Listing 6.6.

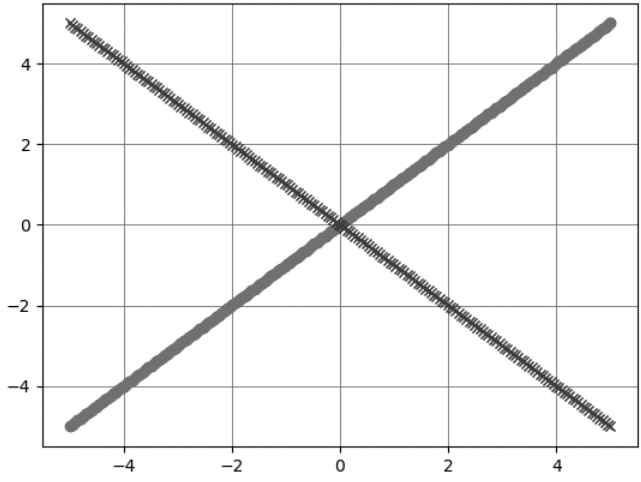

FIGURE 6.6 A grid of line segments.

A COLORED GRID IN MATPLOTLIB

Listing 6.7 displays the contents of `plotgrid2.py` that illustrates how to display a colored grid.

LISTING 6.7: plotgrid2.py

```
import matplotlib.pyplot as plt
from matplotlib import colors
import numpy as np

data = np.random.rand(10, 10) * 20

# create discrete colormap
cmap = colors.ListedColormap(['red', 'blue'])
bounds = [0,10,20]
norm = colors.BoundaryNorm(bounds, cmap.N)

fig, ax = plt.subplots()
ax.imshow(data, cmap=cmap, norm=norm)

# draw gridlines
ax.grid(which='major', axis='both', linestyle='-',
color='k', linewidth=2)
ax.set_xticks(np.arange(-.5, 10, 1));
ax.set_yticks(np.arange(-.5, 10, 1));

plt.show()
```

Listing 6.7 defines the `NumPy` variable `data` that defines a 2D set of points with ten rows and ten columns. The Pyplot API `plot()` uses the `data` variable to display a colored grid-like pattern.

Figure 6.7 displays a colored grid whose equations are contained in Listing 6.7.

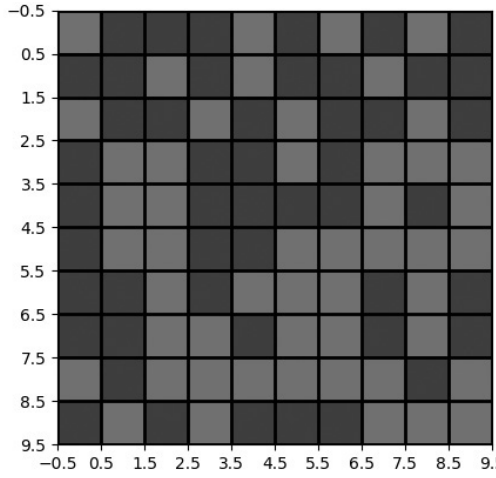

FIGURE 6.7 A colored grid of line segments.

A COLORED SQUARE IN AN UNLABELED GRID IN MATPLOTLIB

Listing 6.8 displays the contents of `plotgrid3.py` that illustrates how to display a colored square inside a grid.

LISTING 6.8: plotgrid3.py

```
import matplotlib.pyplot as plt
import numpy as np
from itertools import product
import Matplotlib.pyplot as plt
import Matplotlib.colors as colors

N = 15
# create an empty data set
data = np.ones((N, N)) * np.nan

# fill in some fake data
for j in range(3)[::-1]:
  data[N//2 - j : N//2 + j +1, N//2 - j : N//2 + j +1] = j

# make a figure + axes
fig, ax = plt.subplots(1, 1, tight_layout=True)

# make color map
my_cmap = colors.ListedColormap(['r', 'g', 'b'])

# set the 'bad' values (nan) to be white and transparent
my_cmap.set_bad(color='w', alpha=0)

# draw the grid
for x in range(N + 1):
  ax.axhline(x, lw=2, color='k', zorder=5)
  ax.axvline(x, lw=2, color='k', zorder=5)

# draw the boxes
ax.imshow(data, interpolation='none', cmap=my_cmap,
extent=[0, N, 0, N], zorder=0)

# turn off the axis labels
ax.axis('off')
plt.show()
```

Listing 6.8 defines the `NumPy` variable `data` that defines an NxN set of 2D points, followed by a `for` loop that initializes the variable `data` as a 15x15 array of `NaN` values. The Pyplot API `plot()` uses the `data` variable to display a colored square inside a grid-like pattern.

Figure 6.8 displays a colored square in a grid whose equations are contained in Listing 6.8.

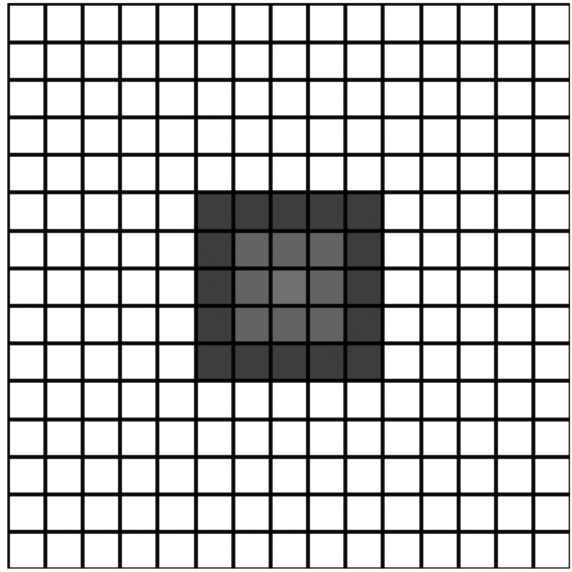

FIGURE 6.8 A colored square in a grid of line segments.

RANDOMIZED DATA POINTS IN MATPLOTLIB

Listing 6.9 displays the contents of `lin_reg_plot.py` that illustrates how to plot a graph of random points.

LISTING 6.9: lin_plot_reg.py

```
import numpy as np
import matplotlib.pyplot as plt

trX = np.linspace(-1, 1, 101) # Linear space of 101 and
[-1,1]

#Create the y function based on the x axis
trY = 2*trX + np.random.randn(*trX.shape)*0.4+0.2

#create figure and scatter plot of the random points
plt.figure()
plt.scatter(trX,trY)

# Draw one line with the line function
plt.plot (trX, .2 + 2 * trX)
plt.show()
```

Listing 6.9 defines the `NumPy` variable `trX` that contains 101 equally spaced numbers that are between –1 and 1 (inclusive). The variable `trY` is defined in two parts: the first part is `2*trX` and the second part is a random value that is partially based on the length of the one-dimensional array `trX`. The variable

trY is the sum of these two "parts", which creates a "fuzzy" line segment. The next portion of Listing 6.9 creates a scatterplot based on the values in trX and trY, followed by the Pyplot API plot() that renders a line segment.

Figure 6.9 displays a random set of points based on the code in Listing 6.9.

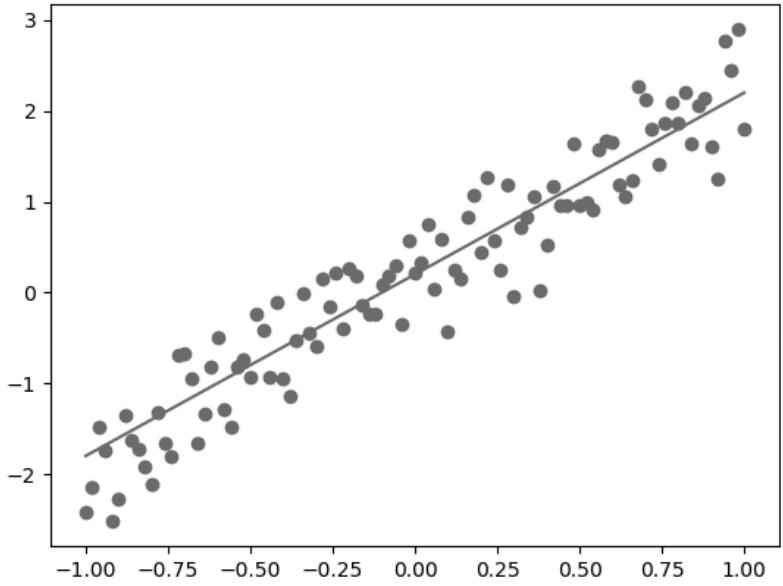

FIGURE 6.9 A random set of points.

A HISTOGRAM IN MATPLOTLIB

Listing 6.10 displays the contents of histogram1.py that illustrates how to plot a histogram using Matplotlib.

LISTING 6.10: histogram1.py

```
import numpy as np
import Matplotlib.pyplot as plt

max1 = 500
max2 = 500

appl_count = 28 + 4 * np.random.randn(max1)
bana_count = 24 + 4 * np.random.randn(max2)

plt.hist([appl_count, appl_count],stacked=True,color=['r','b'])
plt.show()
```

Listing 6.10 is straightforward: the NumPy variables appl_count and bana_count contain a random set of values whose upper bound is max1 and max2, respectively. The Pyplot API hist() uses the points appl_count

and `bana_count` in order to display a histogram. Figure 6.10 displays a histogram whose shape is based on the code in Listing 6.10.

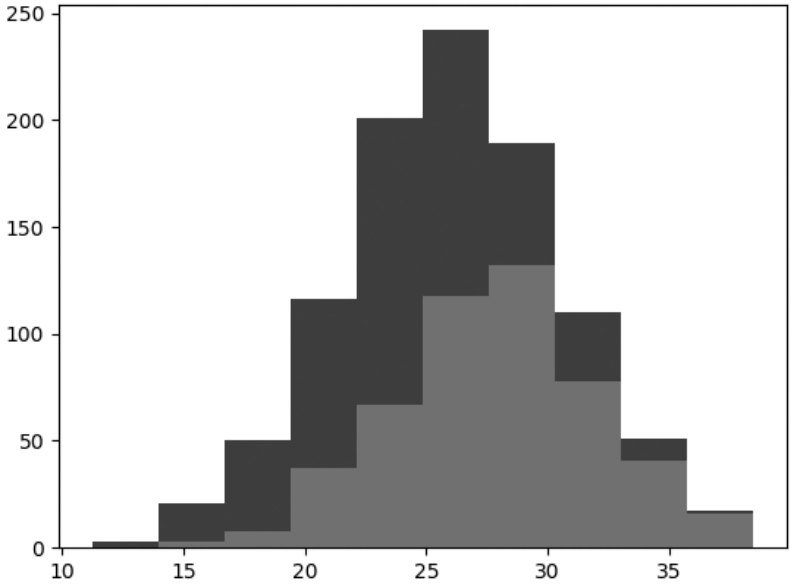

FIGURE 6.10 A histogram based on random values.

A SET OF LINE SEGMENTS IN `MATPLOTLIB`

Listing 6.11 displays the contents of `line_segments.py` that illustrates how to plot a set of connected line segments in `Matplotlib`.

LISTING 6.11: line_segments.py

```
import numpy as np
import matplotlib.pyplot as plt

x = [7,11,13,15,17,19,23,29,31,37]

plt.plot(x) # OR: plt.plot(x, 'ro-') or bo
plt.ylabel('Height')
plt.xlabel('Weight')
plt.show()
```

Listing 6.11 defines the array `x` that contains a hard-coded set of values. The Pyplot API `plot()` uses the variable `x` to display a set of connected line segments. Figure 6.11 displays the result of launching the code in Listing 6.11.

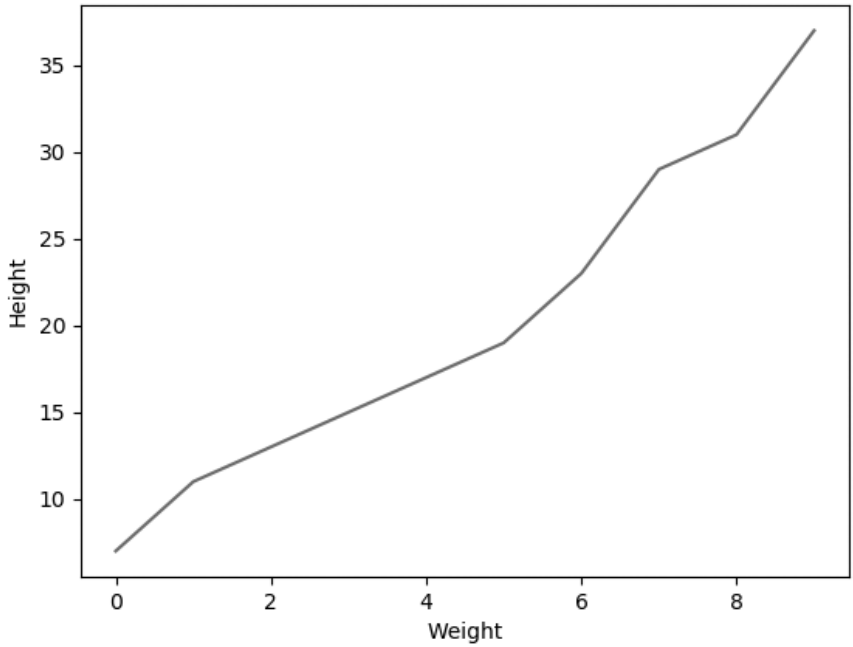

FIGURE 6.11 A set of connected line segments.

PLOTTING MULTIPLE LINES IN `MATPLOTLIB`

Listing 6.12 displays the contents of `plt_array2.py` that illustrates the ease with which you can plot multiple lines in `Matplotlib`.

