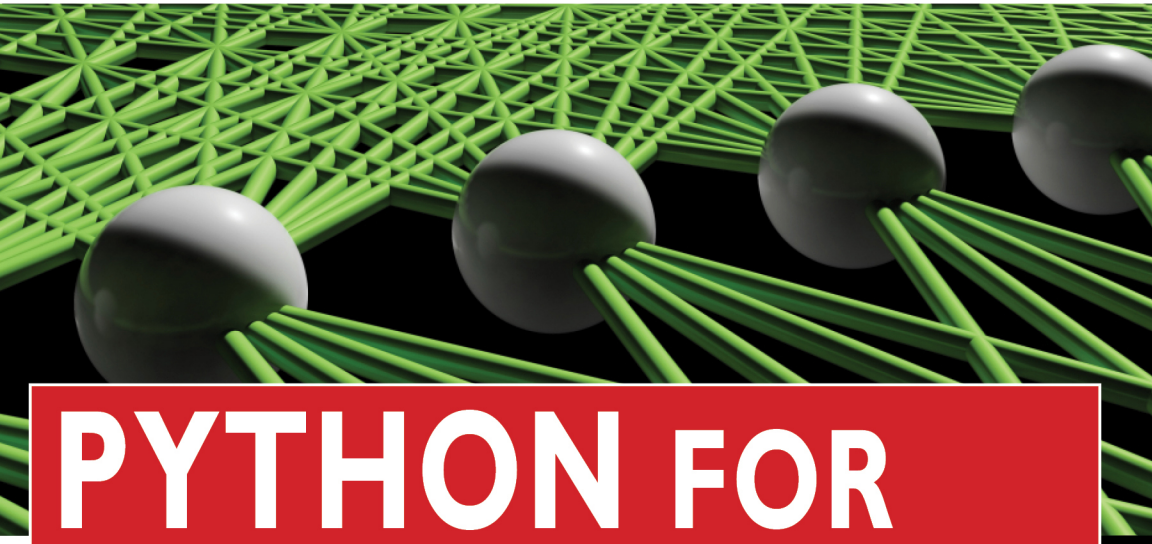

PYTHON FOR TENSOR FLOW

POCKET PRIMER

DISC Included!

OSWALD CAMPESATO

PYTHON FOR TENSORFLOW

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 insure 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 FOR TENSORFLOW

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 for TensorFlow Pocket Primer.*
ISBN: 978-1-68392-361-9

Library of Congress Control Number: 2019939380

192021321 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 GOAL?

The goal of this book is to show advanced beginners code samples in various Python libraries that are useful for TensorFlow programs written in Python. The first four chapters contain Python-related material, and the fifth chapter gives you an introduction to TensorFlow. However, the fifth chapter does not contain any code samples that show you how to solve machine learning problems (such as linear regression, logistic regression, and so forth) in TensorFlow.

The material in this book is primarily for those who have some programming experience, and not really suitable for "absolute beginners." It is suitable as a fast-paced introduction to various "core" features. The purpose of the material in the chapters is to illustrate how to solve a variety of tasks, after which you can do further reading to deepen your knowledge.

IS THIS BOOK IS FOR ME AND WHAT WILL I LEARN?

This book is intended for software developers who are advanced beginners (perhaps also some intermediate-level developers). You need some familiarity with working from the command line in a Unix-like environment. However, there are subjective prerequisites, such as a strong desire to learn how to write TensorFlow programs, along with the motivation and discipline to read and understand the code samples.

If you are adequately prepared and motivated, then in Chapter 5 you will learn how to write basic TensorFlow programs.

This book saves you the time required to search for relevant code samples, adapting them to your specific needs, which is a potentially time-consuming process. In any case, if you're not sure whether or not you can absorb the

material in this book, glance through the code samples to get a feel for the level of complexity.

HOW WERE THE CODE SAMPLES CREATED?

The code samples in this book were created and tested using bash on a Macbook Pro with OS X 10.12.6 (MacOS Sierra). Regarding their content: the code samples are derived primarily from the author, and in some cases there are code samples that incorporate short sections of code from discussions in online forums. The key point to remember is that the overwhelming majority of the code samples follow the "Four Cs": they must be Clear, Concise, Complete, and Correct to the extent that it's possible to do so, given the size of this book.

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

WHICH TOPICS ARE EXCLUDED?

This book does not cover Python-related topics that are not helpful for learning the fundamentals of TensorFlow 1.x (and there are many such topics). However, there is a follow-up book that does contain code samples for machine learning in TensorFlow 1.x and Python, and the title of that book is the *TensorFlow Pocket Primer*.

DO I NEED EVERYTHING IN THIS BOOK TO LEARN TENSORFLOW?

No. There are sections in Chapter 4 (such as the material pertaining to Seaborn) that you could skip and proceed to TensorFlow. However, the "extra" topics are included because you will probably encounter them in the future. So, even though this book is primarily intended to prepare you for learning TensorFlow, there is some material that may be relevant if you decide to learn about machine learning with Python instead of TensorFlow.

HOW DO I SET UP A COMMAND SHELL?

If you are a Mac user, there are three ways to do so. The first method 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).

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. The best answer is to try out a new tool or technique from the book on a problem or task you care about, professionally or personally. Precisely what that might be depends on what you do, as the needs of a data scientist, manager, student, or developer are all different. In addition, keep what you learned in mind as you tackle new data cleaning or manipulation challenges. Sometimes knowing that a technique is possible, makes finding a solution easier, even if you have to re-read the section to remember exactly how the syntax works.

If you have reached the limits of what you have learned here and want to get further technical depth regarding TensorFlow 1.x, consider the recommended book that is mentioned earlier in this Preface, as well as an upcoming book about TensorFlow 2.

COMPANION FILES

Code samples and figures appearing in the book may be obtained by writing to the publisher at info@merclearning.com.

CHAPTER 1

INTRODUCTION TO PYTHON

This chapter provides a fast-paced introduction to Python, along with tools for installing Python modules, and examples of how to work with some data types in Python. This chapter does not contain any historical information about Python, nor topics such as creating Python class, inheritance, threads, or any other topics that are beyond introductory material. The reason is simple: the purpose of this chapter is to familiarize you with enough Python so that you can learn the material in subsequent chapters.

The first part of this chapter covers the installation of 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.

If you like to read documentation, one of the best third party `stdlib` documentation publications is `pymotw` ("Python Module of the Week") by Doug Hellman, and its home page is here:

http://pymotw.com/2/

NOTE *The Python scripts in this book are for Python 2.7.5 and although most of them are probably compatible with Python 2.6, these scripts require a modest degree of conversion (e.g., the syntax of the* `print()` *command) in order to be compatible with Python 3.*

TOOLS FOR PYTHON

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

http://continuum.io/downloads

Anaconda is well suited for modules such as `NumPy` and `SciPy` (discussed in Chapter 2 and Chapter 4), 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

Two very nice features of `IPython` are tab expansion and "?" and an example of tab expansion is shown here:

```
python
Python 2.7.5 |Anaconda 1.6.1 (x86_64)| (default, Jun 28
                                                2013, 22:20:13)
Type "copyright", "credits" or "license" for more
                                                  information.

IPython 0.13.2 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for
                                                 extra details.

In [1]: di
%dirs   dict    dir     divmod
```

In the preceding session, if you type the characters `di`, iPython responds with the following line that contains all the functions that start with the letters `di`:

```
%dirs   dict    dir     divmod
```

If you enter a question mark ("?"), `ipython` provides textual assistance, the first part of which is here:

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

IPython offers a combination of convenient shell features,
special commands and a history mechanism for both input
(command history) and output (results caching, similar
to Mathematica). It is intended to be a fully compatible
replacement for the standard Python interpreter, while
offering vastly improved functionality and flexibility.
```

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 2.7.5 :: Anaconda 1.6.1 (x86_64)
```

NOTE *Install Python 2.7.5 (or as close as possible to this version) on your machine so that you will have the same version of Python that was used to test the Python scripts in this book.*

If you need to install Python on your machine, navigate to the Python home page and select the download 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 described in the following instructions:

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 2.7.5 |Anaconda 1.6.1 (x86_64)| (default, Jun 28
                                              2013, 22:20:13)
[GCC 4.0.1 (Apple Inc. build 5493)] 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 are 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 MULTI-LINES

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 "GHJ"
```

Multi-line 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 do not fit on a single line.

You can specify multiple statements on 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 semi-colons and the continuation character is 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 metacharacters, 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
b1: ' b2: "
```

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 the 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 on 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 whitespace, 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 a 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`, `sys`, 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
```

Some 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 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 non-existent 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 is 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(97,123):
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 non-zero) 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 is 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` command:

```
>>> 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 would 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.

Listing 1.3 displays the contents of `CharTypes.py` that illustrates how to determine if a string contains digits or characters.

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:

```
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 NEWLINE CHARACTERS

If you need to suppress white space and a newline 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 second technique for suppressing a new line character involves the `write()` function, as shown in the following code block:

```
import sys
write = sys.stdout.write
write('123')
write('456\n')
```

The output is here:

```
123456
```

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 "Another format: ",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. However, 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 is similar to the exception handling in Java code.

Listing 1.10 displays the contents of `Exception1.py` that illustrates how to handle various types of exceptions.

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 = raw_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 is 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 = raw_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` 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 = raw_input(msg)

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

msg = 'Enter a number:'
val2 = raw_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 above script as follows:

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

This will produce the 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.

EXERCISES

Exercise 1: Write a Python script that is a temperature converter from Celsius to Fahrenheit using this formula: `F = C*9/5 + 32`. In addition, convert from Fahrenheit to Celsius using the formula `C = (F-32)*5/9`. Test your program with several values for Celsius and Fahrenheit.

Exercise 2: Extend the Python script that you wrote in exercise #1 to support user input. Make sure that you include exception-handling code in this new version of the script.

Exercise 3: Write a Python script that accepts a line of input text from users and then prints the first and last character of the input text.

Exercise 4: Write a Python script that accepts a line of input text from users and then prints the first word of the input text.

Exercise 5: Write a Python script that accepts a line of input text from users and then checks if the first five characters form a palindrome (words that are spelled the same when you read them from left-to-right or from right-to-left). The Python `chr()` function will help you complete this exercise.

CHAPTER 2

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 4), 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 images 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 package that provides many convenience methods and also better performance. `NumPy` provides a core library for scientific computing

in Python, with performant multi-dimensional 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 is very common in TensorFlow code as well as Python code.

Useful NumPy Features

The `NumPy` package provides the *ndarray* object that encapsulates *multi-dimensional* 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:

- NumPy arrays have a fixed size, whereas Python lists can expand dynamically. Whenever you modify the size of an *ndarray*, a new array is created and the original array is deleted.
- NumPy arrays are homogeneous, which means that the elements in a NumPy array must have the same data type. Except for NumPy arrays of objects, the elements in NumPy arrays of any other data type must have the same size in memory.
- NumPy arrays support more efficient execution (and require less code) of various types of operations on large numbers of data.
- Many scientific Python-based packages rely on NumPy arrays, and knowledge of NumPy arrays is becoming increasingly important.

Now that you have a general idea about `NumPy`, let us 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 operations 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 2.1 displays the contents of nparray1.py that illustrates some operations on a NumPy array.

LISTING 2.1: 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 2.1 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 2.1 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 are ready to look at some loop operations for lists and arrays.

WORKING WITH LOOPS

Listing 2.2 displays the contents of loop1.py that illustrates how to iterate through the elements of a NumPy array and a Python list.

LISTING 2.2: 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 2.2 initializes the variable `list`, which is a Python list, and also the variable `arr1`, which is a `NumPy` array. The next portion of Listing 2.2 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 2.2 is here:

```
1
2
3
1
2
3
```

APPENDING ELEMENTS TO ARRAYS (1)

Listing 2.3 displays the contents of `append1.py` that illustrates how to append elements to a `NumPy` array and a Python list.

LISTING 2.3: 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 2.3 initializes the variable `list`, which is a Python list, and also the variable `arr1`, which is a `NumPy` array. The output from launching Listing 2.3 is here:

```
1
2
3
4
```

```
5
2
4
6
8
10
```

APPENDING ELEMENTS TO ARRAYS (2)

Listing 2.4 displays the contents of `append2.py` that illustrates how to append elements to a `NumPy` array and a Python list.

LISTING 2.4: 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 2.4 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 2.4 is here:

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

MULTIPLY LISTS AND ARRAYS

Listing 2.5 displays the contents of `multiply1.py` that illustrates how to multiply elements in a Python list and a `NumPy` array.

LISTING 2.5: 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 2.5 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 2.5:

```
('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 2.6 displays the contents of `double-list1.py` that illustrates how to double the elements in a Python list.

LISTING 2.6: 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 2.6 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 2.6 is here:

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

LISTS AND EXPONENTS

Listing 2.7 displays the contents of `exponent-list1.py` that illustrates how to compute exponents of the elements in a Python list.