LISTING 6.12: plt_array2.py

```
import matplotlib.pyplot as plt

x = [7,11,13,15,17,19,23,29,31,37]
data = [[8, 4, 1], [5, 3, 3], [6, 0, 2], [1, 7, 9]]
plt.plot(data, 'd-')
plt.show()
```

Listing 6.12 defines the array `data` that contains a hard-coded set of values. The Pyplot API `plot()` uses the variable `data` to display a line segment. Figure 6.12 displays multiple lines based on the code in Listing 6.12.

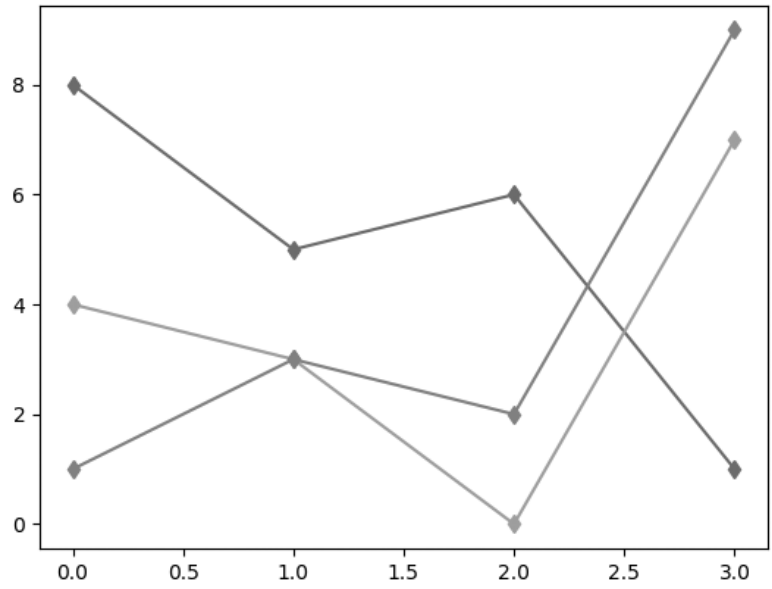

FIGURE 6.12 Multiple lines in Matplotlib.

TRIGONOMETRIC FUNCTIONS IN `MATPLOTLIB`

In case you're wondering, you can display the graph of trigonometric functions as easily as you can render "regular" graphs using `Matplotlib`. Listing 6.13 displays the contents of `sincos.py` that illustrates how to plot a sine function and a cosine function in `Matplotlib`.

LISTING 6.13: sincos.py

```
import numpy as np
import math

x = np.linspace(0, 2*math.pi, 101)
s = np.sin(x)
c = np.cos(x)

import matplotlib.pyplot as plt
plt.plot (s)
plt.plot (c)
plt.show()
```

Listing 6.13 defines the `NumPy` variables `x`, `s`, and `c` using the `NumPy` APIs `linspace()`, `sin()`, and `cos()`, respectively. Next, the Pyplot API `plot()` uses these variables to display a sine function and a cosine function.

Figure 6.13 displays a graph of two trigonometric functions based on the code in Listing 6.13.

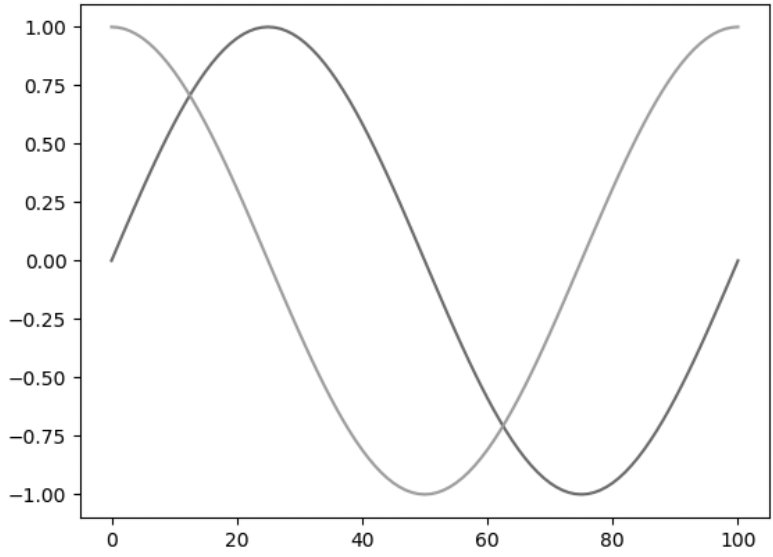

FIGURE 6.13 Sine and cosine trigonometric functions.

Now let's look at a simple dataset consisting of discrete data points, which is the topic of the next section.

DISPLAY IQ SCORES IN MATPLOTLIB

Listing 6.14 displays the contents of `iq_scores.py` that illustrates how to plot a histogram that displays IQ scores (based on a normal distribution).

LISTING 6.14: iq_scores.py

```
import numpy as npf
import matplotlib.pyplot as plt

mu, sigma = 100, 15
x = mu + sigma * np.random.randn(10000)

# the histogram of the data
n, bins, patches = plt.hist(x, 50, normed=1, facecolor='g',
alpha=0.75)

plt.xlabel('Intelligence')
plt.ylabel('Probability')
plt.title('Histogram of IQ')
plt.text(60, .025, r'$\mu=100,\ \sigma=15$')
plt.axis([40, 160, 0, 0.03])
plt.grid(True)
plt.show()
```

Listing 6.14 defines the scalar variables `mu` and `sigma`, followed by the `NumPy` variable `x` that contains a random set of points. Next, the variables `n`,

bins, and patches are initialized via the return values of the NumPy hist() API. Finally, these points are plotted via the usual plot() API to display a histogram.

Figure 6.14 displays a histogram whose shape is based on the code in Listing 6.14.

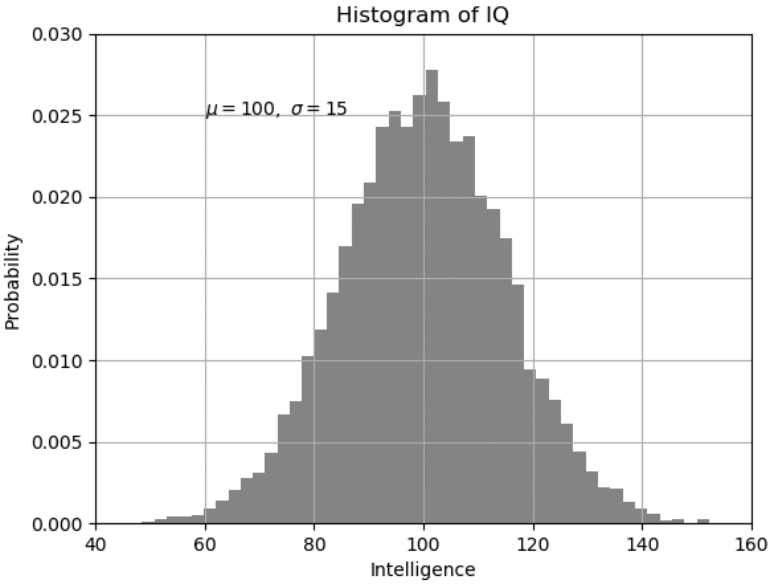

FIGURE 6.14 A histogram to display IQ scores.

PLOT A BEST-FITTING LINE IN MATPLOTLIB

Listing 6.15 displays the contents of plot_best_fit.py that illustrates how to plot a best-fitting line in Matplotlib.

LISTING 6.15: plot_best_fit.py

```
import numpy as np

xs = np.array([1,2,3,4,5], dtype=np.float64)
ys = np.array([1,2,3,4,5], dtype=np.float64)

def best_fit_slope(xs,ys):
  m = (((np.mean(xs)*np.mean(ys))-np.mean(xs*ys)) /
       ((np.mean(xs)**2) - np.mean(xs**2)))
  b = np.mean(ys) - m * np.mean(xs)

  return m, b

m,b = best_fit_slope(xs,ys)
print('m:',m,'b:',b)
```

```
regression_line = [(m*x)+b for x in xs]

import matplotlib.pyplot as plt
from matplotlib import style
style.use('ggplot')

plt.scatter(xs,ys,color='#0000FF')
plt.plot(xs, regression_line)
plt.show()
```

Listing 6.15 defines the `NumPy` array variables `xs` and `ys` that are "fed" into the Python function `best_fit_slope()` that calculates the slope `m` and the y-intercept `b` for the best-fitting line. The Pyplot API `scatter()` displays a scatterplot of the points `xs` and `ys`, followed by the `plot()` API that displays the best-fitting line. Figure 6.15 displays a simple line based on the code in Listing 6.15.

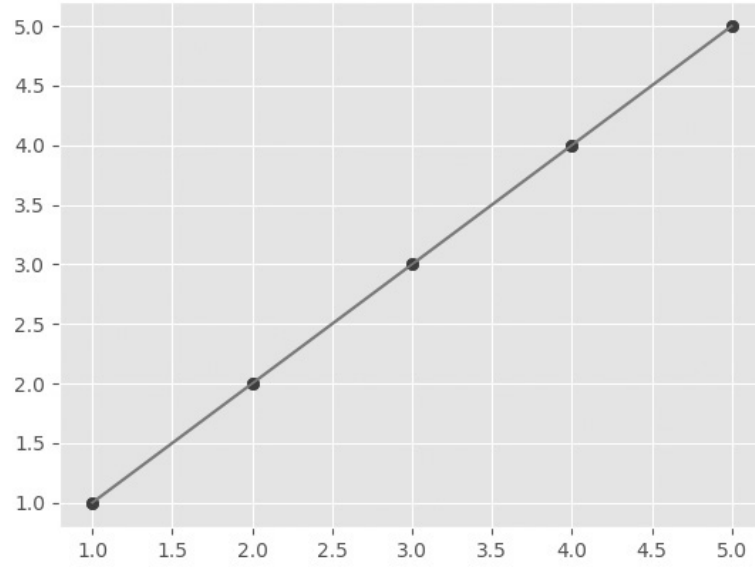

FIGURE 6.15 A best-fitting line for a 2D dataset.

This concludes the portion of the chapter regarding `NumPy` and `Matplotlib`. The next section introduces you to `Sklearn`, which is a powerful Python-based library that supports many algorithms for machine learning. After you have read the short introduction, subsequent sections contain Python code samples that combine `Pandas`, `Matplotlib`, and `Sklearn` built-in datasets.

INTRODUCTION TO SKLEARN (SCIKIT-LEARN)

`Sklearn` (which is installed as `sklearn`) is Python's premier general-purpose machine learning library, and its home page is here:

https://scikit-learn.org/stable/.

Before we discuss any code samples, please keep in mind that Sklearn is an immensely useful Python library that supports a huge number of machine learning algorithms. In particular, Sklearn supports many classification algorithms, such as logistic regression, naive Bayes, decision trees, random forests, and SVMs (support vector machines). Although entire books are available that are dedicated to Sklearn, this chapter contains only a few pages of Sklearn material.

If you decide that you want to acquire a deep level of knowledge about Sklearn, navigate to the Web pages that contain very detailed documentation for Sklearn. Moreover, if you have "how to" questions involving Sklearn, you can almost always find suitable answers on stackoverflow.

Sklearn is well-suited for classification tasks as well as regression and clustering tasks in machine learning. Sklearn supports a vast collection of machine learning algorithms including linear regression, logistic regression, kNN ("K Nearest Neighbor"), kMeans, decision trees, random forests, MLPs (multilayer perceptrons), and SVMs.

Moreover, Sklearn supports dimensionality reduction techniques such as PCA, "hyper parameter" tuning, methods for scaling data, and is suitable for preprocessing data, cross-validation, and so forth.

ML code samples often contain a combination of Sklearn, NumPy, Pandas, and Matplotlib. In addition, Sklearn provides various built-in datasets that we can display visually. One of those datasets is the Digits dataset, which is the topic of the next section.

The next section of this chapter provides several Python code samples that contain a combination of Pandas, Matplotlib, and the Sklearn built-in Digits dataset.

THE DIGITS DATASET IN SKLEARN

The Digits dataset in Sklearn comprises 1797 small 8x8 images: each image is a handwritten digit, which is also the case for the MNIST dataset. Listing 6.16 displays the contents of load_digits1.py that illustrates how to plot the Digits dataset.

LISTING 6.16: load_digits1.py

```
from sklearn import datasets

# Load in the 'digits' data
digits = datasets.load_digits()

# Print the 'digits' data
print(digits)
```

Listing 6.16 is very straightforward: after importing the datasets module, the variable digits is initialized with the contents of the Digits dataset.