LISTING 2.7: exponent-list1.py

```
import numpy as np

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

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

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

Listing 2.7 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 2.7 is here:

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

ARRAYS AND EXPONENTS

Listing 2.8 displays the contents of `exponent-array1.py` that illustrates how to compute exponents of the elements in a `NumPy` array.

LISTING 2.8: 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 2.8 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 2.8 is here:

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

MATH OPERATIONS AND ARRAYS

Listing 2.9 displays the contents of `mathops-array1.py` that illustrates how to compute exponents of the elements in a `NumPy` array.

LISTING 2.9: 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 2.9 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 2.9 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 2.10 displays the contents of `npsubarray2.py` that illustrates how to compute exponents of the elements in a `NumPy` array.

LISTING 2.10: 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 2.10 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 2.10 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 2.11 displays the contents of np2darray2.py that illustrates how to compute exponents of the elements in a NumPy array.

LISTING 2.11: 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 2.11 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 2.11 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 values.
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 in Chapter 4). 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 2.12 displays the contents of `array-vector.py` that illustrates how to perform vector operations on the elements in a `NumPy` array.

LISTING 2.12: 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 2.12 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 2.12 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 2.13 displays the contents of `dotproduct1.py` that illustrates how to perform the dot product on the elements in a `NumPy` array.

LISTING 2.13: 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 2.13 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`, their inner product that is calculated in three different ways. The output from launching Listing 2.13 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 2.14 displays the contents of `dotproduct2.py` that illustrates how to calculate the dot product of two `NumPy` arrays.

LISTING 2.14: 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 2.14 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 is calculated in three different ways. The output from launching Listing 2.14 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 "NORM" 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 2.15 displays the contents of array-norm.py that illustrates how to calculate the magnitude ("norm") of a NumPy array of numbers.

LISTING 2.15: 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 2.15 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 2.15 consists of six print statements, each of which displays the computed values. The output from launching Listing 2.15 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 2.16 displays the contents of otherops.py that illustrates how to perform other operations on a NumPy array.

LISTING 2.16: 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 2.16 contains two NumPy arrays called a and b followed by five print statements that display the contents of a and b, their Hadamard product, and also their inner product that is calculated in two different ways. The output from launching Listing 2.16 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 2.17 displays the contents of numpy-reshape.py that illustrates how to use the reshape() method on a NumPy array.

LISTING 2.17: 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 2.17 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 2.17 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 on-line 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 2.18 displays the contents of sample-mean-std.py that illustrates how to calculate statistical values from a NumPy array.

LISTING 2.18: 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 2.18 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 2.18 is here:

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

CALCULATING 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 2.19 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 2.19 displays the contents of `stat-values.py` that illustrates how to display various statistical values from a `NumPy` array of random numbers.

LISTING 2.19: 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 2.19 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 2.19 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
```

Chapter 4 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 is 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 constitute 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 will use the terms "line" and "line segment" interchangeably, and now we will delve into the details of lines in the Euclidean plane. Just in case you are a bit fuzzy on the details, here is the equation of a (non-vertical) 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 non-vertical line intersects the y-axis). In case you are 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 will not be working with vertical lines, so will be sticking with the first formula. Figure 2.1 displays three horizontal lines whose equations (from top to bottom) are `y = 4`, `y = 0`, and `y = -3`, respectively.

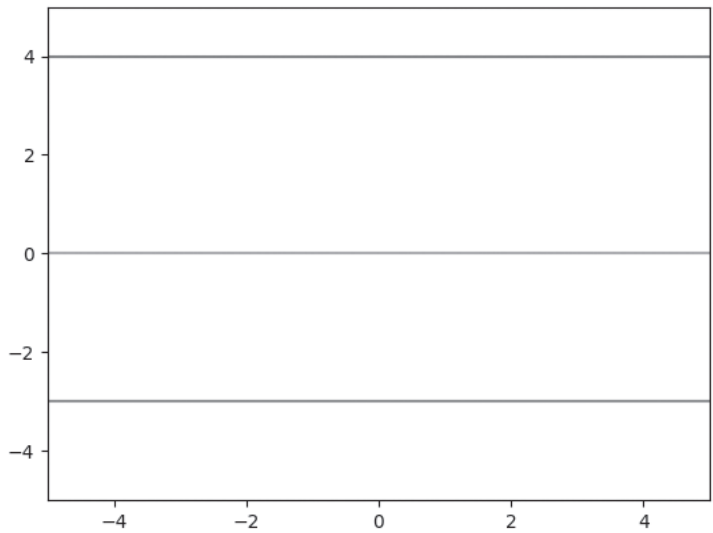

FIGURE 2.1 A Graph of Three Horizontal Line Segments.

Figure 2.2 displays two slanted lines whose equations are y = x and y = –x, respectively.

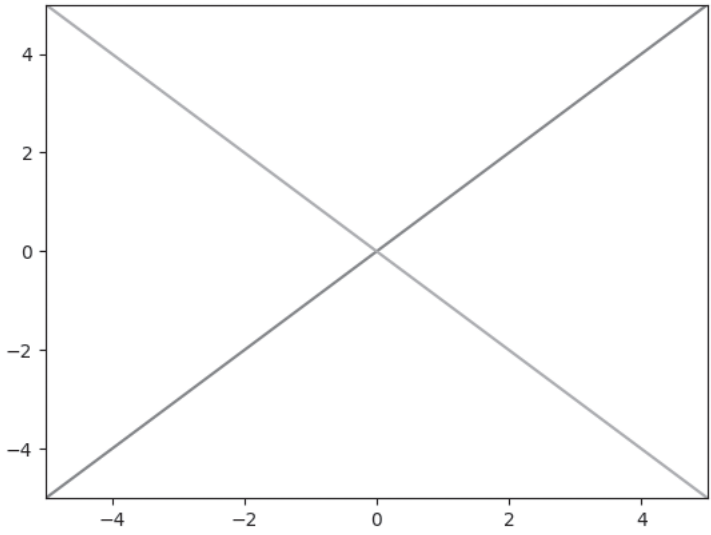

FIGURE 2.2 A Graph of Two Diagonal Line Segments.

Figure 2.3 displays two slanted parallel lines whose equations are `y = 2*x` and `y = 2*x+3`, respectively.

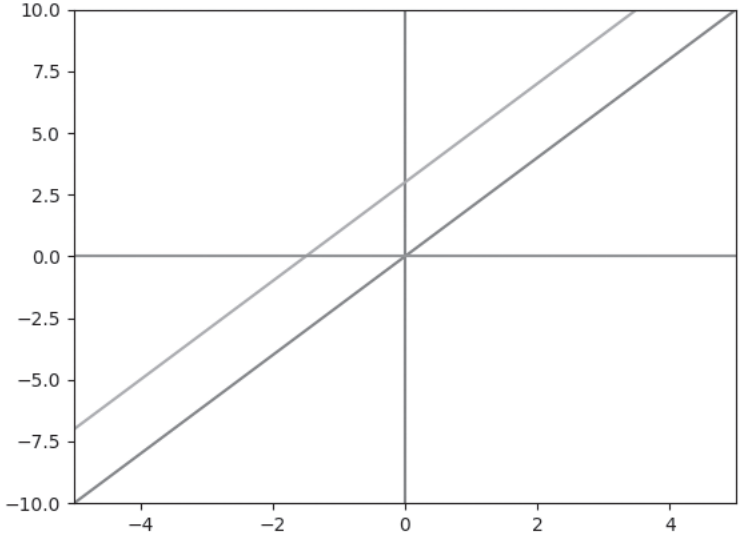

FIGURE 2.3 A Graph of Two Slanted Parallel Line Segments.

Figure 2.4 displays a piecewise linear graph consisting of connected line segments.

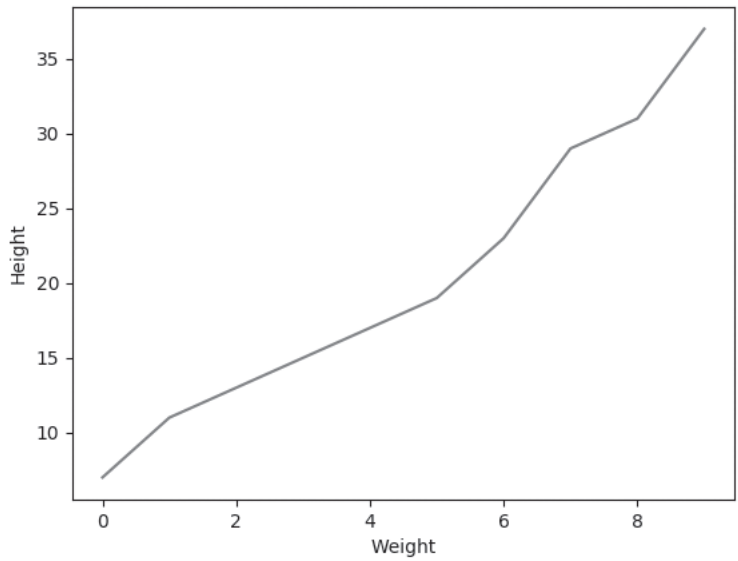

FIGURE 2.4 A Piecewise Linear Graph of Line Segments.

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

PLOTTING A LINE WITH NUMPY AND MATPLOTLIB

Listing 2.20 displays the contents of `np-plot.py` that illustrates how to plot multiple points on a line in the plane.

LISTING 2.20: 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 2.20 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 2.5 displays the output generated by the code in Listing 2.20.

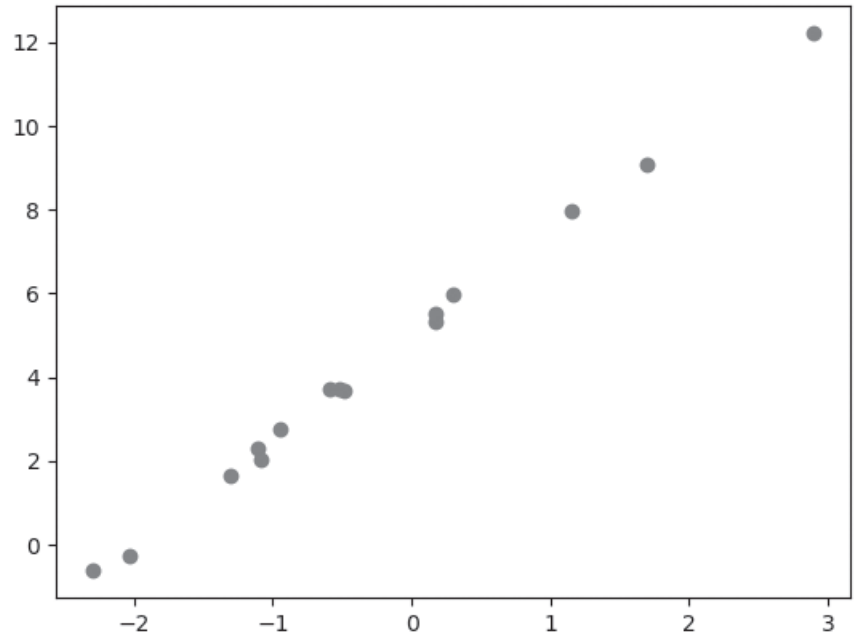

FIGURE 2.5 Datasets with Potential Linear Regression.

PLOTTING A QUADRATIC WITH NUMPY AND MATPLOTLIB

Listing 2.21 displays the contents of `np-plot-quadratic.py` that illustrates how to plot a quadratic function in the plane.

LISTING 2.21: 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**2+ 2*np.random.randn(100,1)

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

Listing 2.21 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 2.6 displays the output generated by the code in Listing 2.21.

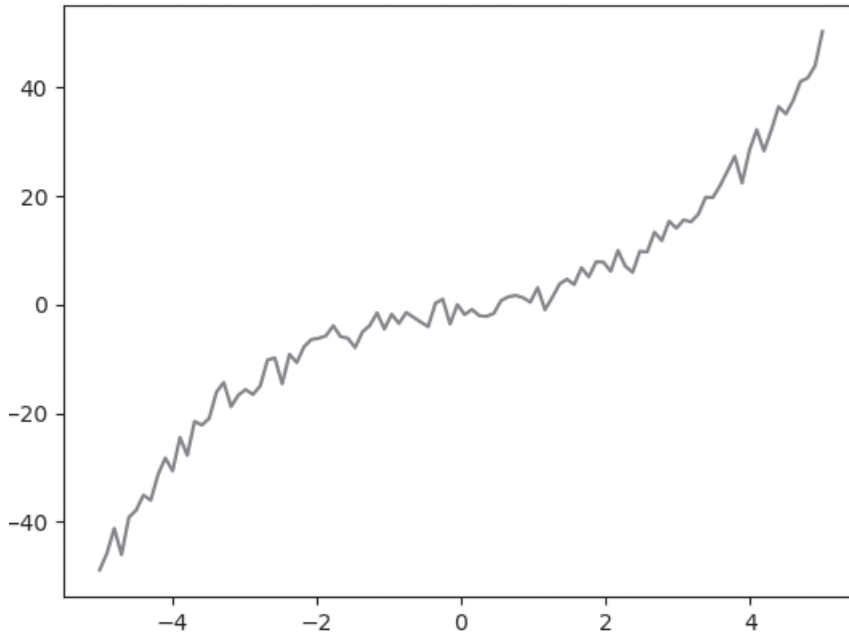

FIGURE 2.6 Datasets with Potential Linear Regression.

Now that you have seen an assortment of line graphs and scatterplots, let us delve into Linear Regression, which is the topic of the next section.

WHAT IS LINEAR REGRESSION?

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 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 a 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 non-vertical points in the plane. For any triple of non-collinear 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 Non-Linear Datasets?

Simple linear regression finds the best fitting line that "represents" a dataset, but what happens if the dataset does not fit a line in the plane? This is an important question. 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 is 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 "represents" 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 2.7.

Figure 2.7 displays a dataset containing four points that do not fit a line.

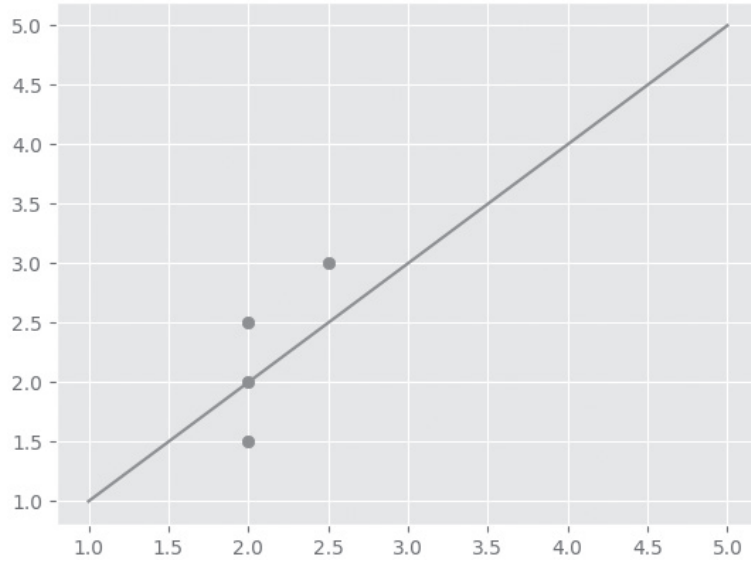

FIGURE 2.7 A Non-Linear Dataset.

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 us defer the non-linear scenario and 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 is called Mean Squared Error (MSE).

THE MSE (MEAN SQUARED ERROR) FORMULA

Figure 2.8 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 (where the latter is the `y` value that each data point would have if that data point were actually on the best fitting line).

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

FIGURE 2.8 The MSE Formula.

Figure 2.8 displays the formula for MSE (Mean Squared Error) for calculating the best fitting line for a set of points in the plane.

Other Error Types

Although we will only discuss MSE for linear regression in this book, there are other types of formulas 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 is easier to compute the derivative of MSE (because it is a differentiable function), it is 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.

Non-Linear 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 mean absolute error.

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

CALCULATING THE MSE MANUALLY

Let us look at two simple graphs, each of which contains a line that approximates a set of points in a scatter plot. 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 2.9 displays a set of points and a line that is a potential candidate for best fitting line for the data.

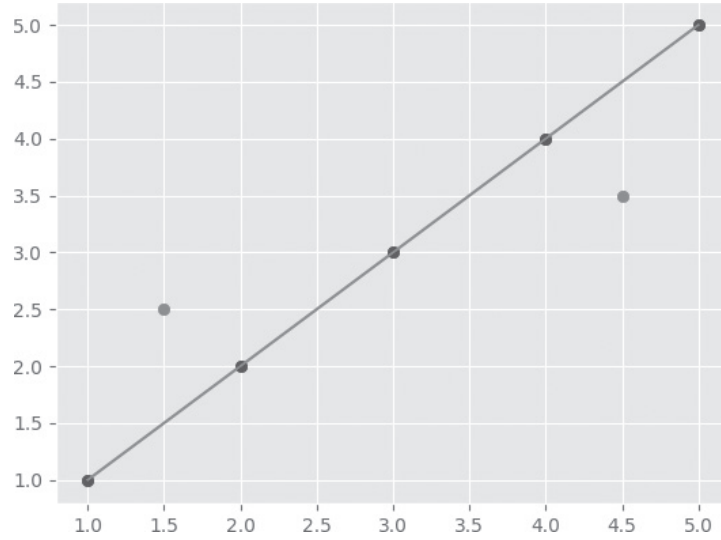

FIGURE 2.9 A Line Graph that Approximates Points of a Scatter Plot.

The MSE for the line in Figure 2.9 is computed as follows:

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

Now look at Figure 2.10 that also displays a set of points and a line that is a potential candidate for best fitting line for the data.

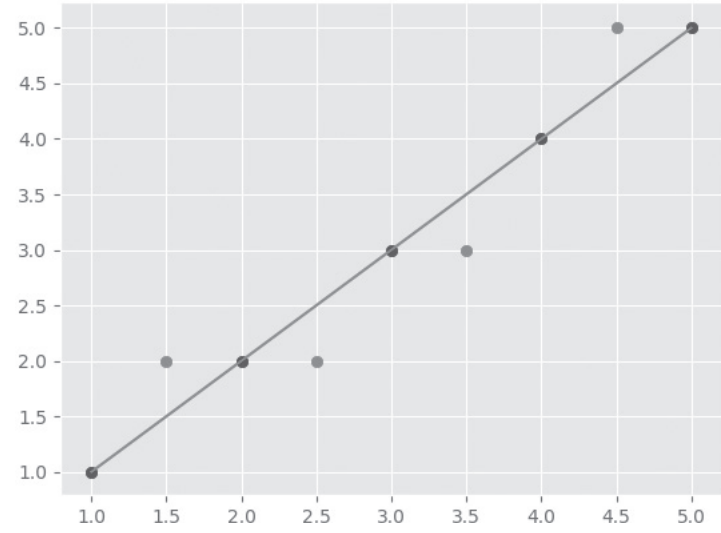

FIGURE 2.10 A Line Graph that Approximates Points of a Scatter Plot.

The MSE for the line in Figure 2.10 is computed as follows:

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

Thus, the line in Figure 2.10 has a smaller MSE than the line in Figure 2.9, 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 is significantly more difficult. For instance, if we plot 10 points in the Euclidean plane that do not closely fit a line, with individual terms that involve non-integer values, we would 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 non-zero 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 2.22 displays the contents of `plot-best-fit2.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 2.22: plot-best-fit2.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**xs)))
  b = np.mean(ys) - m * np.mean(xs)

  return m, b

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

Listing 2.22 starts with two `NumPy` arrays `xs` and `ys` that are initialized with the first five positive integers. The Python function `best_fit_slope()` calculate the optimal values of `m` (the slope) and `b` (the y-intercept) of a set of numbers. The output from Listing 2.22 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.

Figure 2.11 displays another line segment that approximates scatter plot consisting of a larger number of points.

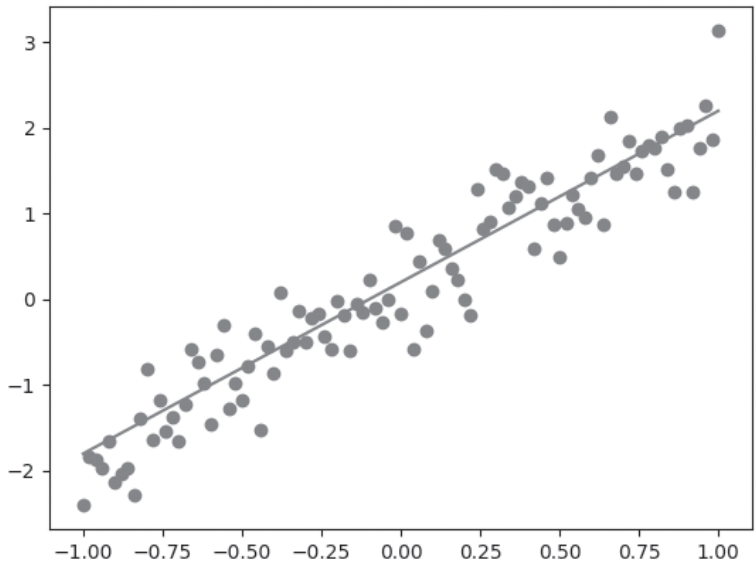

FIGURE 2.11 A Line Graph that Approximates a Generalized Scatter Plot.

If you are really interested, you can search online to find the derivation for the values of `m` and `b`. In this chapter, we are 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 2.23 displays the contents of `plain-linreg1.py` that illustrates how to compute the MSE with simulated data.

LISTING 2.23: 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]

costs = []
#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 Cost
  Y_pred = np.multiply(W, X) + b
  loss_error = 0.5 * (Y_pred - Y)**2
  cost = np.sum(loss_error)/10

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

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

  if i%10 == 0:
    print("Cost at", i,"iteration = ", cost)

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

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

Listing 2.23 defines the variables X and Y that are simple arrays of numbers (this is our dataset). Next, the costs array is initialized as an empty array, and we will append successive cost 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 2.23 is a for loop that executes 100 times. During each iteration, the variables `Y_pred`, `loss_error`, and `cost` are computed, and they correspond to the predicted value, the error, and the cost, respectively (remember: we are performing linear regression). The value of `cost` (which is the error for the current iteration) is then appended to the `costs` 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 cost, 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 cost values on the vertical axis and the loop iterations on the horizontal axis. The output from Listing 2.23 is here:

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

Figure 2.12 displays the plot of cost-versus-iterations for Listing 2.23.

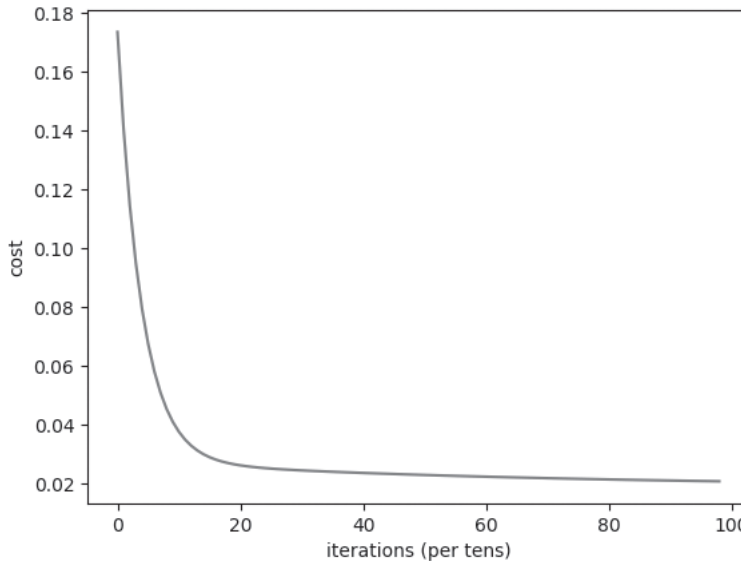

FIGURE 2.12 A Plot of Cost-Versus-Iterations.

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 do not already know, the number of epochs specifies the number of times that an inner loop is executed.

Listing 2.24 displays the contents of `plain-linreg2.py` that illustrates how to compute the MSE with simulated data.