The `print()` statement displays the contents of the `digits` variable, which is displayed here:

```
{images': array(
     [[[0.,   0.,   5., ...,       1.,    0.,   0.],
       [0.,   0.,  13., ...,  15.,   5.,   0.],
       [0.,   3.,  15., ...,  11.,   8.,   0.],
       ...,
       [0.,   4.,  11., ...,  12.,   7.,   0.],
       [0.,   2.,  14., ...,  12.,   0.,   0.],
       [0.,   0.,   6., ...,   0.,   0.,   0.]]),
'target': array([0, 1, 2, ..., 8, 9, 8]), 'frame': None,
'feature_names': ['pixel_0_0', 'pixel_0_1', 'pixel_0_2',
'pixel_0_3', 'pixel_0_4', 'pixel_0_5', 'pixel_0_6',
'pixel_0_7', 'pixel_1_0', 'pixel_1_1', 'pixel_1_2',
'pixel_1_3', 'pixel_1_4', 'pixel_1_5', 'pixel_1_6',
'pixel_1_7', 'pixel_2_0', 'pixel_2_1', 'pixel_2_2',
'pixel_2_3', 'pixel_2_4', 'pixel_2_5', 'pixel_2_6',
'pixel_2_7', 'pixel_3_0', 'pixel_3_1', 'pixel_3_2',
'pixel_3_3', 'pixel_3_4', 'pixel_3_5', 'pixel_3_6',
'pixel_3_7', 'pixel_4_0', 'pixel_4_1', 'pixel_4_2',
'pixel_4_3', 'pixel_4_4', 'pixel_4_5', 'pixel_4_6',
'pixel_4_7', 'pixel_5_0', 'pixel_5_1', 'pixel_5_2',
'pixel_5_3', 'pixel_5_4', 'pixel_5_5', 'pixel_5_6',
'pixel_5_7', 'pixel_6_0', 'pixel_6_1', 'pixel_6_2',
'pixel_6_3', 'pixel_6_4', 'pixel_6_5', 'pixel_6_6',
'pixel_6_7', 'pixel_7_0', 'pixel_7_1', 'pixel_7_2',
'pixel_7_3', 'pixel_7_4', 'pixel_7_5', 'pixel_7_6',
'pixel_7_7'], 'target_names': array([0, 1, 2, 3, 4, 5, 6,
7, 8, 9]), 'images': array([[[ 0.,  0.,  5., ...,  1.,  0.,
0.],
        [ 0.,  0., 13., ..., 15.,  5.,  0.],
        [ 0.,  3., 15., ..., 11.,  8.,  0.],
// data omitted for brevity
])}
```

Listing 6.17 displays the contents of `load_digits2.py` that illustrates how to plot one of the `Digits` datasets (which you can change in order to display a different digit).

LISTING 6.17: load_digits2.py

```
from sklearn.datasets import load_digits
from matplotlib import pyplot as plt

digits = load_digits()
#set interpolation='none'

fig = plt.figure(figsize=(3, 3))
plt.imshow(digits['images'][66], cmap="gray",
interpolation='none')
plt.show()
```

Listing 6.17 imports the `load_digits` class from `Sklearn` in order to initialize the variable `digits` with the contents of the `Digits` dataset that is

available in Sklearn. The next portion of Listing 6.17 initializes the variable fig and invokes the method imshow() of the plt class in order to display a number in the Digits dataset.

Figure 6.16 displays a plot of one of the digits in the Digits dataset based on the code in Listing 6.17.

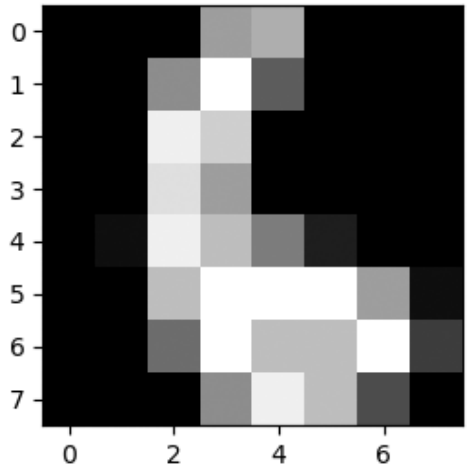

FIGURE 6.16 A digit in the Sklearn digits dataset.

Listing 6.18 displays the contents of sklearn_digits.py that illustrates how to access the Digits dataset in Sklearn.

LISTING 6.18: sklearn_digits.py

```
from sklearn import datasets

digits = datasets.load_digits()
print("digits shape:",digits.images.shape)
print("data   shape:",digits.data.shape)

n_samples, n_features = digits.data.shape
print("(samples,features):", (n_samples, n_features))

import matplotlib.pyplot as plt
#plt.imshow(digits.images[-1], cmap=plt.cm.gray_r)
#plt.show()

plt.imshow(digits.images[0], cmap=plt.cm.binary, interpolat
ion='nearest')
plt.show()
```

Listing 6.18 starts with one import statement followed by the variable digits that contains the Digits dataset. The output from Listing 6.18 is here:

```
digits shape: (1797, 8, 8)
data   shape: (1797, 64)
(samples,features): (1797, 64)
```

Figure 6.17 displays the images in the `Digits` dataset based on the code in Listing 6.18.

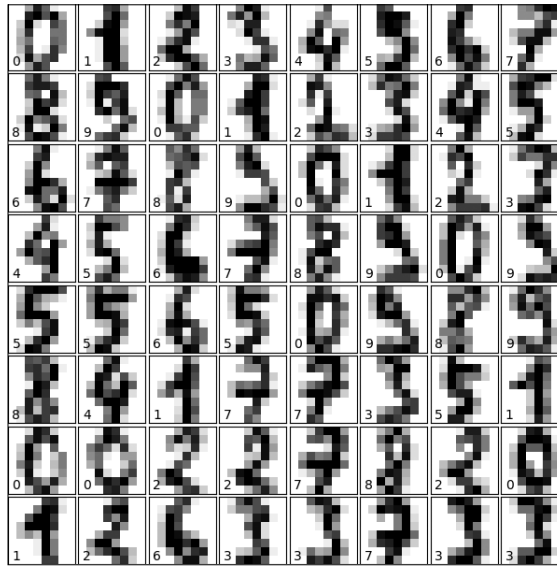

FIGURE 6.17 The digits in the `Digits` dataset.

THE `IRIS` DATASET IN `SKLEARN` (1)

Listing 6.19 displays the contents of `sklearn_iris.py` that illustrates how to access the `Iris` dataset in `Sklearn`.

In addition to support for machine learning algorithms, `Sklearn` provides various built-in datasets that you can access with literally one line of code. In fact, Listing 6.19 displays the contents of `sklearn_iris1.py` that illustrates how you can easily load the `Iris` dataset and display its contents.

LISTING 6.19: sklearn_iris1.py

```
import numpy as np
from sklearn.datasets import load_iris

iris = load_iris()

print("=> iris keys:")
for key in iris.keys():
  print(key)
print()

#print("iris dimensions:")
```

```
#print(iris.shape)
#print()

print("=> iris feature names:")
for feature in iris.feature_names:
  print(feature)
print()

X = iris.data[:, [2, 3]]
y = iris.target
print('=> Class labels:', np.unique(y))
print()

print("=> target:")
print(iris.target)
print()

print("=> all data:")
print(iris.data)
```

Listing 6.19 contains several import statements and then initializes the variable `iris` with the `Iris` dataset. Next, a for loop displays the keys in the dataset, followed by another for loop that displays the feature names.

The next portion of Listing 6.19 initializes the variable `X` with the feature values in columns 2 and 3, and then initializes the variable `y` with the values of the target column.

Launch the code in Listing 6.19 and you will see the following output (truncated to save space):

```
Pandas df1:

=> iris keys:
data
target
target_names
DESCR
feature_names
filename

=> iris feature names:
sepal length (cm)
sepal width (cm)
petal length (cm)
petal width (cm)

=> Class labels: [0 1 2]

=> x_min: 0.5 x_max: 7.4
=> y_min: -0.4 y_max: 3.0

=> target:
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
```

```
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2
2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2
 2 2]

=> all data:
[[5.1 3.5 1.4 0.2]
 [4.9 3.  1.4 0.2]
 [4.7 3.2 1.3 0.2]
 // details omitted for brevity
 [6.5 3.  5.2 2. ]
 [6.2 3.4 5.4 2.3]
 [5.9 3.  5.1 1.8]]
```

Sklearn, Pandas, and the Iris Dataset

Listing 6.20 displays the contents of `pandas_iris.py` that illustrates how to load the contents of the `Iris` dataset (from `Sklearn`) into a `Pandas` dataframe.

LISTING 6.20: pandas_iris.py

```
import numpy as np
import pandas as pd
from sklearn.datasets import load_iris

iris = load_iris()

print("=> IRIS feature names:")
for feature in iris.feature_names:
  print(feature)
print()

# Create a dataframe with the feature variables
df = pd.DataFrame(iris.data, columns=iris.feature_names)

print("=> number of rows:")
print(len(df))
print()

print("=> number of columns:")
print(len(df.columns))
print()

print("=> number of rows and columns:")
print(df.shape)
print()

print("=> number of elements:")
print(df.size)
print()
```

```
print("=> IRIS details:")
print(df.info())
print()

print("=> top five rows:")
print(df.head())
print()

X = iris.data[:, [2, 3]]
y = iris.target
print('=> Class labels:', np.unique(y))
```

Listing 6.20 contains several `import` statements and then initializes the variable `iris` with the `Iris` dataset. Next, a `for` loop displays the feature names. The next code snippet initializes the variable `df` as a `Pandas dataframe` that contains the data from the `Iris` dataset.

The next block of code invokes some attributes and methods of a `Pandas dataframe` to display the number of rows, columns, and elements in the dataframe, as well as the details of the `Iris` dataset, the first five rows, and the unique labels in the `Iris` dataset. Launch the code in Listing 6.20 and you will see the following output:

```
=> IRIS feature names:
sepal length (cm)
sepal width (cm)
petal length (cm)
petal width (cm)

=> number of rows:
150

=> number of columns:
4

=> number of rows and columns:
(150, 4)

=> number of elements:
600

=> IRIS details:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 4 columns):
sepal length (cm)    150 non-null float64
sepal width (cm)     150 non-null float64
petal length (cm)    150 non-null float64
petal width (cm)     150 non-null float64
dtypes: float64(4)
memory usage: 4.8 KB
None
```

```
=> top five rows:
   sepal length (cm)  sepal width (cm)  petal length (cm)
petal width (cm)
0                5.1               3.5                1.4
0.2
1                4.9               3.0                1.4
0.2
2                4.7               3.2                1.3
0.2
3                4.6               3.1                1.5
0.2
4                5.0               3.6                1.4
0.2

=> Class labels: [0 1 2]
```

THE IRIS DATASET IN SKLEARN (2)

The Iris dataset in Sklearn consists of the lengths of three different types of Iris-based petals and sepals: Setosa, Versicolor, and Virginica. These numeric values are stored in a 150x4 NumPy.ndarray.

Note that the rows in the Iris dataset are the sample images, and the columns consist of the values for the Sepal Length, Sepal Width, Petal Length, and Petal Width of each image. Listing 6.21 displays the contents of sklearn_iris2.py that illustrates how to display detailed information about the Iris dataset and a chart that displays the distributions of the four features.

LISTING 6.21: sklearn_iris2.py

```
from sklearn import datasets
from sklearn.model_selection import train_test_split

iris = datasets.load_iris()
data = iris.data

print("iris data shape:  ",data.shape)
print("iris target shape:",iris.target.shape)
print("first 5 rows iris:")
print(data[0:5])
print("keys:",iris.keys())
print("")

n_samples, n_features = iris.data.shape
print('Number of samples: ', n_samples)
print('Number of features:', n_features)
print("")

print("sepal length/width and petal length/width:")
print(iris.data[0])

import numpy as np
np.bincount(iris.target)
```

```
print("target names:",iris.target_names)

print("mean: %s " % data.mean(axis=0))
print("std:  %s " % data.std(axis=0))

#print("mean: %s " % data.mean(axis=1))
#print("std:  %s " % data.std(axis=1))

# load the data into train and test datasets:
X_train, X_test, y_train, y_test = train_test_split(iris.
data, iris.target, random_state=0)

from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(X_train)

# rescale the train datasest:
X_train_scaled = scaler.transform(X_train)
print("X_train_scaled shape:",X_train_scaled.shape)

print("mean : %s " % X_train_scaled.mean(axis=0))
print("standard deviation : %s " % X_train_scaled.
std(axis=0))

import matplotlib.pyplot as plt

x_index = 3
colors = ['blue', 'red', 'green']

for label, color in zip(range(len(iris.target_names)),
colors):
  plt.hist(iris.data[iris.target==label, x_index],
           label=iris.target_names[label],
           color=color)

plt.xlabel(iris.feature_names[x_index])
plt.legend(loc='upper right')
plt.show()
```

Listing 6.21 starts with an `import` statement followed by the variables `iris` and `data`, where the latter contains the `Iris` dataset. The first half of Listing 6.21 consists of self-explanatory code, such as displaying the number of images and the number of features in the `Iris` dataset.

The second portion of Listing 6.21 imports the `StandardScaler` class in `Sklearn`, which rescales each value in `X_train` by subtracting the mean and then dividing by the standard deviation. The final block of code in Listing 6.21 generates a histogram that displays some of the images in the `Iris` dataset. The output from Listing 6.26 is here:

```
iris data shape:   (150, 4)
iris target shape: (150,)
first 5 rows iris:
[[5.1 3.5 1.4 0.2]
 [4.9 31.4 0.2]
```

```
 [4.7 3.2 1.3 0.2]
 [4.6 3.1 1.5 0.2]
 [53.6 1.4 0.2]]
keys: dict_keys(['target', 'target_names', 'data',
'feature_names', 'DESCR'])

Number of samples:  150
Number of features: 4

sepal length/width and petal length/width:
[5.1 3.5 1.4 0.2]
target names: ['setosa' 'versicolor' 'virginica']
mean: [5.84333333 3.054      3.75866667 1.19866667]
std:  [0.82530129 0.43214658 1.75852918 0.76061262]
X_train_scaled shape: (112, 4)
mean : [ 1.21331516e-15 -4.41115398e-17  7.13714802e-17
2.57730345e-17]
standard deviation : [1. 1. 1. 1.]
```

Figure 6.18 displays the images in the `Iris` dataset based on the code in Listing 6.21.

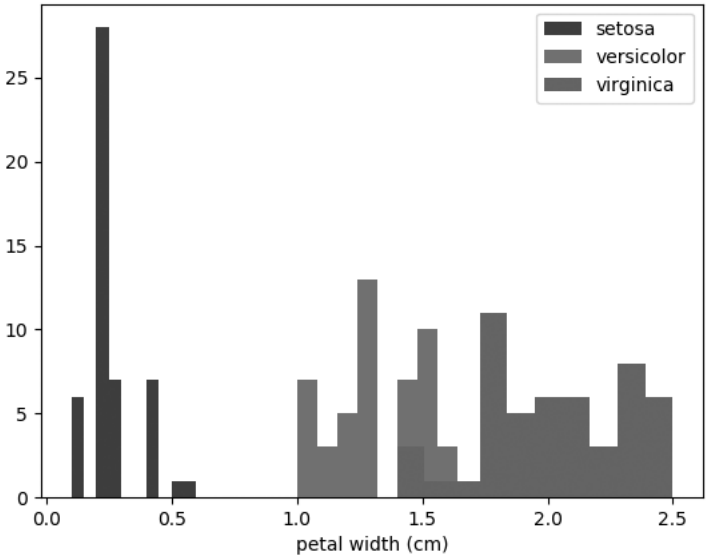

FIGURE 6.18 The Iris dataset.

THE FACES DATASET IN SKLEARN (OPTIONAL)

The Olivetti faces dataset contains a set of face images that were taken between April 1992 and April 1994 at AT&T Laboratories Cambridge. As you will see in Listing 6.22, the `Sklearn.datasets.fetch_olivetti_faces` function is the data fetching and caching function that downloads the data archive from AT&T.

Listing 6.22 displays the contents of `sklearn_faces.py` that displays the contents of the `Faces` dataset in `Sklearn`.

LISTING 6.22: sklearn_faces.py

```
import Sklearn
from sklearn.datasets import fetch_olivetti_faces

faces = fetch_olivetti_faces()

import matplotlib.pyplot as plt

# display figures in inches
fig = plt.figure(figsize=(6, 6))
fig.subplots_adjust(left=0, right=1, bottom=0, top=1,
hspace=0.05, wspace=0.05)

# plot the faces:
for i in range(64):
  ax = fig.add_subplot(8, 8, i + 1, xticks=[], yticks=[])
  ax.imshow(faces.images[i], cmap=plt.cm.bone,
interpolation='nearest')

plt.show()
```

Listing 6.22 starts with `import` statements and then initializes the variable `faces` with the contents of the `Olivetti Faces` dataset. The next portion of Listing 6.22 contains some plot-related code, followed by a `for` loop that displays 64 images in an 8x8 grid pattern (similar to an earlier code sample).

Launch Listing 6.22 and you will see the plotted image as shown in Figure 6.19.

FIGURE 6.19 The Sklearn faces dataset.

This concludes the portion of the chapter pertaining to Sklearn. Now let's turn our attention to Seaborn, which is a very nice data visualization package for Python.

WORKING WITH SEABORN

Seaborn is a Python package for data visualization that also provides a high-level interface to Matplotlib. Seaborn is easier to work with than Matplotlib, and actually extends Matplotlib, but keep in mind that Seaborn is not as powerful as Matplotlib.

Seaborn addresses two challenges of Matplotlib. The first involves the default Matplotlib parameters. Seaborn works with different parameters, which provides greater flexibility than the default rendering of Matplotlib plots. Seaborn addresses the limitations of the Matplotlib default values for features such as colors, tick marks on the upper and right axes, and style (among others).

In addition, Seaborn makes it easier to plot entire dataframes (somewhat like pandas) than doing so in Matplotlib. Nevertheless, since Seaborn extends Matplotlib, knowledge of the latter is advantageous and will simplify your learning curve.

Features of Seaborn

Some of the features of Seaborn include:

- scale Seaborn plots
- set the plot style
- set the figure size
- rotate label text
- set xlim or ylim
- set log scale
- add titles

Some useful methods:

- plt.xlabel()
- plt.ylabel()
- plt.annotate()
- plt.legend()
- plt.ylim()
- plt.savefig()

Seaborn supports various built-in datasets, just like NumPy and Pandas, including the Iris dataset and the Titanic dataset, both of which you will see in subsequent sections. As a starting point, the three-line code sample in the next section shows you how to display the rows in the built-in "tips" dataset.

SEABORN BUILT-IN DATASETS

Listing 6.23 displays the contents of `seaborn_tips.py` that illustrates how to read the `tips` dataset into a dataframe and display the first five rows of the dataset.

LISTING 6.23: seaborn_tips.py

```
import seaborn as sns
df = sns.load_dataset("tips")
print(df.head())
```

Listing 6.23 is very simple: after importing `seaborn`, the variable `df` is initialized with the data in the built-in dataset `tips`, and the `print()` statement displays the first five rows of `df`. Note that the `load_dataset()` API searches for online or built-in datasets. The output from Listing 6.23 is here:

```
   total_bill   tip     sex smoker  day    time  size
0       16.99  1.01  Female     No  Sun  Dinner     2
1       10.34  1.66    Male     No  Sun  Dinner     3
2       21.01  3.50    Male     No  Sun  Dinner     3
3       23.68  3.31    Male     No  Sun  Dinner     2
4       24.59  3.61  Female     No  Sun  Dinner     4
```

THE IRIS DATASET IN SEABORN

Listing 6.24 displays the contents of `seaborn_iris.py` that illustrates how to plot the `Iris` dataset.

LISTING 6.24: seaborn_iris.py

```
import seaborn as sns
import Matplotlib.pyplot as plt

# Load iris data
iris = sns.load_dataset("iris")

# Construct iris plot
sns.swarmplot(x="species", y="petal_length", data=iris)

# Show plot
plt.show()
```

Listing 6.24 imports `seaborn` and `Matplotlib.pyplot` and then initializes the variable `iris` with the contents of the built-in `Iris` dataset. Next, the `swamplot()` API displays a graph with the horizontal axis labeled `species` and the vertical axis labeled `petal_length`, and the displayed points are from the `Iris` dataset.

Figure 6.20 displays the images in the `Iris` dataset based on the code in Listing 6.24.

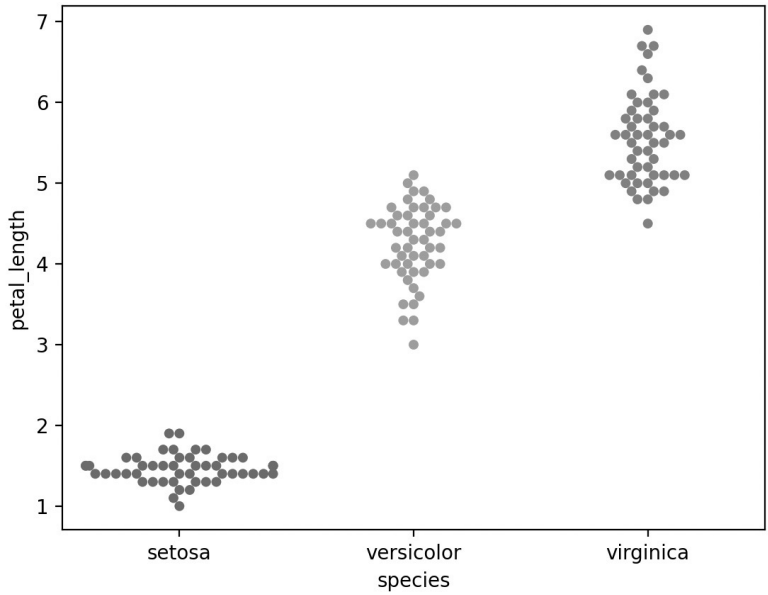

FIGURE 6.20 The `Iris` dataset.

THE `TITANIC` DATASET IN `SEABORN`

Listing 6.25 displays the contents of `seaborn_titanic_plot.py` that illustrates how to plot the `Titanic` dataset.

LISTING 6.25: seaborn_titanic_plot.py

```
import matplotlib.pyplot as plt
import seaborn as sns

titanic = sns.load_dataset("titanic")
g = sns.factorplot("class", "survived", "sex",
data=titanic, kind="bar", palette="muted", legend=False)

plt.show()
```

Listing 6.25 contains the same `import` statements as Listing 6.24, and then initializes the variable `titanic` with the contents of the built-in `Titanic` dataset. Next, the `factorplot()` API displays a graph with dataset attributes that are listed in the API invocation.

Figure 6.21 displays a plot of the data in the `Titanic` dataset based on the code in Listing 6.25.

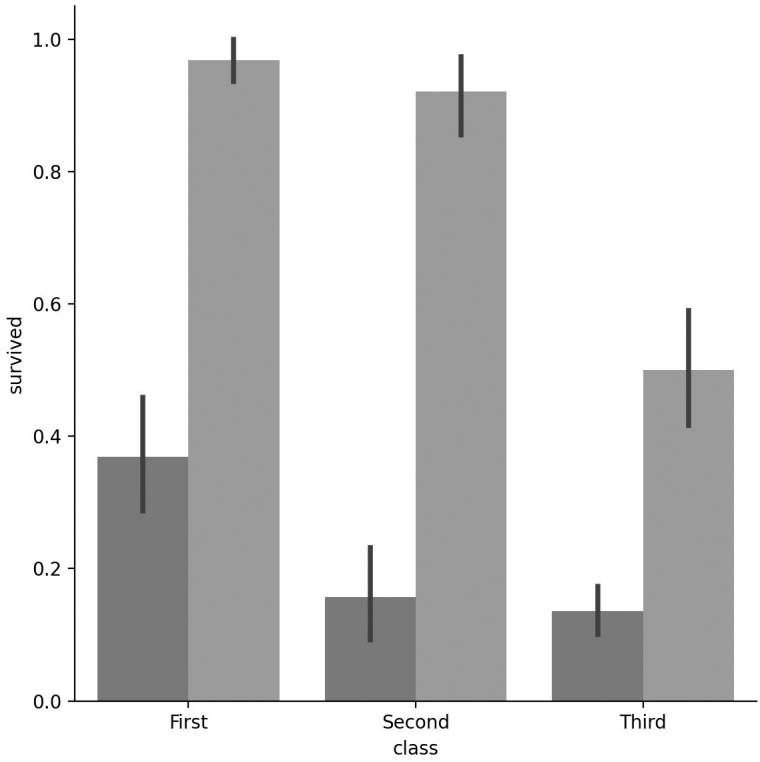

FIGURE 6.21 A histogram of the Titanic dataset.

EXTRACTING DATA FROM THE `TITANIC` DATASET IN `SEABORN` (1)

Listing 6.26 displays the contents of `seaborn_titanic.py` that illustrates how to extract subsets of data from the `Titanic` dataset.

LISTING 6.26: seaborn_titanic.py

```
import matplotlib.pyplot as plt
import seaborn as sns

titanic = sns.load_dataset("titanic")
print("titanic info:")
titanic.info()

print("first five rows of titanic:")
print(titanic.head())

print("first four ages:")
print(titanic.loc[0:3,'age'])

print("fifth passenger:")
print(titanic.iloc[4])
```

```
#print("first five ages:")
#print(titanic['age'].head())

#print("first five ages and gender:")
#print(titanic[['age','sex']].head())

#print("descending ages:")
#print(titanic.sort_values('age', ascending = False).
head())

#print("older than 50:")
#print(titanic[titanic['age'] > 50])

#print("embarked (unique):")
#print(titanic['embarked'].unique())

#print("survivor counts:")
#print(titanic['survived'].value_counts())

#print("counts per class:")
#print(titanic['pclass'].value_counts())

#print("max/min/mean/median ages:")
#print(titanic['age'].max())
#print(titanic['age'].min())
#print(titanic['age'].mean())
#print(titanic['age'].median())
```

Listing 6.26 contains the same `import` statements as Listing 6.25, and then initializes the variable `titanic` with the contents of the built-in `Titanic` dataset. The next portion of Listing 6.26 displays various aspects of the `Titanic` dataset, such as its structure, the first five rows, the first four ages, and the details of the fifth passenger.

As you can see, there is a large block of "commented out" code that you can uncomment in order to see the associated output, such as age, gender, persons over 50, unique rows, and so forth. The output from Listing 6.26 is here:

```
#print(titanic['age'].mean())
titanic info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 15 columns):
survived       891 non-null int64
pclass         891 non-null int64
sex            891 non-null object
age            714 non-null float64
sibsp          891 non-null int64
parch          891 non-null int64
fare           891 non-null float64
embarked       889 non-null object
class          891 non-null category
who            891 non-null object
adult_male     891 non-null bool
```

```
deck           203 non-null category
embark_town    889 non-null object
alive          891 non-null object
alone          891 non-null bool
dtypes: bool(2), category(2), float64(2), int64(4),
object(5)
memory usage: 80.6+ KB
first five rows of titanic:
   survived  pclass     sex   age  sibsp  parch     fare
embarked  class  \
0         0       3    male  22.0      1      0   7.2500
S  Third
1         1       1  female  38.0      1      0  71.2833
C  First
2         1       3  female  26.0      0      0   7.9250
S  Third
3         1       1  female  35.0      1      0  53.1000
S  First
4         0       3    male  35.0      0      0   8.0500
S  Third

     who  adult_male deck  embark_town alive  alone
0    man        True  NaN  Southampton    no  False
1  woman       False    C    Cherbourg   yes  False
2  woman       False  NaN  Southampton   yes   True
3  woman       False    C  Southampton   yes  False
4    man        True  NaN  Southampton    no   True
first four ages:
0    22.0
1    38.0
2    26.0
3    35.0
Name: age, dtype: float64
fifth passenger:
survived                 0
pclass                   3
sex                    male
age                      35
sibsp                     0
parch                     0
fare                   8.05
embarked                  S
class                 Third
who                     man
adult_male             True
deck                    NaN
embark_town    Southampton
alive                   no
alone                 True
Name: 4, dtype: object
counts per class:
3    491
1    216
2    184
Name: pclass, dtype: int64
max/min/mean/median ages:
```

```
80.0
0.42
29.69911764705882
28.0
```

EXTRACTING DATA FROM THE TITANIC DATASET IN SEABORN (2)

Listing 6.27 displays the contents of seaborn_titanic2.py that illustrates how to extract subsets of data from the Titanic dataset.