LISTING 2.24: 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()

costs = []
#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 Cost
    Y_pred = np.multiply(W, X) + b
    Loss_error = 0.5 * (Y_pred - Y)**2
    cost = np.sum(Loss_error)/10

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

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

    if i%50 == 0:
      print("Cost at epoch", j,"= ", cost)

#Plot cost versus # of iterations
print("W = ", W,"& b = ",  b)
plt.plot(costs)
```

```
plt.ylabel('cost')
plt.xlabel('iterations (per tens)')
plt.show()
```

Compare the new contents of Listing 2.24 (shown in bold) with the contents of Listing 2.23; 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 2.24 is here:

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

Figure 2.13 displays the plot of cost-versus-iterations for Listing 2.24.

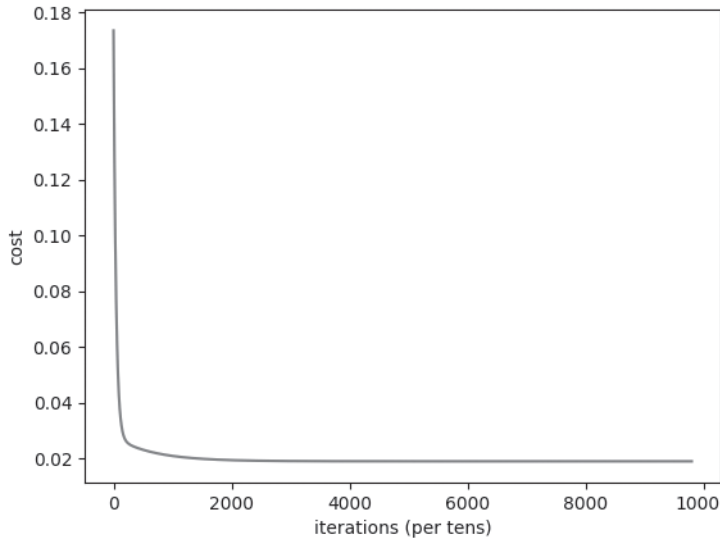

FIGURE 2.13 A Plot of Cost-Versus-Iterations.

Notice that Figure 2.13 has 10,000 iterations on the horizontal axis, whereas Figure 2.12 has only 100 iterations on the horizontal axis.

SUMMARY

This chapter introduced you to the `NumPy` package 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 4), 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 mean squared error (MSE), and how to calculate MSE with the `NumPy linspace()` method.

CHAPTER 3

INTRODUCTION TO PANDAS

This chapter starts with an introduction to the Pandas package for Python that provides a rich and powerful set of APIs for managing datasets. These APIs are very useful for Machine Learning and Deep Learning tasks that involve dynamically "slicing and dicing" subsets of datasets.

The first part of this chapter briefly describes Pandas and some of its useful features. This section contains code samples that illustrate some nice features of DataFrames and a brief discussion of series, which are two of the main features of Pandas. 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.

The second section of this chapter shows you how to manipulate the contents of DataFrames with various operations. In particular, you will also see code samples that illustrate how to create Pandas DataFrames from CSV files, Excel spreadsheets, and data that is retrieved from a URL. The third section of this chapter gives you an overview of important data cleaning tasks that you can perform with Pandas APIs.

The final section of this chapter introduces you to Jupyter, which is a Python-based application for displaying and executing Python code in a browser. You will also learn about the Google Colaboratory environment, which is fully online and supports Jupyter notebooks and provides 12 hours of daily GPU usage for free.

After you have completed this chapter, glance through the follow blog post that discusses an initiative for parallelizing Pandas, as well as a chart containing the most frequently used Pandas APIs in Kaggle competitions:

https://rise.cs.berkeley.edu/blog/pandas-on-ray-early-lessons

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 2.x:

```
pip install pandas
```

Launch this command to install Pandas for Python 3.x:

```
pip3 install pandas
```

In many ways the Pandas package has the semantics of a spreadsheet, and it also works with xsl, xml, html, and csv file types. Pandas provides a data type called a DataFrame (similar to a Python dictionary) with extremely powerful functionality, which is discussed in the next section.

Pandas DataFrames support a variety of input types, such as ndarrays, lists, dicts, or Series. Pandas also provides another data type called Pandas Series (briefly discussed in this chapter); this data structure provides another mechanism for managing data (search online for more details).

Pandas Dataframes

In simplified terms, a Pandas DataFrame is a two-dimensional data structure, and it is 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.

By way of analogy, it might be useful to think of a DataFrame as the counterpart to a spreadsheet, which makes it a very useful data type in Pandas-related Python scripts. The source of the dataset can be a data file, database tables, web service, and so forth. Pandas DataFrame features include:

Data Frame Methods
Data Frame Statistics
Grouping, Pivoting, and Reshaping
Dealing With Missing Data
Joining Data Frames

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
- Rename columns

A LABELED PANDAS DATAFRAME

Listing 3.1 displays the contents of `pandas-labeled-df.py` that illustrates how to define a `Pandas DataFrame` whose rows and columns are labeled.

LISTING 3.1: pandas-labeled-df.py

```
import numpy
import pandas

myarray = numpy.array([[10,30,20], [50,40,60],
                       [1000,2000,3000]])

rownames = ['apples', 'oranges', 'beer']
colnames = ['January', 'February', 'March']

mydf = Pandas.DataFrame(myarray, index=rownames,
                                          columns=colnames)

print(mydf)
print(mydf.describe())
```

Listing 3.1 contains two important 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 data in `myarray`. Next, the variable `mydf` is initialized as a `Pandas DataFrame` with the specified datasource (i.e., `myarray`).

You might be surprised to see that the first portion of the output below 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 3.1 is 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  3000.000000
```

PANDAS NUMERIC DATAFRAMES

Listing 3.2 displays the contents of `pandas-numeric-df.py` that illustrates how to define a `Pandas DataFrame` whose rows and columns are numbers (but the column labels are characters).

LISTING 3.2: pandas-numeric-df.py

```
import pandas as pd

df1 = pd.DataFrame(np.random.randn(10, 4),
                                    columns=['A','B','C','D'])
df2 = pd.DataFrame(np.random.randn(7, 3),
                                       columns=['A','B','C'])
df3 = df1 + df2
```

The essence of Listing 3.2 involves initializing the `DataFrames df1` and `df2`, and then defining the `DataFrame df3` as the sum of `df1` and `df2`. The output from Listing 3.2 is here:

```
        A       B       C   D
0  0.0457 -0.0141  1.3809 NaN
1 -0.9554 -1.5010  0.0372 NaN
2 -0.6627  1.5348 -0.8597 NaN
3 -2.4529  1.2373 -0.1337 NaN
4  1.4145  1.9517 -2.3204 NaN
5 -0.4949 -1.6497 -1.0846 NaN
6 -1.0476 -0.7486 -0.8055 NaN
7     NaN     NaN     NaN NaN
8     NaN     NaN     NaN NaN
9     NaN     NaN     NaN NaN
```

Keep in mind that the default behavior for operations involving a `DataFrame` and `Series` is to align the `Series` index on the `DataFrame` columns; this results in a row-wise output. Here is a simple illustration:

```
names = pd.Series(['SF', 'San Jose', 'Sacramento'])
sizes = pd.Series([852469, 1015785, 485199])

df = pd.DataFrame({ 'Cities': names, 'Size': sizes })
df = pd.DataFrame({ 'City name': names,'sizes': sizes })

print(df)
```

The output of the preceding code block is here:

```
    City name    sizes
0          SF   852469
1    San Jose  1015785
2  Sacramento   485199
```

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 3.3 displays the contents of pandas-boolean-df.py that illustrates how to define a Pandas DataFrame whose rows and columns are Boolean values.

LISTING 3.3: 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 3.3 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 3.3 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 `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 3.4 displays the contents of `pandas-random-df.py` that illustrates how to create a `Pandas DataFrame` with random numbers.

LISTING 3.4: 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 3.4 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 3.4 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
```

COMBINING PANDAS DATAFRAMES (1)

Listing 3.5 displays the contents of `Pandas-combine-df.py` that illustrates how to combine `Pandas DataFrame`s.

LISTING 3.5: 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 3.5 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 3.5 displays the contents of `df` and `foo1`. The output of Listing 3.5 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
```

COMBINING PANDAS DATAFRAMES (2)

`Pandas` supports the "concat" method in `DataFrame`s in order to concatenate `DataFrame`s. Listing 3.6 displays the contents of `weather-data.py` that illustrates how to combine two `Pandas DataFrame`s.

LISTING 3.6: weather-data.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 3.6 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 USA, respectively. The Pandas dataframe df is the concatenation of `can_weather` and `us_weather`. The output from Listing 3.6 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 (1)

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, and also the overall profit/loss.

Listing 3.7 displays the contents of `pandas-quarterly-df1.py` that illustrates how to define a `Pandas DataFrame` consisting of income-related values.

LISTING 3.7: 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 3.7 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 DataFrames`.

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 3.7 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 us 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, and also the overall profit/loss.

Listing 3.8 displays the contents of `pandas-quarterly-df1.py` that illustrates how to define a `Pandas DataFrame` consisting of income-related values.

LISTING 3.8: 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 3.8 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 3.8 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 us 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 3.9 displays the contents of `pandas-quarterly-df1.py` that illustrates how to define a `Pandas DataFrame` consisting of income-related values.

LISTING 3.9: 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 3.9 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 3.9 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 is 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` function `read_csv()` function (shown in this section) to read the contents of CSV files.

Listing 3.10 displays the contents of `weather-data.py` that illustrates how to read a CSV file, initialize a `Pandas DataFrame` with the contents of that CSV file, and display various subsets of the data in the `Pandas DataFrames`.

LISTING 3.10: 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 3.10 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 3.10 is here:

```
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
```

In some situations you might need to apply Boolean conditional logic to "filter out" some rows of data, based on a conditional condition that is applied to a column value.

Listing 3.11 displays the contents of the CSV file `people.csv` and Listing 3.12 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 3.11: people.csv

```
fname,lname,age,gender,country
john,smith,30,m,usa
jane,smith,31,f,france
jack,jones,32,f,france
dave,stone,33,f,france
sara,stein,34,f,france
eddy,bower,35,f,france
```

LISTING 3.12: 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 3.12 populate the `Pandas` dataframe `df` with the contents of the CSV file `people.csv`. The next portion of Listing 3.12 displays the structure of `df`, followed by the first names of all the people. The next portion of Listing 3.12 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 3.12 displays a tabular list of two rows containing all the details of the people who are over 33. The output from Listing 3.12 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 (1)

Listing 3.13 displays the contents of `people-xslx.py` that illustrates how to read data from an Excel spreadsheet and create a `Pandas DataFrame` with that data.

LISTING 3.13: people-xslx.py

```
import pandas as pd

df = pd.read_excel("people.xlsx")
print("Contents of Excel spreadsheet:")
print(df)
```

Listing 3.13 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 3.13 is here:

```
  fname  lname  age gender country
0  john  smith   30      m     usa
1  jane  smith   31      f  france
2  jack  jones   32      f  france
```

```
3  dave  stone   33      f  france
4  sara  stein   34      f  france
5  eddy  bower   35      f  france
```

PANDAS DATAFRAMES AND EXCEL SPREADSHEETS (2)

Listing 3.14 displays the contents of `employees-xslx.py` that illustrates how to read data from an Excel spreadsheet and create a `Pandas DataFrame` with that data.

LISTING 3.14: employees-xslx.py

```
import pandas as pd

df = pd.read_excel("employees.xlsx")
print("Contents of Excel spreadsheet:")
print(df)

print("Q1 sum, mean, min, max:")
print(df["q1"].sum(), df["q1"].mean(),df["q1"].
                                        min(),df["q1"].max())

print("Q2 sum, mean, min, max:")
print(df["q2"].sum(), df["q2"].mean(),df["q2"].
                                        min(),df["q2"].max())

print("Q3 sum, mean, min, max:")
print(df["q3"].sum(), df["q3"].mean(),df["q3"].
                                        min(),df["q3"].max())

print("Q4 sum, mean, min, max:")
print(df["q4"].sum(), df["q4"].mean(),df["q4"].
                                        min(),df["q4"].max())

sum_col=df[["q1","q2","q3","q4"]].sum()
print("Quarter totals:")
print(sum_col)
df = pd.read_excel("people.xlsx")
print("Contents of Excel spreadsheet:")
print(df)
```

Listing 3.14 starts by reading the contents of the spreadsheet `people.xlsx` into the `Pandas` dataframe `df`, just as you saw in Listing 3.13. The rest of Listing 3.14 displays various statistical values, such as the sum, mean, min, and max values of quarter 1, quarter 3, and quarter 4. The output from Listing 3.14 is here:

```
Contents of Excel spreadsheet:
     id fname  lname gender      title     q1     q2
                                          q3     q4 country
0  1000  john  smith      m  marketing  20000  12000
                                       18000  25000     usa
```

```
1  2000  jane  smith      f  developer  30000  15000
                                        11000  35000  france
2  3000  jack  jones      m      sales  10000  19000
                                        12000  15000     usa
3  4000  dave  stone      m    support  15000  17000
                                        14000  18000  france
4  5000  sara  stein      f    analyst  25000  22000
                                        18000  28000   italy
5  6000  eddy  bower      m  developer  14000  32000
                                        28000  10000  france
Q1 sum, mean, min, max:
114000 19000.0 10000 30000
Q2 sum, mean, min, max:
117000 19500.0 12000 32000
Q3 sum, mean, min, max:
101000 16833.333333333332 11000 28000
Q4 sum, mean, min, max:
131000 21833.333333333332 10000 35000
Quarter totals:
q1    114000
q2    117000
q3    101000
q4    131000
dtype: int64
```

PANDAS DATAFRAMES AND SIMULATED DATASETS

So far you have seen code samples with hard-coded data inside the Python scripts and also data from a CSV file. In this section, you will see an example of working with simulated data (i.e., data generated by an external source).