LISTING 6.27: seaborn_titanic2.py

```
import matplotlib.pyplot as plt
import seaborn as sns

titanic = sns.load_dataset("titanic")

# Returns a scalar
# titanic.ix[4, 'age']
print("age:",titanic.at[4, 'age'])

# Returns a Series of name 'age', and the age values
associated
# to the index labels 4 and 5
# titanic.ix[[4, 5], 'age']
print("series:",titanic.loc[[4, 5], 'age'])

# Returns a Series of name '4', and the age and fare values
# associated to that row.
# titanic.ix[4, ['age', 'fare']]
print("series:",titanic.loc[4, ['age', 'fare']])

# Returns a DataFrame with rows 4 and 5, and columns 'age'
and 'fare'
# titanic.ix[[4, 5], ['age', 'fare']]
print("dataframe:",titanic.loc[[4, 5], ['age', 'fare']])

query = titanic[
    (titanic.sex == 'female')
    & (titanic['class'].isin(['First', 'Third']))
    & (titanic.age > 30)
    & (titanic.survived == 0)
]
print("query:",query)
```

Listing 6.27 contains the same import statements as Listing 6.26, and then initializes the variable titanic with the contents of the built-in Titanic dataset. The next code snippet displays the age of the passenger with index 4 in the dataset (which equals 35).

The following code snippet displays the ages of passengers with index values 4 and 5 in the dataset:

```
print("series:",titanic.loc[[4, 5], 'age'])
```

The next snippet displays the age and fare of the passenger with index 4 in the dataset, followed by another code snippet that displays the age and fare of the passengers with index 4 and index 5 in the dataset.

The final portion of Listing 6.27 is the most interesting part: it defines a variable `query` as shown here:

```
query = titanic[
    (titanic.sex == 'female')
    & (titanic['class'].isin(['First', 'Third']))
    & (titanic.age > 30)
    & (titanic.survived == 0)
]
```

The preceding code block will retrieve the female passengers who are in either first class or third class, and who are also over 30, and who did not survive the accident. The entire output from Listing 6.27 is here:

```
age: 35.0
series: 4    35.0
5     NaN
Name: age, dtype: float64
series: age       35
fare    8.05
Name: 4, dtype: object
dataframe:     age    fare
4  35.0  8.0500
5   NaN  8.4583
query:      survived  pclass     sex   age  sibsp  parch
fare embarked  class  \
18          0       3  female  31.0      1      0  18.0000
S  Third
40          0       3  female  40.0      1      0   9.4750
S  Third
132         0       3  female  47.0      1      0  14.5000
S  Third
167         0       3  female  45.0      1      4  27.9000
S  Third
177         0       1  female  50.0      0      0  28.7125
C  First
254         0       3  female  41.0      0      2  20.2125
S  Third
276         0       3  female  45.0      0      0   7.7500
S  Third
362         0       3  female  45.0      0      1  14.4542
C  Third
396         0       3  female  31.0      0      0   7.8542
S  Third
503         0       3  female  37.0      0      0   9.5875
S  Third
610         0       3  female  39.0      1      5  31.2750
S  Third
638         0       3  female  41.0      0      5  39.6875
S  Third
```

```
657         0       3   female  32.0      1       1   15.5000
Q   Third
678         0       3   female  43.0      1       6   46.9000
S   Third
736         0       3   female  48.0      1       3   34.3750
S   Third
767         0       3   female  30.5      0       0    7.7500
Q   Third
885         0       3   female  39.0      0       5   29.1250
Q   Third
```

VISUALIZING A PANDAS DATASET IN SEABORN

Listing 6.28 displays the contents of `pandas_seaborn.py` that illustrates how to display a `Pandas` dataset in `Seaborn`.

LISTING 6.28: pandas_seaborn.py

```
import pandas as pd
import random
import matplotlib.pyplot as plt
import seaborn as sns

df = pd.DataFrame()

df['x'] = random.sample(range(1, 100), 25)
df['y'] = random.sample(range(1, 100), 25)

print("top five elements:")
print(df.head())

# display a density plot
#sns.kdeplot(df.y)

# display a density plot
#sns.kdeplot(df.y, df.x)

#sns.distplot(df.x)

# display a histogram
#plt.hist(df.x, alpha=.3)
#sns.rugplot(df.x)

# display a boxplot
#sns.boxplot([df.y, df.x])

# display a violin plot
#sns.violinplot([df.y, df.x])

# display a heatmap
#sns.heatmap([df.y, df.x], annot=True, fmt="d")

# display a cluster map
#sns.clustermap(df)
```

```
# display a scatterplot of the data points
sns.lmplot('x', 'y', data=df, fit_reg=False)
plt.show()
```

Listing 6.28 contains several familiar `import` statements, followed by the initialization of the `Pandas` variable `df` as a `Pandas dataframe`. The next two code snippets initialize the columns and rows of the dataframe, and the `print()` statement displays the first five rows.

For your convenience, Listing 6.28 contains an assortment of "commented out" code snippets that use `Seaborn` in order to render a density plot, a histogram, a boxplot, a violin plot, a heatmap, and a cluster. Uncomment the portions that interest you in order to see the associated plot. The output from Listing 6.28 is here:

```
top five elements:
    x   y
0  52  34
1  31  47
2  23  18
3  34  70
4  71   1
```

Figure 6.22 displays a plot of the data in the `Titanic` dataset based on the code in Listing 6.28.

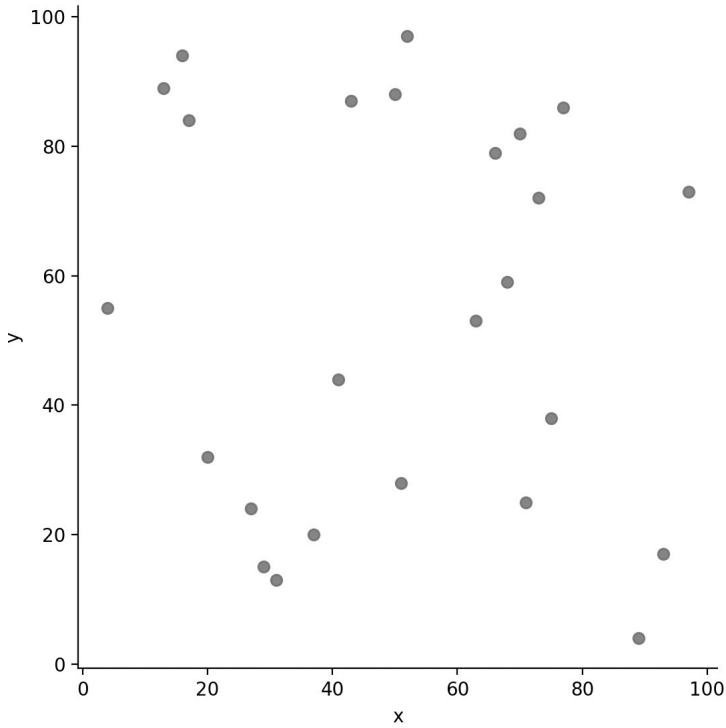

FIGURE 6.22 A `Pandas dataframe` displayed via `Seaborn`.

DATA VISUALIZATION IN PANDAS

Although Matplotlib and Seaborn are often the "go to" Python libraries for data visualization, you can also use Pandas for such tasks.

Listing 6.29 displays the contents pandas_viz1.py that illustrates how to render various types of charts and graphs using Pandas and Matplotlib.

LISTING 6.29: pandas_viz1.py

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

df = pd.DataFrame(np.random.rand(16,3),
columns=['X1','X2','X3'])
print("First 5 rows:")
print(df.head())
print()

print("Diff of first 5 rows:")
print(df.diff().head())
print()

# bar chart:
#ax = df.plot.bar()

# horizontal stacked bar chart:
#ax = df.plot.barh(stacked=True)

# vertical stacked bar chart:
ax = df.plot.bar(stacked=True)

# stacked area graph:
#ax = df.plot.area()

# non-stacked area graph:
#ax = df.plot.area(stacked=False)

#plt.show(ax)
```

Listing 6.29 initializes the dataframe df with a 16x3 matrix of random numbers, followed by the contents of df. The bulk of Listing 6.29 contains code snippets for generating a bar chart, a horizontal stacked bar chart, a vertical stacked bar chart, a stacked area graph, and a non-stacked area graph. You can uncomment the individual code snippet that displays the graph of your choice with the contents of df. Launch the code in Listing 6.29 and you will see the following output:

```
First 5 rows:
         X1        X2        X3
0  0.051089  0.357183  0.344414
1  0.800890  0.468372  0.800668
```

```
2  0.492981  0.505133  0.228399
3  0.461996  0.977895  0.471315
4  0.033209  0.411852  0.347165

Diff of first 5 rows:
         X1        X2        X3
0       NaN       NaN       NaN
1  0.749801  0.111189  0.456255
2 -0.307909  0.036760 -0.572269
3 -0.030984  0.472762  0.242916
4 -0.428787 -0.566043 -0.124150
```

WHAT IS BOKEH?

`Bokeh` is an open-source project that depends on `Matplotlib` as well as `Sklearn`. As you will see in the subsequent code sample, `Bokeh` generates an HTML Web page that is based on Python code, and then launches that Web page in a browser. `Bokeh` and D3.js (which is a JavaScript layer of abstraction over SVG) both provide elegant visualization effects that support animation effects and user interaction.

`Bokeh` enables the rapid creation of statistical visualization, and it works with other tools with as Python Flask and Django. In addition to Python, `Bokeh` supports Julia, Lua, and R (JSON files are generated instead of HTML Web pages).

Listing 6.30 displays the contents `bokeh_gtrig.py` that illustrates how to create a graphics effect using various `Bokeh` APIs.

LISTING 6.30: bokeh_trig.py

```
# pip3 install bokeh
from bokeh.plotting import figure, output_file, show
from bokeh.layouts import column
import bokeh.colors as colors
import numpy as np
import math

deltaY = 0.01
maxCount = 150
width  = 800
height = 400
band_width = maxCount/3

x = np.arange(0, math.pi*3, 0.05)
y1 = np.sin(x)
y2 = np.cos(x)

white = colors.RGB(255,255,255)

fig1 = figure(plot_width = width, plot_height = height)

for i in range(0,maxCount):
```

```
  rgb1 = colors.RGB(i*255/maxCount, 0, 0)
  rgb2 = colors.RGB(i*255/maxCount, i*255/maxCount, 0)
  fig1.line(x, y1-i*deltaY,line_width = 2, line_color =
rgb1)
  fig1.line(x, y2-i*deltaY,line_width = 2, line_color =
rgb2)

for i in range(0,maxCount):
  rgb1 = colors.RGB(0, 0, i*255/maxCount)
  rgb2 = colors.RGB(0, i*255/maxCount, 0)
  fig1.line(x, y1+i*deltaY,line_width = 2, line_color =
rgb1)
  fig1.line(x, y2+i*deltaY,line_width = 2, line_color =
rgb2)
  if (i % band_width == 0):
    fig1.line(x, y1+i*deltaY,line_width = 5, line_color =
white)

show(fig1)
```

Listing 6.30 starts with a commented out `pip3` code snippet that you can launch from the command line in order to install `Bokeh` (in case you haven't done so already).

The next code block contains several `Bokeh`-related statements as well as `NumPy` and `Math`.

Notice that the variable `white` is defined as an (R,G,B) triple of integers, which represents the red, green, and blue components of a color. In particular, (255,255,255) represents the color white (check online if you are unfamiliar with RGB). The next portion of Listing 6.30 initializes some scalar variables that are used in the two `for` loops that are in the second half of Listing 6.30.

Next, the `NumPy` variable `x` is a range of values from 0 to `math.PI/3`, with an increment of 0.05 between successive values. Then the `NumPy` variables `y1` and `y2` are defined as the sine and cosine values, respectively, of the values in `x`. The next code snippet initializes the variable `fig1` that represents a context in which the graphics effects will be rendered. This completes the initialization of the variables that are used in the two `for` loops.

The next portion of Listing 6.30 contains the first `for` loop that creates a gradient-like effect by defining (R,G,B) triples whose values are based partially on the value of the loop variable `i`. For example, the variable `rgb1` ranges in a linear fashion from (0,0,0) to (255,0,0), which represent the colors black and red, respectively. The variable `rgb2` ranges in a linear fashion from (0,0,0) to (255,255,0), which represent the colors black and yellow, respectively. The next portion of the `for` loop contains two invocations of the `fig1.line()` API that renders a sine wave and a cosine wave in the context variable `fig1`.

The second `for` loop is similar to the first `for` loop: the main difference is that the variable `rgb1` varies from black to blue, and the variable `rgb2` varies from black to green. The final code snippet in Listing 6.30 invokes the `show()` method that generates an HTML Web page (with the same prefix as the Python file) and then launches the Web page in a browser.

Figure 6.23 displays the graphics effect based on the code in Listing 6.30. If this image is displayed as black and white, launch the code from the command line and you will see the gradient-like effects in the image.

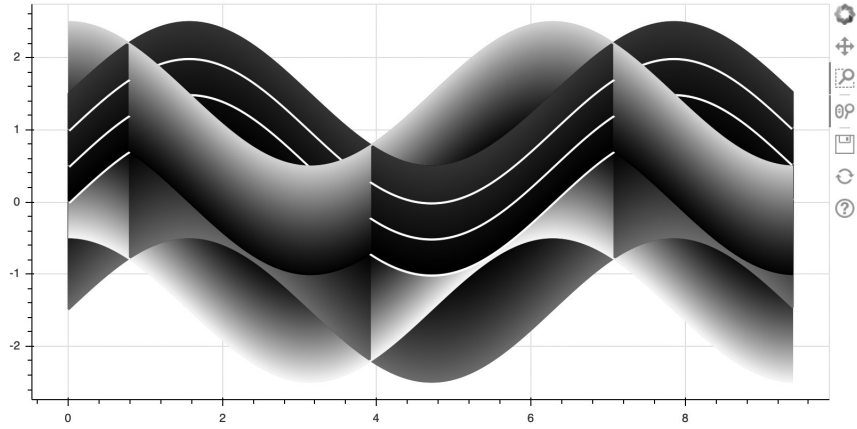

FIGURE 6.23 A `Bokeh` graphics sample.

SUMMARY

This chapter started with some basic features of `Matplotlib`, along with examples of plotting lines, histograms, and simple trigonometric functions.

You also learned about `Sklearn`, including examples of working with the `Digits` and `Iris` datasets, and also how to process images. In addition, you saw how to perform linear regression with `Sklearn`.

Then you were introduced to `Seaborn`, which is an extension of `Matplotlib`, and saw examples of plotting lines and histograms, and also how to plot a `Pandas dataframe` using `Seaborn`. Finally, you saw an example of rendering graphics in `Bokeh`.

APPENDIX

REGULAR EXPRESSIONS

This appendix introduces you to regular expressions, which is a very powerful language feature in Python. Since regular expressions are available in other programming languages (such as JavaScript and Java), the knowledge that you gain from the material in this appendix will be useful to you outside of Python.

Why would anyone be interested in learning regular expressions in a book for Python data analytics? The answer is threefold. First, you have already seen `Pandas` code samples in Chapter 2 that use regular expressions, which demonstrates that regular expressions are relevant to `Pandas`. Second, if you plan to use `Pandas` extensively or perhaps also work with NLP, then regular expressions will prove useful because of the ease with which you can solve certain types of tasks (such as removing HTML tags) with regular expressions. Third, the knowledge you gain from the material in this appendix will instantly transfer to other languages that support regular expressions.

This appendix contains a mixture of code blocks and complete code samples, with varying degrees of complexity, that are suitable for beginners as well as people who have had some exposure to regular expressions. In fact, you have probably used (albeit simple) regular expressions in a command line on a laptop, whether it be Windows, Unix, or Linux-based systems.

The first part of this appendix shows you how to define regular expressions with digits and letters (uppercase as well as lowercase), and also how to use character classes in regular expressions. You will also learn about character sets and character classes.

The second portion discusses the Python `re` module, which contains several useful methods, such as the `re.match()` method for matching groups of characters, the `re.search()` method to perform searches in character strings, and the `findAll()` method. You will also learn how to use character classes (and how to group them) in regular expressions.

The final portion of this appendix contains an assortment of code samples, such as modifying text strings, splitting text strings with the `re.split()` method, and substituting text strings with the `re.sub()` method.

As you read the code samples in this appendix, some concepts and facets of regular expressions might make you feel overwhelmed with the density of the material if you are a novice. However, practice and repetition will help you become comfortable with regular expressions.

WHAT ARE REGULAR EXPRESSIONS?

Regular expressions are referred to as REs, or regexes, or regex patterns, and they enable you to specify expressions that can match specific subsets of a string. For instance, you can define a regular expression to match a single character or digit, a telephone number, a zip code, or an email address. You can use metacharacters and character classes (defined in the next section) as part of regular expressions to search text documents for specific patterns. As you learn how to use RE you will find other ways to use them as well.

The `re` module (added in Python 1.5) provides Perl-style regular expression patterns. Note that earlier versions of Python provided the `regex` module that was removed in Python 2.5. The `re` module provides an assortment of methods (discussed later in this appendix) for searching text strings or replacing text strings, which is similar to the basic search and/or replace functionality that is available in word processors (but usually without regular expression support). The `re` module also provides methods for splitting text strings based on regular expressions.

Before delving into the methods in the `re` module, you need to learn about metacharacters and character classes, which are the topic of the next section.

METACHARACTERS IN PYTHON

Python supports a set of metacharacters, most of which are the same as the metacharacters in other scripting languages such as Perl, as well as programming languages such as JavaScript and Java. The complete list of metacharacters in Python is here:

```
. ^ $ * + ? { } [ ] \ | ( )
```

The meaning of the preceding metacharacters is here:

- `?` (matches 0 or 1): the expression `a?` matches the string `a` (but not `ab`)
- `*` (matches 0 or more): the expression `a*` matches the string `aaa` (but not `baa`)
- + (matches 1 or more): the expression a+ matches aaa (but not baa)
- `^` (beginning of line): the expression `^[a]` matches the string `abc` (but not `bc`)

- $ (end of line): [c]$ matches the string abc (but not cab)
- . (a single dot): matches any character (except newline)

Sometimes you need to match the metacharacters themselves rather than their representation, which can be done in two ways. The first way involves "escaping" their symbolic meaning with the backslash ("\") character. Thus, the sequences \?, *, \+, \^, \$, and \. represent the literal characters instead of their symbolic meaning. You can also "escape" the backslash character with the sequence "\\". If you have two consecutive backslash characters, you need an additional backslash for each of them, which means that "\\\\" is the "escaped" sequence for "\\".

The second way is to list the metacharacters inside a pair of square brackets. For example, [+?] treats the two characters "+" and "?" as literal characters instead of metacharacters. The second approach is obviously more compact and less prone to error (it's easy to forget a backslash in a long sequence of metacharacters). As you might surmise, the methods in the re module support metacharacters.

NOTE *The "^" character that is to the left (and outside) of a sequence in square brackets (such as ^[A-Z]) "anchors" the regular expression to the beginning of a line, whereas the "^" character that is the first character inside a pair of square brackets* negates *the regular expression (such as [^A-Z]) inside the square brackets.*

The interpretation of the "^" character in a regular expression depends on its location in a regular expression, as shown here:

- "^[a-z]" means any string that starts with any lowercase letter
- "[^a-z]" means any string that does *not* contain any lowercase letters
- "^[^a-z]" means any string that starts with anything *except* a lowercase letter
- "^[a-z]$" means a single lowercase letter
- "^[^a-z]$" means a single character (including digits) that is *not* a lowercase letter

As a quick preview of the re module that is discussed later in this appendix, the re.sub() method enables you to remove characters (including metacharacters) from a text string. For example, the following code snippet removes all occurrences of a forward slash ("/") and the plus sign ("+") from the variable str:

```
>>> import re
>>> str = "this string has a / and + in it"
>>> str2 = re.sub("[/]+","",str)
>>> print('original:',str)
original: this string has a / and + in it
>>> print('replaced:',str2)
replaced: this string has a and + in it
```

We can easily remove occurrences of other metacharacters in a text string by listing them inside the square brackets, just as we have done in the preceding code snippet.

Listing A.1 displays the contents of `RemoveMetaChars1.py` that illustrates how to remove other metacharacters from a line of text.

LISTING A.1: RemoveMetaChars1.py

```
import re

text1 = "meta characters ? and / and + and ."
text2 = re.sub("[/\.*?=+]+","",text1)

print('text1:',text1)
print('text2:',text2)
```

The regular expression in Listing A.1 might seem daunting if you are new to regular expressions, but let's demystify its contents by examining the entire expression and then the meaning of each character. First of all, the term `[/\.*?=+]` matches a forward slash ("/"), a dot (".), a question mark ("?"), an equals sign ("="), or a plus sign ("+"). Notice that the dot "." is preceded by a backslash character "\". Doing so "escapes" the meaning of the "." metacharacter (which matches anything except a newline character) and treats it as a literal character.

Thus, the term `[/\.*?=+]+` means "one or more occurrences of any of the metacharacters—treated as literal characters—inside the square brackets".

Consequently, the expression `re.sub("[/\.*?=+]+","",text1)` matches any occurrence of the previously listed metacharacters, and then replaces them with an empty string in the text string specified by the variable `text1`. The output from Listing A.1 is here:

```
text1: meta characters ? and / and + and .
text2: meta characters and and and
```

Later in this appendix you will learn about other functions in the `re` module that enable you to modify and split text strings.

CHARACTER SETS IN PYTHON

A single digit in base 10 is a number between 0 and 9 inclusive, which is represented by the sequence `[0-9]`. Similarly, a lowercase letter can be any letter between a and z, which is represented by the sequence `[a-z]`. An uppercase letter can be any letter between `A` and `Z`, which is represented by the sequence `[A-Z]`.

The following code snippets illustrate how to specify sequences of digits and sequences of character strings using a shorthand notation that is much simpler than specifying every matching digit:

- `[0-9]` matches a single digit
- `[0-9][0-9]` matches 2 consecutive digits
- `[0-9]{3}` matches 3 consecutive digits
- `[0-9]{2,4}` matches 2, 3, or 4 consecutive digits
- `[0-9]{5,}` matches 5 or more consecutive digits
- `^[0-9]+$` matches a string consisting solely of digits

You can define similar patterns using uppercase or lowercase letters in a way that is much simpler than explicitly specifying every lowercase letter or every uppercase letter:

- `[a-z][A-Z]` matches a single lowercase letter that is followed by 1 uppercase letter
- `[a-zA-Z]` matches any upper- or lowercase letter

Working with "^" and "\"

The purpose of the "^" character depends on its context in a regular expression. For example, the following expression matches a text string that starts with a digit:

```
^[0-9]
```

However, the following expression matches a text string that does *not* start with a digit because of the "^" metacharacter that is at the beginning of an expression in square brackets as well as the "^" metacharacter that is to the left (and outside) the expression in square brackets (which you learned in a previous note):

```
^[^0-9]
```

Thus, the "^" character inside a pair of matching square brackets ("[]") negates the expression immediately to its right that is also located inside the square brackets.

The backslash ("\") allows you to "escape" the meaning of a metacharacter. Consequently, a dot "." matches a single character (except for a newline character), whereas the sequence "\." matches the dot "." character. Other examples involving the backslash metacharacter are here:

- `\.H.*` matches the string `.Hello`
- `H.*` matches the string `Hello`
- `H.*\.` matches the string `Hello.`
- `.ell.` matches the string `Hello`
- `.*` matches the string `Hello`
- `\..*` matches the string `.Hello`

CHARACTER CLASSES IN PYTHON

Character classes are convenient expressions that are shorter and simpler than their "bare" counterparts that you saw in the previous section. Some convenient character sequences that express patterns of digits and letters are as follows:

- `\d` matches a single digit
- `\w` matches a single character (digit or letter)
- `\s` matches a single whitespace (space, newline, return, or tab)
- `\b` matches a boundary between a word and a nonword
- `\n`, `\r`, `\t` represent a newline, a return, and a tab, respectively
- `\` "escapes" any character

Based on the preceding definitions, `\d+` matches one or more digits and `\w+` matches one or more characters, both of which are more compact expressions than using character sets. In addition, we can reformulate the expressions in the previous section:

- `\d` is the same as `[0-9]` and `\D` is the same as `[^0-9]`
- `\s` is the same as `[ \t\n\r\f\v]` and it matches every whitespace character, whereas \S is the opposite (it matches `[^ \t\n\r\f\v]`)
- `\w` is the same as `[a-zA-Z0-9_]` and it matches any alphanumeric character, whereas `\W` is the opposite (it matches `[^a-zA-Z0-9_]`)

Additional examples are here:

- `\d{2}` is the same as `[0-9][0-9]`
- `\d{3}` is the same as `[0-9]{3}`
- `\d{2,4}` is the same as `[0-9]{2,4}`
- `\d{5,}` is the same as `[0-9]{5,}`
- `^\d+$` is the same as `^[0-9]+$`

The curly braces ("`{}`") are called quantifiers, and they specify the number (or range) of characters in the expressions that precede them.

MATCHING CHARACTER CLASSES WITH THE `re` MODULE

The `re` module provides the following methods for matching and searching one or more occurrences of a regular expression in a text string:

- `match()`: Determine if the RE matches at the *beginning* of the string
- `search()`: Scan through a string, looking for *any* location where the RE matches

- `findall()`: Find *all* substrings where the RE matches and return them as a list
- `finditer()`: Find all substrings where the RE matches and return them as an iterator

NOTE *The* `match()` *function only matches patterns from the start of a string.*

The next section shows you how to use the `match()` function in the `re` module.

USING THE `RE.MATCH()` METHOD

The `re.match()` method attempts to match RE patterns in a text string (with optional flags), and it has the following syntax:

```
re.match(pattern, string, flags=0)
```

The `pattern` parameter is the regular expression that you want to match in the `string` parameter. The flags parameter allows you to specify multiple flags using the bitwise OR operator that is represented by the pipe "|" symbol.

The `re.match` method returns a match object on success and `None` on failure. Use the `group(num)` or `groups()` function of the match object to get a matched expression.

- `group(num=0)`: This method returns the entire match (or specific subgroup `num`)
- `groups()`: This method returns all matching subgroups in a tuple (empty if there weren't any)

NOTE *The* `re.match()` *method only matches patterns from the start of a text string, which is different from the* `re.search()` *method discussed later in this appendix.*

The following code block illustrates how to use the `group()` function in regular expressions:

```
>>> import re
>>> p = re.compile('(a(b)c)de')
>>> m = p.match('abcde')
>>> m.group(0)
'abcde'
>>> m.group(1)
'abc'
>>> m.group(2)
'b'
```

Notice that the higher numbers inside the `group()` method match more deeply nested expressions that are specified in the initial regular expression.

Listing A.2 displays the contents of `MatchGroup1.py` that illustrates how to use the `group()` function to match an alphanumeric text string and an alphabetic string.

LISTING A.2: MatchGroup1.py