As a simple illustration, the following data is from the first five rows of data in a file that contains simulated data:

```
  first_name last_name            company_name   address  \

0      James      Butt       Benton, John B Jr    6649 N
                                                 Blue Gum St
1  Josephine   Darakjy   Chanay, Jeffrey A Esq   4 B Blue
                                                  Ridge Blvd
2        Art    Venere     Chemel, James L Cpa  8 W
                                               Cerritos Ave #54
3      Lenna  Paprocki  Feltz Printing Service  639 Main St
4    Donette    Foller     Printing Dimensions  34 Center St

          city      county state   zip   phone1   phone2  \
0  New Orleans     Orleans    LA  70116  504-621-8927
                                                 504-845-1427
1     Brighton  Livingston    MI  48116  810-292-9388
                                                 810-374-9840
2   Bridgeport  Gloucester    NJ   8014  856-636-8749
                                                 856-264-4130
```

```
3    Anchorage   Anchorage    AK  99501  907-385-4412
                                                907-921-2010
4     Hamilton      Butler    OH  45011  513-570-1893
                                                513-549-4561

                            email                          web
0                jbutt@gmail.com          http://
                                        www.bentonjohnbjr.com
1  josephine_darakjy@darakjy.org      http://
                                     www.chanayjeffreyaesq.com
2                 art@venere.org        http://
                                       www.chemeljameslcpa.com
3          lpaprocki@hotmail.com  http://
                                  www.feltzprintingservice.com
4         donette.foller@cox.net    http://
                                   www.printingdimensions.com
```

Notice the backslash ("\") character that appears twice in the preceding output: this symbol is a continuation character, which means that the output is too wide to fit on a single line.

READING DATA FILES WITH DIFFERENT DELIMITERS

This section contains an example of reading a text file that contains different delimiters: some rows use a space as a delimiter, whereas other rows start with a space and also use a colon ":" as well as a space as a separator.

Listing 3.15 displays the contents of `multiple-delims.dat` that contains data rows with different delimiters, followed by Listing 3.16 that displays the contents of `multiple-delims.py` that reads the contents of `multiple-delims.dat` into a `Pandas DataFrame`.

LISTING 3.15: multiple-delims.dat

```
c stuff
c more header
c begin data
 1 1:.5
 1 2:6.5
 1 3:5.3
```

LISTING 3.16: multiple-delims.py

```
import pandas as pd

df = pd.read_csv('multidelim.dat', skiprows=3,
                 names=['a', 'b', 'c'],
                 sep=' |:', engine='python')

print("dataframe:")
print(df)
print(data.head())
```

Listing 3.16 invokes the `Pandas read_csv()` function to read the contents of `multidelim.dat` into the `Pandas` dataframe df. Compare the output shown below with the contents of Listing 3.15 to understand the code in Listing 3.16:

```
dataframe:
   a  b    c
0  1  1  0.5
1  1  2  6.5
2  1  3  5.3
```

TRANSFORMING DATA WITH THE SED COMMAND (OPTIONAL)

The preceding section contains an example of a data file with different delimiters, but there is a limitation: the first set of rows must have the same type and the second set of rows must also be of the same type.

However, you might have a more heterogeneous dataset with a set of rows in random order, where each row contains multiple delimiters. The solution in this section involves three files: an initial randomized dataset `multiple-delims2.dat`, a shell script `multiple-delims2.sh` for creating a "clean" dataset called `multiple-delims2b.dat`, and a Python script `multiple-delims2.py` that reads the data in `multiple-delims2b.dat` into a `Pandas DataFrame`.

Listing 3.17 displays the contents of `multiple-delims2.dat` that contains a mixture of delimiters in multiple rows (in random order).

LISTING 3.17: multiple-delims2.dat

```
1000|Jane:Edwards^Sales
2000:Tom:Smith^Development
3000|Dave:Del Ray^Marketing
4000^Steven^Andrews:Marketing
```

Listing 3.18 displays the contents of the shell script `multiple-delims.sh` that transforms `multiple-delims2.dat` into the dataset `multiple-delims2b.dat`, where the latter dataset has only a comma "," as a delimiter between columns in every row.

LISTING 3.18: multiple-delims2.sh

```
inputfile="multiple-delims2.dat"
cat $inputfile | sed -e 's/:/,/' -e 's/|/,/' -e 's/\^/,/g'
```

Listing 3.18 specifies the name of a text file whose contents are "piped" to the Unix `sed` command that replaces all occurrences of the characters ",", "|", and "^" with a comma ",". The trailing `g` in the `sed` command ensures that the replacement is performed globally. The resulting output will contain only a "," as a delimiter (shown in Listing 3.19).

Open a command shell and navigate to the directory that contains the shell script in Listing 3.18 and execute the following pair of commands:

```
chmod +x multiple-delims2.sh
./multiple-delims2.sh > multiple-delims2b.dat
```

Listing 3.19 displays the contents of `multiple-delims2b.dat` that you created in the preceding step.

LISTING 3.19: multiple-delims2b.dat

```
1000,Jane,Edwards,Sales
2000,Tom,Smith,Development
3000,Dave,Del Ray,Marketing
4000,Steven,Andrews,Marketing
```

Listing 3.20 displays the contents of `multiple-delims2b.py` that reads the contents of `multiple-delims2b.dat` into a `Pandas DataFrame`.

LISTING 3.20: multiple-delims2b.py

```
import pandas as pd

df = pd.read_csv('multiple-delims2b.dat',
                 names=['a', 'b', 'c', 'd'],
                 sep=',', engine='python')

print("dataframe:")
print(df)
```

Listing 3.20 imports `pandas` and then initializes the variable `df` with the contents of the text file `multiple-delims2b.dat`. The output from launching the code in Listing 3.20 is here:

```
dataframe:
      a       b        c            d
0  1000    Jane  Edwards        Sales
1  2000     Tom    Smith  Development
2  3000    Dave  Del Ray    Marketing
3  4000  Steven  Andrews    Marketing
```

Once again, the "heavy lifting" is performed by the cryptic-looking `sed` command in the shell script `multiple-delims2.sh`, which is in Chapter 4 of the following book: *Data Cleaning Pocket Primer* (ISBN-13: 978-1683922179)

The preceding book contains a detailed explanation of the `sed` command that will enable you to understand the contents of `multiple-delims2.sh`, as well as chapters that discuss the `grep` and `awk` commands and numerous examples of how to use them for various data cleaning tasks.

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 sequence of operations on the contents of the dataframe `df`:

```
df['three'] = df['one'] * df['two']
df['flag'] = df['one'] > 2
print(df)
```

The output from the preceding code block is here:

```
   one  two  three   flag
a  1.0  1.0    1.0  False
b  2.0  2.0    4.0  False
c  3.0  3.0    9.0   True
d  NaN  4.0    NaN  False
```

Columns can be deleted or popped, like with a Python `dict`, as shown in the following code snippet:

```
del df['two']
three = df.pop('three')
print(df)
```

The output from the preceding code block is here:

```
   one   flag
a  1.0  False
b  2.0  False
c  3.0   True
d  NaN  False
```

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   flag  foo
a  1.0  False  bar
b  2.0  False  bar
c  3.0   True  bar
d  NaN  False  bar
```

When inserting a Series that does not have the same index as the `DataFrame`, it will be "conformed" to the index of the `DataFrame`:

```
df['one_trunc'] = df['one'][:2]
print(df)
```

The output from the preceding code snippet is here:

```
   one   flag  foo  one_trunc
a  1.0  False  bar        1.0
b  2.0  False  bar        2.0
c  3.0   True  bar        NaN
d  NaN  False  bar        NaN
```

You can insert raw `ndarrays` but their length must match the length of the index of the `DataFrame`.

PANDAS DATAFRAMES AND SCATTERPLOTS

Listing 3.21 displays the contents of `pandas-scatter-df.py` that illustrates how to generate a scatterplot from a `Pandas DataFrame`.

LISTING 3.21: 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 3.21 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 3.21 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 HISTOGRAMS

Listing 3.22 displays the contents of `pandas-histograms.py` that illustrates how to generate histograms from a `Pandas DataFrame`.

LISTING 3.22: pandas-histograms.py

```
import pandas as pd

df = pd.read_csv("Housing.csv")

print(df.head())
print(df.info())
print(df.describe())

import matplotlib.pyplot as plt
df.hist(bins=50, figsize=(20,15))
#save_fig("housing_histograms")
plt.show()
```

Listing 3.17 initializes the `Pandas DataFrame df` with the contents of the CSV file `Housing.csv`. Next, various portions of `df` are displayed, such as the first five rows and information about the structure of `df`.

The next portion of Listing 3.17 imports the `plt` class so that we can display a scatterplot of the data in `df`: this is done by invoking the `hist()` method of the `df` variable, followed by the `plt.show()` command that actually displays the scatter plot. The output from Listing 3.17 is here:

```
   Unnamed: 0    price  lotsize  bedrooms  bathrms  stories
                                                  driveway recroom  \
0           1  42000.0     5850         3        1
                                           2      yes      no
```

```
1           2  38500.0     4000         2        1
                                          1      yes      no
2           3  49500.0     3060         3        1
                                          1      yes      no
3           4  60500.0     6650         3        1
                                          2      yes     yes
4           5  61000.0     6360         2        1
                                          1      yes      no

  fullbase gashw airco  garagepl prefarea
0      yes    no    no         1       no
1       no    no    no         0       no
2       no    no    no         0       no
3       no    no    no         0       no
4       no    no    no         0       no
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 546 entries, 0 to 545
Data columns (total 13 columns):
Unnamed: 0    546 non-null int64
price         546 non-null float64
lotsize       546 non-null int64
bedrooms      546 non-null int64
bathrms       546 non-null int64
stories       546 non-null int64
driveway      546 non-null object
recroom       546 non-null object
fullbase      546 non-null object
gashw         546 non-null object
airco         546 non-null object
garagepl      546 non-null int64
prefarea      546 non-null object
dtypes: float64(1), int64(6), object(6)
memory usage: 55.5+ KB
None
       Unnamed: 0          price       lotsize
                                     bedrooms     bathrms  \
count  546.000000     546.000000    546.000000
                                      546.000000  546.000000
mean   273.500000   68121.597070   5150.265568
                                        2.965201    1.285714
std    157.760895   26702.670926   2168.158725
                                        0.737388    0.502158
min      1.000000   25000.000000   1650.000000
                                        1.000000    1.000000
25%    137.250000   49125.000000   3600.000000
                                        2.000000    1.000000
50%    273.500000   62000.000000   4600.000000
                                        3.000000    1.000000
75%    409.750000   82000.000000   6360.000000
                                        3.000000    2.000000
max    546.000000  190000.000000  16200.000000
                                        6.000000    4.000000

          stories    garagepl
count  546.000000  546.000000
mean     1.807692    0.692308
```

```
std       0.868203    0.861307
min       1.000000    0.000000
25%       1.000000    0.000000
50%       2.000000    0.000000
75%       2.000000    1.000000
max       4.000000    3.000000
```

Figure 3.1 displays the histograms that are generated by launching the code in Listing 3.22.

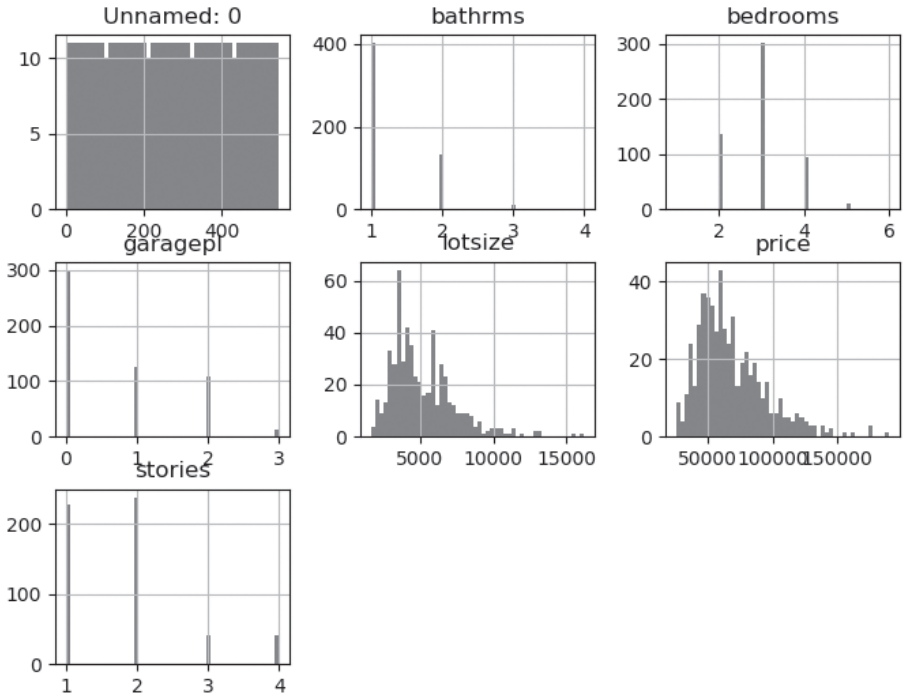

FIGURE 3.1 Histograms for the Housing.csv Dataset.

PANDAS DATAFRAMES AND SIMPLE STATISTICS

Listing 3.23 displays the contents of `housing-stats.py` that illustrates how to gather basic statistics from data in a `Pandas DataFrame`.

LISTING 3.23: 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 3.23 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 3.23 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 3.23 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 3.23 is here:

```
Apples
10
```

STANDARDIZING PANDAS DATAFRAMES

Listing 3.24 displays the contents of `pandas-standardize-df.py` that illustrates how to standardize data in a `Pandas DataFrame`.

LISTING 3.24: pandas-standardize-df.py

```
# Standardize data (0 mean, 1 stdev)
from sklearn.preprocessing import StandardScaler
from pandas import read_csv
import numpy

url = 'https://goo.gl/bDdBiA'
names = ['preg','plas','pres','skin','test','mass','pedi','
                                                    age','class']
```

```
dataframe = read_csv(url, names=names)
array = dataframe.values

# separate array into input and output components
X = array[:,0:8]
Y = array[:,8]
scaler = StandardScaler().fit(X)
rescaledX = scaler.transform(X)

# summarize transformed data
numpy.set_printoptions(precision=3)
print(rescaledX[0:5,:])
```

Listing 3.24 imports the `StandardScaler` class from the `Sklearn` package in order to rescale data values so that they have a mean of 0 and a standard deviation of 1.

Next, the variable `url` is initialized with the location of a website that returns CSV-based data. The `names` variable contains an array of column names that are used to label the columns of the CSV-based data. Next, the variable `dataframe` is initialized with the contents of the CSV-based data (retrieved from the location specified by the `url` variable).

The next portion of Listing 3.24 initializes the variable array with the values in the variable `dataframe`. Next, the variable `X` is initialized with the left-most eight columns of every row in the variable array, and the variable y is initialized with the data in the ninth column of the variable array. The next portion of Listing 3.24 invokes the fit method of the `StandardScaler` class in order to fit the data contained in `X`, and the result is used to initialize the variable `scaler`. The next statement invokes the `transform()` method on the contents of `X` and the results are used to initialize the variable `rescaledX`, which concludes the required data transformations (finally!)

The final portion of Listing 3.24 displays all the columns of the first five rows of the variable `scaler`. The output from Listing 3.24 is here:

```
minimum # of bedrooms: 1
median  # of bedrooms: 3.0
maximum # of bedrooms: 6

median values: [2.735e+02 6.200e+04 4.600e+03 3.000e+00
                          1.000e+00 2.000e+00 0.000e+00]

first 5 prices:
0    42000.0
1    38500.0
2    49500.0
3    60500.0
4    61000.0
Name: price, dtype: float64

median price: 62000.0

correlation matrix:
price         1.000000
```

```
lotsize        0.535796
bathrms        0.516719
stories        0.421190
garagepl       0.383302
Unnamed: 0     0.376007
bedrooms       0.366447
```

PANDAS DATAFRAMES, NUMPY FUNCTIONS, AND LARGE DATASETS

Pandas DataFrames containing numeric data can be used in conjunction with NumPy functions such as log, exp, and sqrt (and various other NumPy functions). Example of such functions are shown here:

```
df.exp(df)
np.asarray(df)
matrix multiplication:
df.T.dot(df)
```

The dot method on Series implements a dot product:

```
s1 = pd.Series(np.arange(5,10))
s1.dot(s1)
```

However, a Pandas DataFrame is not intended to be a direct replacement for ndarray as some of its indexing semantics are quite different from a matrix.

Another challenge that you might face: what do you do with large datasets that exceed the memory of your machine? The solution involves a "chunking" technique for reading portions of data into memory. Chunking enables you to "stream" data from a file into a Pandas DataFrame, and you can specify the number of rows in a "chunk" of data. An example of chunking is shown here:

```
import pandas as pd
mydata = pd.DataFrame()

#Modify chunksize based on your requirements
for chunk in pd.read_csv('myfile.csv', iterator=True,
                                                chunksize=5000):
  mydata = pd.concat([mydata, chunk], ignore_index=True)
```

WORKING WITH PANDAS SERIES

A Pandas Series is a one-dimensional labeled array that can be populated with any data type: integers, strings, floating point numbers, Python objects, and so forth. The axis labels are collectively referred to as the index.

Create a Pandas Series as shown here in the Python REPL:

```
>>> s = pd.Series(data, index=index)
```

The variable `data` in the preceding code snippet can be a scalar value, a Python `dict`, an `ndarray`, and so forth. The variable `index` is a list of axis labels, which consists of different possible values, as discussed in the following subsections.

From ndarray

If the variable `data` in the code snippet below is an `ndarray`, then index must be the same length as the variable `data`:

```
>>> s = pd.Series(data, index=index)
```

However, if no index is passed, an index will be automatically created with the values `[0, ..., len(data) - 1]`. Here is another example:

```
>>> s = pd.Series(np.random.randn(5), index=['a', 'b', 'c',
                                             'd', 'e'])
>>> s
```

The output of the preceding code snippet in the Python `REPL` is here:

```
a    0.4691
b   -0.2829
c   -1.5091
d   -1.1356
e    1.2121
dtype: float64
>> s.index
```

The output of the preceding code snippet in the Python `REPL` is here:

```
Index(['a', 'b', 'c', 'd', 'e'], dtype='object')
>>> pd.Series(np.random.randn(5))
```

The output of the preceding code snippet in the Python `REPL` is here:

```
0   -0.1732
1    0.1192
2   -1.0442
3   -0.8618
4   -2.1046
dtype: float64
```

Note that `Pandas` supports non-unique index values. However, if you invoke an operation that does *not* support duplicate index values, then an exception will be raised if you specify data that has duplicate index values.

Here is an example of a Python `Series` that is instantiated from a Python `dict`:

```
>>> d = {'b' : 1, 'a' : 0, 'c' : 2}
>>> pd.Series(d)
```

The output in the Python `REPL` is here:

```
b    1
a    0
```

```
c    2
dtype: int64
```

Pandas DataFrame from Series

Listing 3.25 displays the contents of pandas-df.py that illustrates how to create a Pandas DataFrame with data from a Pandas Series.

LISTING 3.25: pandas-df.py

```
import pandas as pd

names = pd.Series(['SF', 'San Jose', 'Sacramento'])
sizes = pd.Series([852469, 1015785, 485199])

df = pd.DataFrame({ 'Cities': names, 'Size': sizes })
df = pd.DataFrame({ 'City name': names,'sizes': sizes })

print('df:',df)
```

Listing 3.25 is straightforward: the first portion initializes the Pandas Series names and sizes with cities, and zip codes, respectively. The next portion of Listing 3.25 creates the Pandas DataFrame df with the contents of the series names and sizes. The output from Listing 3.25 is here:

```
('df:',
    City name    Sizes
0          SF   852469
1    San Jose  1015785
2  Sacramento   485199)
```

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 data frame 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`: multiply all values in the "height" column of the data frame 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 data frame 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[["name", "size"]]
```

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'])
```

This concludes the Pandas-related portion of the chapter. The next section contains a brief introduction to Jupyter, which is a Flask-based Python application that enables you to execute Python code in a browser. Instead of Python scripts, you will use Jupyter "notebooks," which support various interactive features for executing Python code. In addition, your knowledge of Jupyter will be very useful when you decide to use Google Colaboratory (discussed later) that also supports Jupyter notebooks in a browser.

WHAT IS JUPYTER?

The Jupyter Notebook is an open-source Web application for creating and sharing documents. Moreover, such documents can contain a combination of code, equations, visualizations, and text. The Jupyter home page is here:

http://jupyter.org/

Jupyter is popular among data scientists, Python developers, and even physicists because it simplifies the sharing of code. Moreover, **Google Colaboratory** (later in this chapter) supports Jupyter notebooks, along with some extra functionality.

First let us take a look at some Jupyter features that are discussed in the next section.

Jupyter Features

Jupyter has gained significant traction among various communities because of its ease of use and useful functionality. Some of the features of Jupyter include:

support for multiple programming languages
support for Python2 and Python3
sharing notebooks
importing notebooks
download notebooks
produce different types of output
big data integration
multi-user version
manage users and authentication

In particular, the Jupyter Notebook supports more than 40 programming languages, including Python, R, Julia, and Scala. Notebooks can be easily shared via email, Dropbox, GitHub, and the Jupyter Notebook Viewer. Jupyter notebooks support interactive output that contains a combination of HTML, images, videos, LaTeX, and custom MIME types.

In addition, Jupyter notebooks support big data integration, such as Apache Spark, where the data has been generated from Python, R, and Scala. A multi-user version of the Jupyter notebook is also available, and it is designed for companies, classrooms, and research labs. You can also manage multiple users and authentication with OAuth, and easily deploy the Jupyter Notebook to all the users in your organization.

Launching Jupyter from the Command Line

Launching Jupyter from the command line is straightforward. First open a command shell, then navigate to the directory that contains the Jupyter notebook basic-stuff.ipynb, and then launch Jupyter with this command:

```
jupyter notebook
```

After a few moments a new browser session is automatically opened and you will see a list of the files in the current directory.

JupyterLab

JupyterLab is an interactive development environment for notebooks, code and data, that fully supports Jupyter notebooks. JupyterLab also enables you to use text editors, terminals, data file viewers, and other custom components side by side with notebooks in a tabbed work area.

JupyterLab provides a high level of integration between notebooks, documents, and activities, including:

- Drag-and-drop to reorder notebook cells and copy them between notebooks.
- Run code blocks interactively from text files (.py, .R, .md, .tex, etc.).
- Link a code console to a notebook kernel to explore code interactively without cluttering up the notebook with temporary scratch work.
- Edit popular file formats with live preview, such as Markdown, JSON, CSV, Vega, VegaLite (and others).

Develop JupyterLab Extensions

While many JupyterLab users will install additional JupyterLab extensions, some of you will want to develop your own. The extension development API is evolving during the beta release series and will stabilize in JupyterLab 1.0. To start developing a JupyterLab extension, see the JupyterLab Extension Developer Guide and the TypeScript or JavaScript extension templates.

JupyterLab itself is co-developed on top of PhosphorJS, a new JavaScript library for building extensible, high-performance, desktop-style web applications. It uses modern JavaScript technologies such as TypeScript, React, Lerna, Yarn, and webpack. Unit tests, documentation, consistent coding standards, and user experience research help us maintain a high-quality application.

GOOGLE COLABORATORY

GPU-based TensorFlow code is typically six to eight times faster than CPU-based TensorFlow code that is executed on a laptop. However, the cost 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 TensorFlow).

Fortunately there is at least one affordable alternative regarding access to GPUs, such as Google Colaboratory, which is a free Jupyter notebook environment with zero configuration required. Colaboratory (often abbreviated as Colab) runs in the cloud, and you can create and share Jupyter notebooks using most major browsers. The Colaboratory website is here:

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

This Jupyter notebook is suitable for training simple models and testing ideas quickly. Colab makes it easy to upload local files, install software in Jupyter notebooks, and even connect Colab to a Jupyter runtime on your local machine. In fact, you can directly install Github repositories in a Colaboratory notebook with a very simple syntax, and also invoke the pip command to install other software components. An example of installing a Github repository and also installing tflearn is here:

```
!git clone my-github-repository
!pip install tflearn
```

Some of the supported features of Colab include TensorFlow execution with GPUs, visualization using Matplotlib (discussed in Chapter 4), and the ability to save a copy of your Colab notebook to Github by navigating to File > Save a copy to GitHub. Moreover, you can load any notebook on GitHub by just adding the path to the URL *colab.research.google.com/github/* (see the website for details).

Colab has support for other technologies such as HTML and SVG, enabling you to render SVG-based graphics in notebooks that are in Colab. One point to keep in mind: any software that you install (or data that you upload) in a Colab 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 Colab session.

GPU Support

As mentioned earlier, there is one *very* nice feature of Colab: 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 TensorFlow code to train neural networks in less than 20 or 30 minutes that would otherwise require many hours of CPU-based execution time.

Open a browser session, navigate to `Google Colaboratory`, open a `Jupyter` notebook (or create a new one), and enable `GPU` support as follows:

1. click on the `Connect` button in the upper-right side of the screen (just once)
2. navigate to `Edit > Settings` and select `GPU` in the drop-down list

Perform the first step whenever you start a new `Google Colaboratory` session, and the second step when you create a new `Jupyter` notebook.

OTHER CLOUD PLATFORMS

GCP (Google Cloud Platform) is a cloud-based service that enables you to train TensorFlow code in the cloud. GCP provides Deep Learning DL images (similar in concept to Amazon AMIs) that are available here:

https://cloud.google.com/deep-learning-vm/docs

The preceding link provides documentation, and also a link to DL images based on different technologies, including TensorFlow and PyTorch, with `GPU` and `CPU` versions of those images. Along with support for multiple versions of Python, you can work in a browser session or from the command line.

Install GCloud SDK on a Mac-based laptop by downloading the software at this link:

https://cloud.google.com/sdk/docs/quickstart-macos

You will also receive $300 USD worth of credit (over one year) if you have never used Google cloud.

Floydhub provides access to a GPU in the cloud, with services for training and deploying applications based on frameworks such as Tensorflow, PyTorch, and Keras, and its home page is here: *https://floydhub.com*

Paperspace is a essentially a virtual desktop in the cloud, with services that support `Jupyter` Notebooks, task execution on Google TPUs, and its home page is here: *www.paperspace.com*

PipelineAI simplifies the workflow process for model development and provides a cloud-based GPU for free. `PipelineAI` facilitates the process of training and deploying models, either from a `Jupyter` Notebook or from a command-line interface. The `PipelineAI` home page is here: *http://pipeline.ai*

Commercial cloud-based services include AWS, which has a tiered pricing system (as does GCP). More details are available on the AWS website.

SUMMARY

This chapter introduced you to `Pandas` for creating labeled `Dataframes` and displaying metadata of `Pandas Dataframes`. Then you learned how to create Pandas `Dataframes` 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 min, mean, and max values in numeric columns. Then you saw how to create `Pandas Dataframe`s from data stored in CSV files. Next, you learned how to invoke a Web Service to retrieve data and populate a `Pandas Dataframe` with that data. In addition, you learned how to generate a scatterplot from data in a `Pandas Dataframe`.

Next, you saw how to use `Jupyter`, which is a Python-based application for displaying and executing Python code in a browser. In addition, you learned how to work with Google `Colaboratory`, which is a fully online environment that supports `Jupyter` notebooks and also provides 12 hours of free GPU usage on a daily basis.

CHAPTER 4

MATPLOTLIB, SKLEARN, AND SEABORN

This chapter discusses several Python packages that are useful not only for Machine Learning but also with TensorFlow (discussed in the next chapter). The examples in this chapter illustrate basic features, and in many cases, the code samples use features of `NumPy`, which is discussed in Chapter 2.

The first part of this chapter discusses `Matplotlib`, along with examples of plotting lines, histograms, and simple trigonometric functions. The second portion of this chapter introduces you to Linear Regression, and explains some of its advantages.

The third section of this chapter discusses `Sklearn`, which enables the implementation of many useful Machine Learning algorithms, such as Linear Regression, Logistic Regression, Trees, Random Forests, and so forth. This section also contains code samples with the `Iris` and `Digits` datasets, and how to process images. The final portion of this chapter discusses `Seaborn`, which is an extension of `Matplotlib`, and contains examples of plotting lines and histograms.

In order to ensure that you can launch all the code samples in this chapter, make sure you have the following installed (using `pip` or `pip3`) on your computer:

- Python 2.7+ or Python 3
- Sklearn
- Matplotlib
- Seaborn

WHAT IS MATPLOTLIB?

`Matplotlib` is a plotting library that supports `NumPy`, `SciPy`, and toolkits such as `wxPython` (among others). The `Matplotlib` support for versions of

Python is a bit complicated. Currently Matplotlib 2.0.x supports Python 2.7 through 3.6, and Matplotlib 1.2 supports Python 3.x. Moreover, Matplotlib 1.4 is the last version to support Python 2.6, and Matplotlib will not support Python 2 after 2020 (the EOL for Python 2).

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://en.wikipedia.org/wiki/Matplotlib*) 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 (non-vertical) 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 2, so you can refresh your memory regarding these APIs.

Now let us proceed with a fast-paced set of short code samples that display various types of line segments, starting with the next section.

HORIZONTAL LINES IN MATPLOTLIB

Listing 4.1 displays the contents of hlines1.py that illustrates how to plot horizontal lines using Matplotlib. Recall that the equation of a non-vertical 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 4.1: 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 4.1 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 4.1 displays three horizontal line segments whose equations are contained in Listing 4.1.

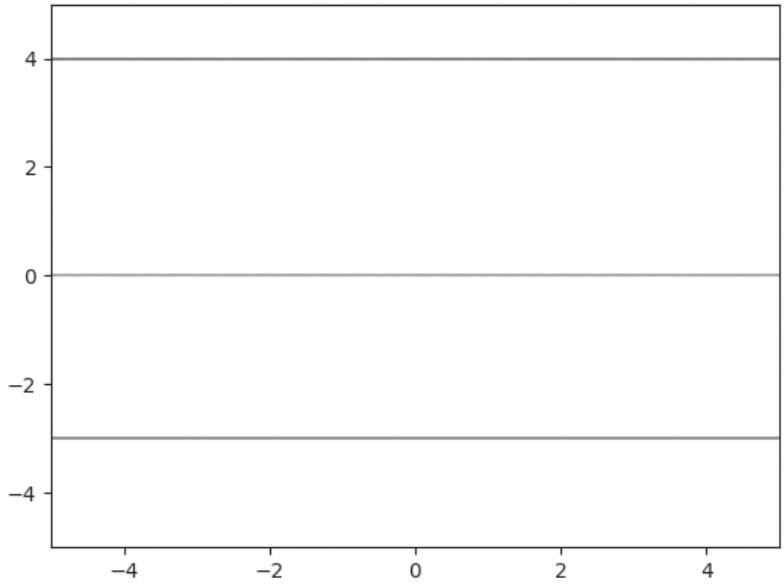

FIGURE 4.1 A Graph of Three Horizontal Line Segments.

SLANTED LINES IN MATPLOTLIB

Listing 4.2 displays the contents of `diagonallines1.py` that illustrates how to plot slanted lines.

LISTING 4.2: diagonallines1.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 4.2 defines two lines using the technique that you saw in Listing 4.1, except that these two lines define y1 = x1 and y2 = -x2, which produces slanted lines instead of horizontal lines.

Figure 4.2 displays two slanted line segments whose equations are defined in Listing 4.2.

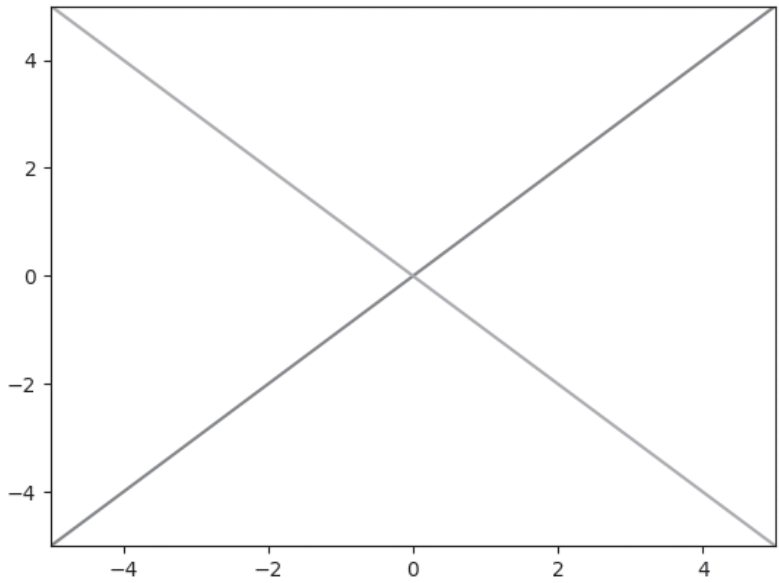

FIGURE 4.2 A Graph of Two Slanted Line Segments.

PARALLEL SLANTED LINES IN `MATPLOTLIB`

If two lines in the Euclidean plane have the same slope, then they are parallel. Listing 4.3 displays the contents of `parallellines1.py` that illustrates how to plot parallel slanted lines.

LISTING 4.3: 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 4.3 defines three lines using the technique that you saw in Listing 4.1, where these three lines are slanted and also parallel to each other.

Figure 4.3 displays two slanted and also parallel line segments whose equations are defined in Listing 4.3.

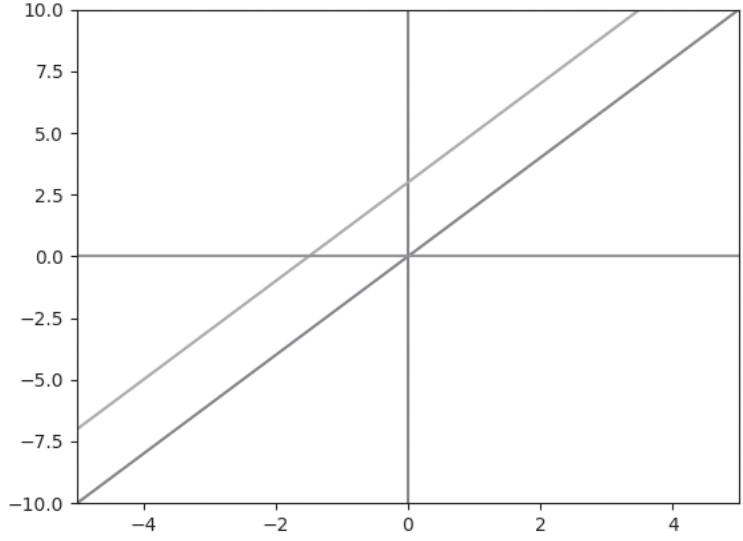

FIGURE 4.3 A Graph of Two Slanted Parallel Line Segments.

A GRID OF POINTS IN MATPLOTLIB

Listing 4.4 displays the contents of `plotgrid.py` that illustrates how to plot a simple grid.

LISTING 4.4: 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 4.4 defines the `NumPy` variable `points` that define 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 4.4 displays a grid of points as defined in Listing 4.3.

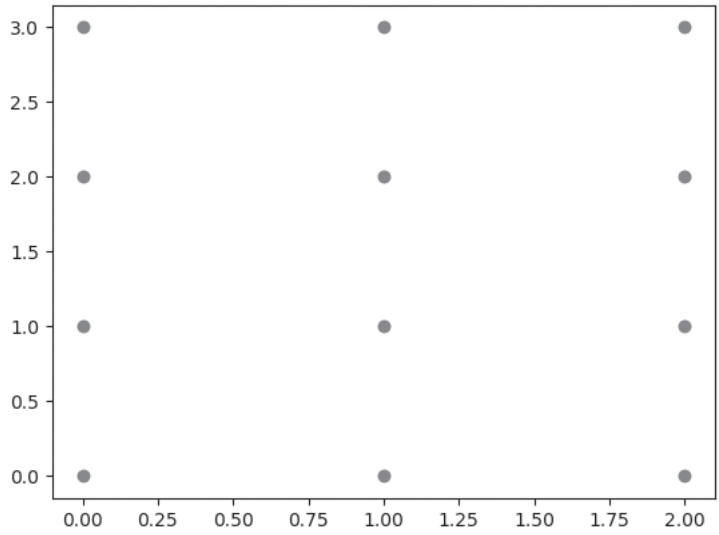

FIGURE 4.4 A Grid of Points.

A DOTTED GRID IN MATPLOTLIB

Listing 4.5 displays the contents of `plotdottedgrid1.py` that illustrates how to plot a "dotted" grid pattern.

LISTING 4.5: 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 4.5 is similar to the code in Listing 4.4 in that both of them plot a grid-like pattern; however, the former renders a "dotted" grid pattern, whereas the latter renders a "dotted" grid pattern by specifying the value `'--'` for the `linestyle` parameter.

The next portion of Listing 4.5 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 4.5 invokes the `gca().xaxis.grid(True)` chained methods to display the vertical grid lines.

Figure 4.5 displays a "dashed" grid pattern based on the code in Listing 4.5.

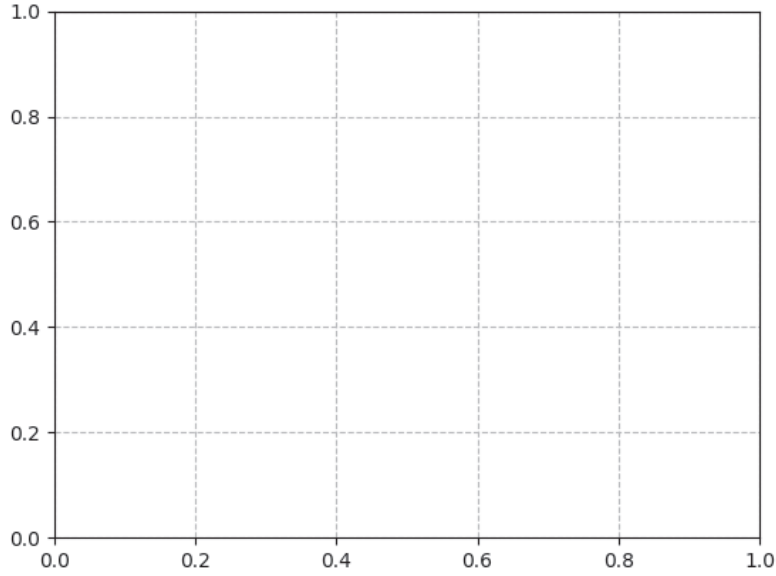

FIGURE 4.5 A "Dashed" Grid Pattern.

LINES IN A GRID IN MATPLOTLIB

Listing 4.6 displays the contents of `plotlinegrid2.py` that illustrates how to plot lines in a grid.

LISTING 4.6: 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 4.6 defines the `NumPy` variable `points` that define 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 4.6 displays a set of "dashed" line segment whose equations are contained in Listing 4.6.

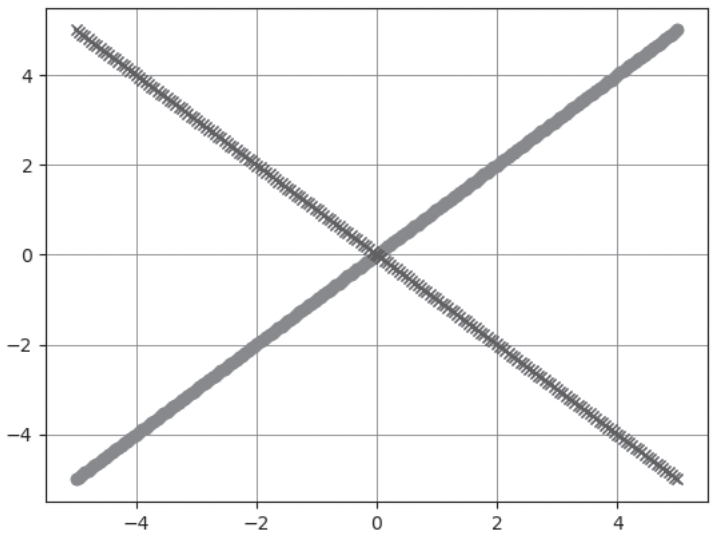

FIGURE 4.6 A Grid of Line Segments.

A COLORED GRID IN MATPLOTLIB

Listing 4.7 displays the contents of `plotgrid2.py` that illustrates how to display a colored grid.

LISTING 4.7: 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 4.7 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 4.7 displays a colored grid whose equations are contained in Listing 4.7.

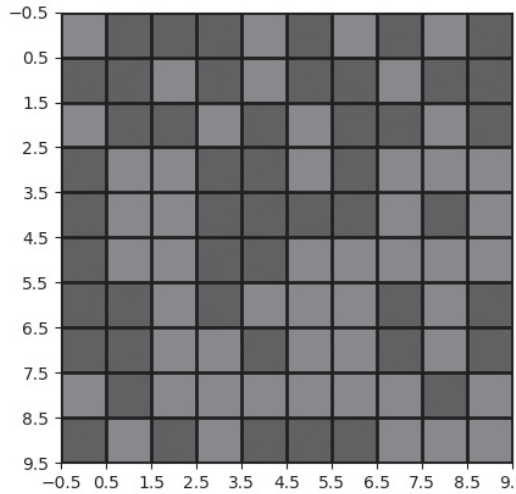

FIGURE 4.7 A Colored Grid of Line Segments.

A COLORED SQUARE IN AN UNLABELED GRID IN MATPLOTLIB

Listing 4.8 displays the contents of `plotgrid3.py` that illustrates how to display a colored square inside a grid.

LISTING 4.8: 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 4.8 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 np.`nan` values. The Pyplot API `plot()` uses the `data` variable to display a colored square inside a grid-like pattern.

Figure 4.8 displays a colored square in a grid whose equations are contained in Listing 4.8.

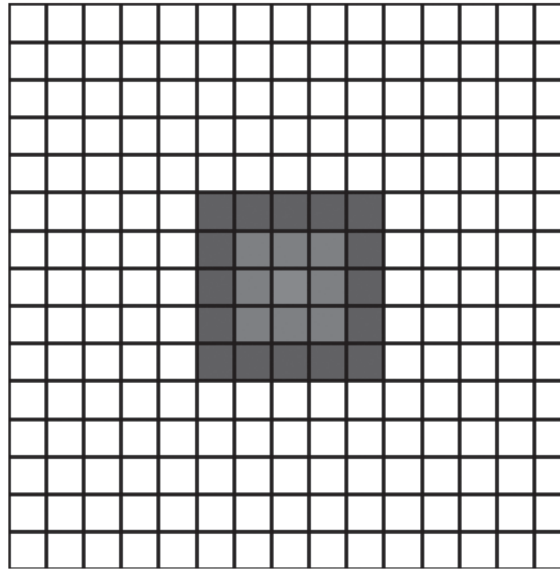

FIGURE 4.8 A Colored Square in a Grid of Line Segments.

RANDOMIZED DATA POINTS IN MATPLOTLIB

Listing 4.9 displays the contents of `lin-reg-plot.py` that illustrates how to plot a graph of random points.

LISTING 4.9: 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 4.9 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 4.9 creates a scatterplot based on the values in trX and trY, followed by the Pyplot API plot() that renders a line segment.

Figure 4.9 displays a random set of points based on the code in Listing 4.9.

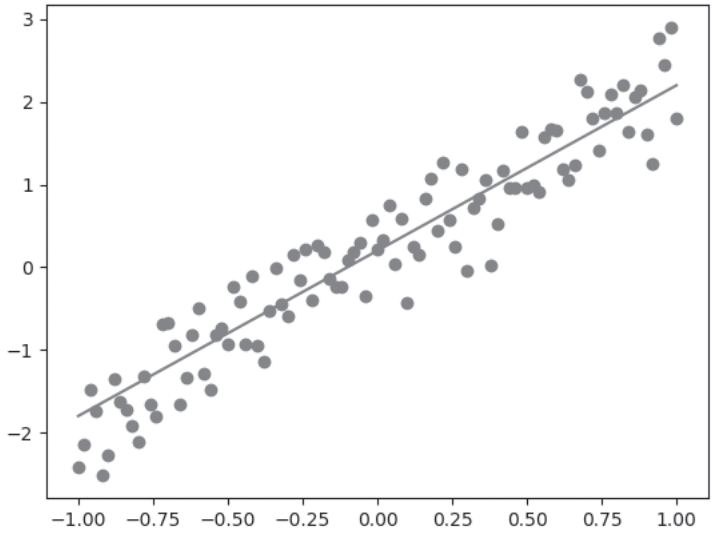

FIGURE 4.9 A Random Set of Points.

A HISTOGRAM IN MATPLOTLIB

Listing 4.10 displays the contents of histogram1.py that illustrates how to plot a histogram using Matplotlib.

LISTING 4.10: histogram1.py

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

greyhounds = 500
labradors  = 500

grey_height = 28 + 4 * np.random.randn(greyhounds)
labs_height = 24 + 4 * np.random.randn(labradors)

plt.hist([grey_height, labs_height],stacked=True,color=['r
                                                        ','b'])
plt.show()
```