```
import re

line1 = 'abcd123'
line2 = 'abcdefg'
mixed = re.compile(r"^[a-z0-9]{5,7}$")
line3 = mixed.match(line1)
line4 = mixed.match(line2)

print('line1:',line1)
print('line2:',line2)
print('line3:',line3)
print('line4:',line4)
print('line5:',line4.group(0))

line6 = 'a1b2c3d4e5f6g7'
mixed2 = re.compile(r"^([a-z]+[0-9]+){5,7}$")
line7 = mixed2.match(line6)

print('line6:',line6)
print('line7:',line7.group(0))
print('line8:',line7.group(1))

line9 = 'abc123fgh4567'
mixed3 = re.compile(r"^([a-z]*[0-9]*){5,7}$")
line10 = mixed3.match(line9)
print('line9:',line9)
print('line10:',line10.group(0))
```

The output from Listing A.2 is here:

```
line1: abcd123
line2: abcdefg
line3: <_sre.SRE_Match object at 0x100485440>
line4: <_sre.SRE_Match object at 0x1004854a8>
line5: abcdefg
line6: a1b2c3d4e5f6g7
line7: a1b2c3d4e5f6g7
line8: g7
line9: abc123fgh4567
line10: abc123fgh4567
```

Notice that `line3` and `line7` involve two similar but different regular expressions. The variable `mixed` specifies a sequence of lowercase letters followed by digits, where the length of the text string is also between 5 and 7. The string `'abcd123'` satisfies all of these conditions.

On the other hand, `mixed2` specifies a pattern consisting of one or more pairs, where each pair contains one or more lowercase letters followed by one

or more digits, where the length of the matching pairs is also between 5 and 7. In this case, the string `'abcd123'` as well as the string `'a1b2c3d4e5f6g7'` both satisfy these criteria.

The third regular expression `mixed3` specifies a pair such that each pair consists of zero or more occurrences of lowercase letters and zero or more occurrences of a digit, and also that the number of such pairs is between 5 and 7. As you can see from the output, the regular expression in `mixed3` matches lowercase letters and digits in any order.

In the preceding example, the regular expression specified a range for the length of the string, which involves a lower limit of 5 and an upper limit of 7. However, you can also specify a lower limit without an upper limit (or an upper limit without a lower limit).

Listing A.3 displays the contents of `MatchGroup2.py` that illustrates how to use a regular expression and the `group()` function to match an alphanumeric text string and an alphabetic string.

LISTING A.3: MatchGroup2.py

```
import re

alphas = re.compile(r"^[abcde]{5,}")
line1 = alphas.match("abcde").group(0)
line2 = alphas.match("edcba").group(0)
line3 = alphas.match("acbedf").group(0)
line4 = alphas.match("abcdefghi").group(0)
line5 = alphas.match("abcdefghi abcdef")

print('line1:',line1)
print('line2:',line2)
print('line3:',line3)
print('line4:',line4)
print('line5:',line5)
```

Listing A.3 initializes the variable `alphas` as a regular expression that matches any string that starts with one of the letters a through e, and consists of at least five characters. The next portion of Listing A.3 initializes the four variables `line1`, `line2`, `line3`, and `line4` by means of the `alphas` RE that is applied to various text strings. These four variables are set to the first matching group by means of the expression `group(0)`.

The output from Listing A.3 is here:

```
line1: abcde
line2: edcba
line3: acbed
line4: abcde
line5: <_sre.SRE_Match object at 0x1004854a8>
```

Listing A.4 displays the contents of `MatchGroup3.py` that illustrates how to use a regular expression with the `group()` function to match words in a text string.

LISTING A.4: MatchGroup3.py

```
import re

line = "Giraffes are taller than elephants";

matchObj = re.match( r'(.*) are(\.*)', line, re.M|re.I)

if matchObj:
  print("matchObj.group() : ", matchObj.group())
  print("matchObj.group(1) : ", matchObj.group(1))
  print("matchObj.group(2) : ", matchObj.group(2))
else:
  print("matchObj does not match line:", line)
```

The code in Listing A.4 produces the following output:

```
matchObj.group() : Giraffes are
matchObj.group(1) : Giraffes
matchObj.group(2) :
```

Listing A.4 contains a pair of delimiters separated by a pipe ("|") symbol. The first delimiter is `re.M` for "multiline" (this example contains only a single line of text), and the second delimiter `re.I` means "ignore case" during the pattern matching operation. The `re.match()` method supports additional delimiters, as discussed in the next section.

OPTIONS FOR THE `RE.MATCH()` METHOD

The `match()` method supports various optional modifiers that affect the type of matching that will be performed. As you saw in the previous example, you can also specify multiple modifiers separated by the OR ("|") symbol. Additional modifiers that are available for RE are shown here:

- `re.I` performs case-insensitive matches (see previous section)
- `re.L` interprets words according to the current locale
- `re.M` makes $ match the end of a line and makes ^ match the start of any line
- `re.S` makes a period (".") match any character (including a newline)
- `re.U` interprets letters according to the Unicode character set

Experiment with these modifiers by writing Python code that uses them in conjunction with different text strings.

MATCHING CHARACTER CLASSES WITH THE `RE.SEARCH()` METHOD

As you saw earlier in this appendix, the `re.match()` method only matches from the beginning of a string, whereas the `re.search()` method can successfully match a substring anywhere in a text string.

The `re.search()` method takes two arguments, a regular expression pattern and a string, and then searches for the specified pattern in the given string. The `search()` method returns a match object (if the search was successful) or `None`.

As a simple example, the following searches for the pattern `tasty` followed by a five-letter word:

```
import re

str = 'I want a tasty pizza'
match = re.search(r'tasty \w\w\w\w\w', str)

if match:
  ## 'found tasty pizza'
  print('found', match.group())
else:
  print('Nothing tasty here')
```

The output of the preceding code block is here:

```
found tasty pizza
```

The following code block further illustrates the difference between the `match()` method and the `search()` method:

```
>>> import re
>>> print(re.search('this', 'this is the one').span())
(0, 4)
>>>
>>> print(re.search('the', 'this is the one').span())
(8, 11)
>>> print(re.match('this', 'this is the one').span())
(0, 4)
>>> print(re.match('the', 'this is the one').span())
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'NoneType' object has no attribute 'span'
```

MATCHING CHARACTER CLASSES WITH THE `FINDALL()` METHOD

Listing A.5 displays the contents of the Python script `RegEx1.py` that illustrates how to define simple character classes that match various text strings.

LISTING A.5: RegEx1.py

```
import re

str1 = "123456"
matches1 = re.findall("(\d+)", str1)
print('matches1:',matches1
```

```
str1 = "123456"
matches1 = re.findall("(\d\d\d)", str1)
print('matches1:',matches1

str1 = "123456"
matches1 = re.findall("(\d\d)", str1)
print('matches1:',matches1

print
str2 = "1a2b3c456"
matches2 = re.findall("(\d)", str2)
print('matches2:',matches2

print
str2 = "1a2b3c456"
matches2 = re.findall("\d", str2)
print('matches2:',matches2

print
str3 = "1a2b3c456"
matches3 = re.findall("(\w)", str3)
print('matches3:',matches3
```

Listing A.5 contains simple regular expressions (which you have seen already) for matching digits in the variables `str1` and `str2`. The final code block of Listing A.5 matches every character in the string `str3`, effectively "splitting" `str3` into a list where each element consists of one character. The output from Listing A.5 is here (notice the blank lines after the first three output lines):

```
matches1: ['123456']
matches1: ['123', '456']
matches1: ['12', '34', '56']

matches2: ['1', '2', '3', '4', '5', '6']

matches2: ['1', '2', '3', '4', '5', '6']

matches3: ['1', 'a', '2', 'b', '3', 'c', '4', '5', '6']
```

Finding Capitalized Words in a String

Listing A.6 displays the contents of the Python script `FindCapitalized.py` that illustrates how to define simple character classes that match various text strings.

LISTING A.6: FindCapitalized.py

```
import re

str = "This Sentence contains Capitalized words"
caps = re.findall(r'[A-Z][\w\.-]+', str)
```

```
print('str: ',str
print('caps:',caps
```

Listing A.6 initializes the string variable `str` and the RE `caps` that matches any word that starts with a capital letter because the first portion of `caps` is the pattern `[A-Z]` that matches any capital letter between `A` and `Z` inclusive.

The output of Listing A.6 is here:

```
str: This Sentence contains Capitalized words
caps: ['This', 'Sentence', 'Capitalized']
```

ADDITIONAL MATCHING FUNCTION FOR REGULAR EXPRESSIONS

After invoking any of the methods `match()`, `search()`, `findAll()`, or `finditer()`, you can invoke additional methods on the "matching object". An example of this functionality using the `match()` method is here:

```
import re

p1 = re.compile('[a-z]+')
m1 = p1.match("hello")
```

In the preceding code block, the `p1` object represents the compiled regular expression for one or more lowercase letters, and the "matching object" `m1` object supports the following methods:

- `group()` return the string matched by the RE
- `start()` return the starting position of the match
- `end()` return the ending position of the match
- `span()` return a tuple containing the (start, end) positions of the match

As a further illustration, Listing A.7 displays the contents of `SearchFunction1.py` that illustrates how to use the `search()` method and the `group()` method.

LISTING A.7: SearchFunction1.py

```
import re

line = "Giraffes are taller than elephants";

searchObj = re.search( r'(.*) are(\.*)', line, re.M|re.I)

if searchObj:
  print("searchObj.group() : ", searchObj.group()
  print("searchObj.group(1) : ", searchObj.group(1)
  print("searchObj.group(2) : ", searchObj.group(2)
else:
  print("searchObj does not match line:", line
```

Listing A.7 contains the variable line that represents a text string, and the variable `searchObj` is an RE involving the `search()` method and pair of pipe-delimited modifiers (discussed in more detail later in the appendix). If `searchObj` is not null, the if/else conditional code in Listing A.7 displays the contents of the three groups resulting from the successful match with the contents of the variable line. The output from Listing A.7 is here:

```
searchObj.group() : Giraffes are
searchObj.group(1) : Giraffes
searchObj.group(2) :
```

GROUPING WITH CHARACTER CLASSES IN REGULAR EXPRESSIONS

In addition to the character classes that you have seen earlier in this appendix, you can specify subexpressions of character classes.

Listing A.8 displays the contents of `Grouping1.py` that illustrates how to use the `search()` method.

LISTING A.8: Grouping1.py

```
import re

p1 = re.compile('(ab)*')
print('match1:',p1.match('abababababab').group()
print('span1: ',p1.match('abababababab').span()

p2 = re.compile('(a)b')
m2 = p2.match('ab')
print('match2:',m2.group(0)
print('match3:',m2.group(1)
```

Listing A.8 starts by defining the RE `p1` that matches zero or more occurrences of the string `ab`. The first `print()` statement displays the result of using the `match()` function of `p1` (followed by the `group()` function) against a string, and the result is a string. This illustrates the use of "method chaining", which eliminates the need for an intermediate object (as shown in the second code block). The second `print()` statement displays the result of using the `match()` function of `p1`, followed by applying the `span()` function, against a string. In this case the result is a numeric range (see the output as follows).

The second part of Listing A.8 defines the RE `p2` that matches an optional letter `a` followed by the letter `b`. The variable `m2` invokes the match method on `p2` using the string `ab`. The third `print()` statement displays the result of invoking `group(0)` on `m2`, and the fourth `print()` statement displays the result of involving `group(1)` on `m2`. Both results are substrings of the input string `ab`. Recall that `group(0)` returns the highest level match that occurred, and `group(1)` returns a more "specific" match that occurred, such as one that

involves the parentheses in the definition of `p2`. The higher the value of the integer in the expression `group(n)`, the more specific the match.

The output from Listing A.8 is here:

```
match1: abababababab
span1: (0, 10)
match2: ab
match3: a
```

USING CHARACTER CLASSES IN REGULAR EXPRESSIONS

This section contains some examples that illustrate how to use character classes to match various strings and also how to use delimiters in order to split a text string. For example, one common date string involves a date format of the form MM/DD/YY. Another common scenario involves records with a delimiter that separates multiple fields. Usually such records contain one delimiter, but as you will see, Python makes it very easy to split records using multiple delimiters.

Matching Strings with Multiple Consecutive Digits

Listing A.9 displays the contents of the Python script `MatchPatterns1.py` that illustrates how to define simple regular expressions in order to split the contents of a text string based on the occurrence of one or more consecutive digits.

Although the regular expressions `\d+/\d+/\d+` and `\d\d/\d\d/\d\d\d\d` both match the string `08/13/2014`, the first regular expression matches more patterns than the second regular expression, which is an "exact match" with respect to the number of matching digits that are allowed.

LISTING A.9: MatchPatterns1.py

```
import re

date1 = '02/28/2013'
date2 = 'February 28, 2013'

# Simple matching: \d+ means match one or more digits
if re.match(r'\d+/\d+/\d+', date1):
  print('date1 matches this pattern')
else:
  print('date1 does not match this pattern')

if re.match(r'\d+/\d+/\d+', date2):
  print('date2 matches this pattern')
else:
  print('date2 does not match this pattern')
```

The output from launching Listing A.9 is here:

```
date1 matches this pattern
date2 does not match this pattern
```

Reversing Words in Strings

Listing A.10 displays the contents of the Python script `ReverseWords1.py` that illustrates how to reverse a pair of words in a string.

LISTING A.10: ReverseWords1.py

```
import re

str1 = 'one two'
match = re.search('([\w.-]+) ([\w.-]+)', str1)

str2 = match.group(2) + ' ' + match.group(1)
print('str1:',str1
print('str2:',str2
```

The output from Listing A.10 is here:

```
str1: one two
str2: two one
```

Now that you understand how to define regular expressions for digits and letters, let's look at some more sophisticated regular expressions.

For example, the following expression matches a string that is any combination of digits, uppercase letters, or lowercase letters (i.e., no special characters):

```
^[a-zA-Z0-9]$
```

Here is the same expression rewritten using character classes:

```
^[\w\W\d]$
```

MODIFYING TEXT STRINGS WITH THE RE MODULE

The Python `re` module contains several methods for modifying strings. The `split()` method uses a regular expression to "split" a string into a list. The `sub()` method finds all substrings where the regular expression matches, and then replaces them with a different string. The `subn()` method performs the same functionality as `sub()`, and also returns the new string and the number of replacements. The following subsections contain examples that illustrate how to use the functions `split()`, `sub()`, and `subn()` in regular expressions.

SPLITTING TEXT STRINGS WITH THE RE.SPLIT() METHOD

Listing A.11 displays the contents of the Python script RegEx2.py that illustrates how to define simple regular expressions in order to split the contents of a text string.

LISTING A.11: RegEx2.py

```
import re

line1 = "abc def"
result1 = re.split(r'[\s]', line1)
print('result1:',result1

line2 = "abc1,abc2:abc3;abc4"
result2 = re.split(r'[,:;]', line2)
print('result2:',result2

line3 = "abc1,abc2:abc3;abc4 123 456"
result3 = re.split(r'[,:;\s]', line3)
print('result3:',result3
```

Listing A.11 contains three blocks of code, each of which uses the split() method in the re module in order to tokenize three different strings. The first regular expression specifies a whitespace, the second regular expression specifies three punctuation characters, and the third regular expression specifies the combination of the first two regular expressions.

The output from launching RegEx2.py is here:

```
result1: ['abc', 'def']
result2: ['abc1', 'abc2', 'abc3', 'abc4']
result3: ['abc1', 'abc2', 'abc3', 'abc4', '123', '456']
```

SPLITTING TEXT STRINGS USING DIGITS AND DELIMITERS

Listing A.12 displays the contents of SplitCharClass1.py that illustrates how to use a regular expression consisting of a character class, the "." character, and a whitespace to split the contents of two text strings.

LISTING A.12: SplitCharClass1.py

```
import re

line1 = '1. Section one 2. Section two 3. Section three'
line2 = '11. Section eleven 12. Section twelve 13. Section
thirteen'

print(re.split(r'\d+\. ', line1)
print(re.split(r'\d+\. ', line2)
```

Listing A.12 contains two text strings that can be split using the same regular expression '\d+\. '. Note that if you use the expression '\d\. ', only the first text string will split correctly. The result of launching Listing A.12 is here:

```
['', 'Section one ', 'Section two ', 'Section three']
['', 'Section eleven ', 'Section twelve ', 'Section
thirteen']
```

SUBSTITUTING TEXT STRINGS WITH THE RE.SUB() METHOD

Earlier in this appendix you saw a preview of using the sub() method to remove all the metacharacters in a text string. The following code block illustrates how to use the re.sub() method to substitute alphabetic characters in a text string.

```
>>> import re
>>> p = re.compile( '(one|two|three)')
>>> p.sub( 'some', 'one book two books three books')
'some book some books some books'
>>>
>>> p.sub( 'some', 'one book two books three books',
count=1)
'some book two books three books'
```

The following code block uses the re.sub() method in order to insert a line feed after each alphabetic character in a text string:

```
>>> line = 'abcde'
>>> line2 = re.sub('', '\n', line)
>>> print('line2:',line2
line2:
a
b
c
d
e
```

MATCHING THE BEGINNING AND THE END OF TEXT STRINGS

Listing A.13 displays the contents of the Python script RegEx3.py that illustrates how to find substrings using the startswith() function and endswith() function.

LISTING A.13: RegEx3.py

```
import re

line2 = "abc1,Abc2:def3;Def4"
result2 = re.split(r'[,:;]', line2)
```

```
for w in result2:
  if(w.startswith('Abc')):
    print('Word starts with Abc:',w
  elif(w.endswith('4')):
    print('Word ends with 4:',w
  else:
    print('Word:',w
```

Listing A.13 starts by initializing the string `line2` (with punctuation characters as word delimiters) and the RE `result2` that uses the `split()` function with a comma, colon, and semicolon as "split delimiters" in order to tokenize the string variable `line2`.

The output after launching Listing A.13 is here:

```
Word: abc1
Word starts with Abc: Abc2
Word: def3
Word ends with 4: Def4
```

Listing A.14 displays the contents of the Python script `MatchLines1.py` that illustrates how to find substrings using character classes.

LISTING A.14: MatchLines1.py

```
import re

line1 = "abcdef"
line2 = "123,abc1,abc2,abc3"
line3 = "abc1,abc2,123,456f"

if re.match("^[A-Za-z]*$", line1):
  print('line1 contains only letters:',line1

# better than the preceding snippet:
line1[:-1].isalpha()
  print('line1 contains only letters:',line1

if re.match("^[\w]*$", line1):
  print('line1 contains only letters:',line1

if re.match(r"^[^\W\d_]+$", line1, re.LOCALE):
  print('line1 contains only letters:',line1
print

if re.match("^[0-9][0-9][0-9]", line2):
  print('line2 starts with 3 digits:',line2

if re.match("^\d\d\d", line2):
  print('line2 starts with 3 digits:',line2
print

# does not work: fixme
if re.match("[0-9][0-9][0-9][a-z]$", line3):
  print('line3 ends with 3 digits and 1 char:',line3
```

```
# does not work: fixme
if re.match("[a-z]$", line3):
  print('line3 ends with 1 char:',line3
```

Listing A.14 starts by initializing three string variables `line1`, `line2`, and `line3`. The first RE contains an expression that matches any line containing uppercase or lowercase letters (or both):

```
if re.match("^[A-Za-z]*$", line1):
```

The following two snippets also test for the same thing:

```
line1[:-1].isalpha()
```

The preceding snippet starts from the rightmost position of the string and checks if each character is alphabetic.

The next snippet checks if `line1` can be tokenized into words (a word contains only alphabetic characters):

```
if re.match("^[\w]*$", line1):
```

The next portion of Listing A.14 checks if a string contains three consecutive digits:

```
if re.match("^[0-9][0-9][0-9]", line2):
  print('line2 starts with 3 digits:',line2

if re.match("^\d\d\d", line2):
```

The first snippet uses the pattern `[0-9]` to match a digit, whereas the second snippet uses the expression `\d` to match a digit.

The output from Listing A.14 is here:

```
line1 contains only letters: abcdef
line1 contains only letters: abcdef
line1 contains only letters: abcdef
line1 contains only letters: abcdef

line2 starts with 3 digits: 123,abc1,abc2,abc3
line2 starts with 3 digits: 123,abc1,abc2,abc3
```

COMPILATION FLAGS

Compilation flags modify the manner in which regular expressions work. Flags are available in the re module as a long name (such as IGNORECASE) and a short, one-letter form (such as I). The short form is the same as the flags in pattern modifiers in Perl. You can specify multiple flags by using the "|" symbol. For example, `re.I | re.M` sets both the `I` and `M` flags.

You can check the online Python documentation regarding all the available compilation flags in Python.

COMPOUND REGULAR EXPRESSIONS

Listing A.15 displays the contents of `MatchMixedCase1.py` that illustrates how to use the pipe ("|") symbol to specify two regular expressions in the same `match()` function.

LISTING A.15: MatchMixedCase1.py

```
import re

line1 = "This is a line"
line2 = "That is a line"

if re.match("^[Tt]his", line1):
  print('line1 starts with This or this:'
  print(line1
else:
  print('no match'

if re.match("^This|That", line2):
  print('line2 starts with This or That:'
  print(line2
else:
  print('no match'
```

Listing A.15 starts with two string variables `line1` and `line2`, followed by an if/else conditional code block that checks if `line1` starts with the RE `[Tt]his`, which matches the string `This` as well as the string `this`.

The second conditional code block checks if `line2` starts with the string `This` or the string `That`. Notice the "^" metacharacter, which in this context anchors the RE to the beginning of the string. The output from Listing A.15 is here:

```
line1 starts with This or this:
This is a line
line2 starts with This or That:
That is a line
```

COUNTING CHARACTER TYPES IN A STRING

You can use a regular expression to check whether a character is a digit, a letter, or some other type of character. Listing A.16 displays the contents of `CountDigitsAndChars.py` that performs this task.

LISTING A.16: CountDigitsAndChars.py

```
import re

charCount = 0
digitCount = 0
otherCount = 0
```

```
line1 = "A line with numbers: 12 345"

for ch in line1:
   if(re.match(r'\d', ch)):
     digitCount = digitCount + 1
   elif(re.match(r'\w', ch)):
     charCount = charCount + 1
   else:
     otherCount = otherCount + 1

print('charcount:',charCount
print('digitcount:',digitCount
print('othercount:',otherCount
```

Listing A.16 initializes three numeric counter-related variables, followed by the string variable `line1`. The next part of Listing A.16 contains a `for` loop that processes each character in the string `line1`. The body of the `for` loop contains a conditional code block that checks whether the current character is a digit, a letter, or some other non-alphanumeric character. Each time there is a successful match, the corresponding "counter" variable is incremented.

The output from Listing A.16 is here:

```
charcount: 16
digitcount: 5
othercount: 6
```

REGULAR EXPRESSIONS AND GROUPING

You can also "group" subexpressions and even refer to them symbolically. For example, the following expression matches zero or 1 occurrences of 3 consecutive letters or digits:

```
^([a-zA-Z0-9]{3,3})?
```

The following expression matches a telephone number (such as 650-555-1212) in the United States:

```
^\d{3,3}[-]\d{3,3}[-]\d{4,4}
```

The following expression matches a zip code (such as `67827` or `94343-04005`) in the United States:

```
^\d{5,5}([-]\d{5,5})?
```

The following code block partially matches an email address:

```
str = 'john.doe@google.com'
  match = re.search(r'\w+@\w+', str)
  if match:
    print(match.group() ## 'doe@google'
```

Exercise: use the preceding code block as a starting point in order to define a regular expression for email addresses.

SIMPLE STRING MATCHES

Listing A.17 displays the contents of the Python script `RegEx4.py` that illustrates how to define regular expressions that match various text strings.

LISTING A.17: RegEx4.py

```
import re

searchString = "Testing pattern matches"

expr1 = re.compile( r"Test" )
expr2 = re.compile( r"^Test" )
expr3 = re.compile( r"Test$" )
expr4 = re.compile( r"\b\w*es\b" )
expr5 = re.compile( r"t[aeiou]", re.I )

if expr1.search( searchString ):
   print('"Test" was found.'

if expr2.match( searchString ):
   print('"Test" was found at the beginning of the line.'

if expr3.match( searchString ):
   print('"Test" was found at the end of the line.'

result = expr4.findall( searchString )

if result:
   print('There are %d words(s) ending in "es":' % \
       ( len( result ) ),

   for item in result:
       print(" " + item,

print

result = expr5.findall( searchString )
if result:
   print('The letter t, followed by a vowel, occurs %d
times:' % \
       ( len( result ) ),

   for item in result:
       print(" "+item,

print
```

Listing A.17 starts with the variable `searchString` that specifies a text string, followed by the REs `expr1`, `expr2`, `expr3`. The RE `expr1` matches the

string `Test` that occurs anywhere in `searchString`, whereas `expr2` matches `Test` if it occurs at the beginning of `searchString`, and `expr3` matches `Test` if it occurs at the end of `searchString`. The RE `expr` matches words that end in the letters `es`, and the RE `expr5` matches the letter `t` followed by a vowel.

The output from Listing A.17 is here:

```
"Test" was found.
"Test" was found at the beginning of the line.
There are 1 words(s) ending in "es": matches
The letter t, followed by a vowel, occurs 3 times: Te ti te
```

ADDITIONAL TOPICS FOR REGULAR EXPRESSIONS

In addition to the Python-based search/replace functionality that you have seen in this appendix, you can also perform a greedy search and substitution. Perform an Internet search to learn what these features are and how to use them in Python code.

SUMMARY

This appendix showed you how to create various types of regular expressions. First you learned how to define primitive regular expressions using sequences of digits, lowercase letters, and uppercase letters. Next you learned how to use character classes, which are more convenient and simpler expressions that can perform the same functionality. You also learned how to use the Python `re` library in order to compile regular expressions and then use them to see if they match substrings of text strings.

EXERCISES

- Exercise 1: Given a text string, find the list of words (if any) that start or end with a vowel, and treat upper- and lowercase vowels as distinct letters. Display this list of words in alphabetical order, and also in descending order based on their frequency.
- Exercise 2: Given a text string, find the list of words (if any) that contain lowercase vowels or digits or both, but no uppercase letters. Display this list of words in alphabetical order, and also in descending order based on their frequency.
- Exercise 3: There is a spelling rule in English specifying that "the letter i is before e, except after c", which means that "receive" is correct but "recieve" is incorrect. Write a Python script that checks for incorrectly spelled words in a text string.

- Exercise 4: Subject pronouns cannot follow a preposition in the English language. Thus, "between you and me" and "for you and me" are correct, whereas "between you and I" and "for you and I" are incorrect. Write a Python script that checks for incorrect grammar in a text string, and search for the prepositions "between", "for", and "with". In addition, search for the subject pronouns "I", "you", "he", and "she". Modify and display the text with the correct grammar usage.
- Exercise 5: Find the words in a text string whose length is at most 4 and then print all the substrings of those characters. For example, if a text string contains the word "text", then print the strings "t", "te", "tex", and "text".

INDEX