Listing 4.10 is straightforward: the `NumPy` variables `grey_height` and `labs_height` contain a random set of values whose upper bound is greyhounds and labradors, respectively. The Pyplot API `hist()` uses the points `grey_height` and `labs_height` in order to display a histogram.

Figure 4.10 displays a histogram whose shape is based on the code in Listing 4.10.

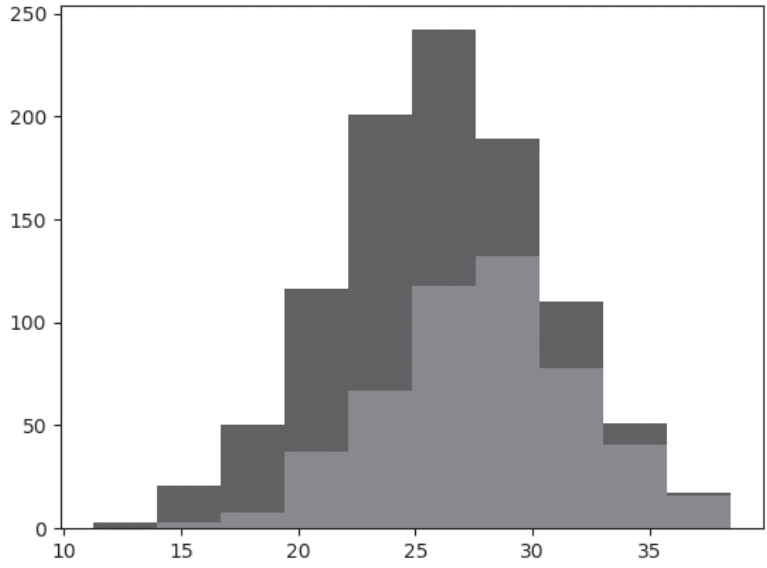

FIGURE 4.10 A Histogram Based on Random Values.

A SIMPLE LINE IN MATPLOTLIB

Listing 4.11 displays the contents of `simple-line.py` that illustrates how to plot a simple line in `Matplotlib`.

LISTING 4.11: simple-line.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 4.11 defines the array `x` that contains a hard-coded set of values. The Pyplot API `plot()` uses the variable `x` to display a line segment. Figure 4.11 displays a simple line based on the code in Listing 4.11.

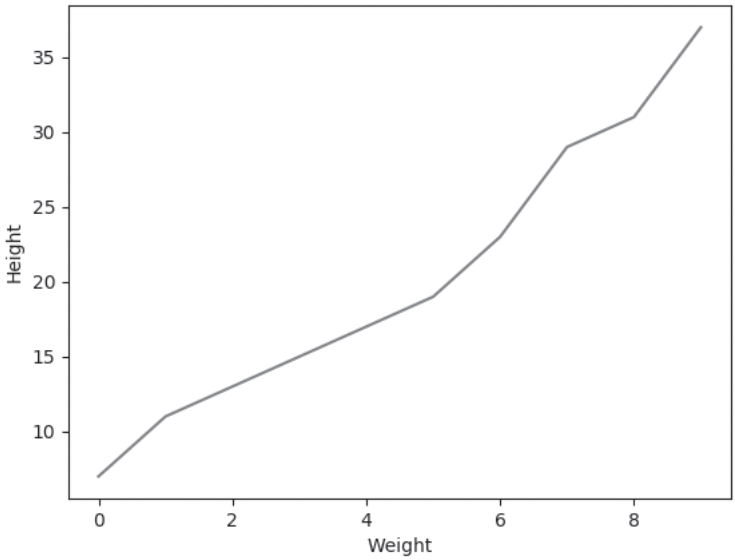

FIGURE 4.11 A Simple Line Segment.

PLOTTING MULTIPLE LINES IN MATPLOTLIB

Listing 4.12 displays the contents of `plt-array2.py` that illustrates the ease with which you can plot multiple lines in `Matplotlib`.

LISTING 4.12: 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 4.12 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 4.11 displays multiple lines based on the code in Listing 4.12.

TRIGONOMETRIC FUNCTIONS IN MATPLOTLIB

In case you were wondering, you can display the graph of trigonometric functions as easily as you can render "regular" graphs using Matplotlib. Listing 4.13 displays the contents of `sincos.py` that illustrates how to plot a sine function and a cosine function in `Matplotlib`.

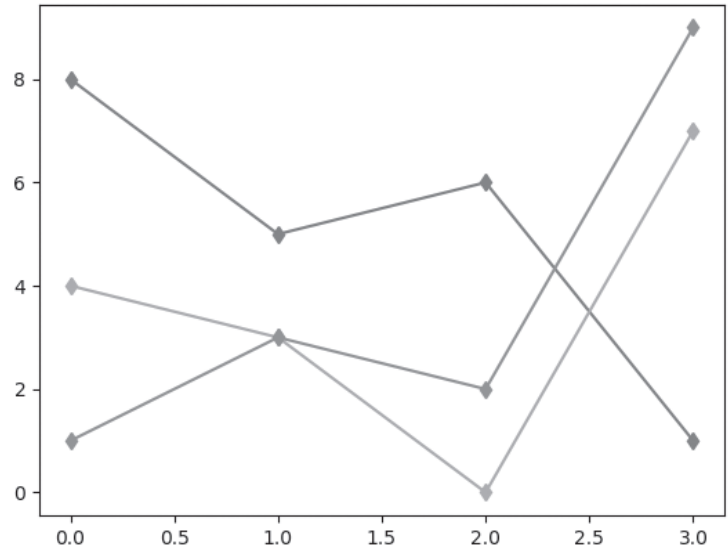

FIGURE 4.12 Multiple Lines in Matplotlib.

LISTING 4.13: 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 4.13 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 4.13 displays a graph of trigonometric functions based on the code in Listing 4.13.

Now let us look at a simple dataset consisting of discrete data points, which is the topic of the next section.

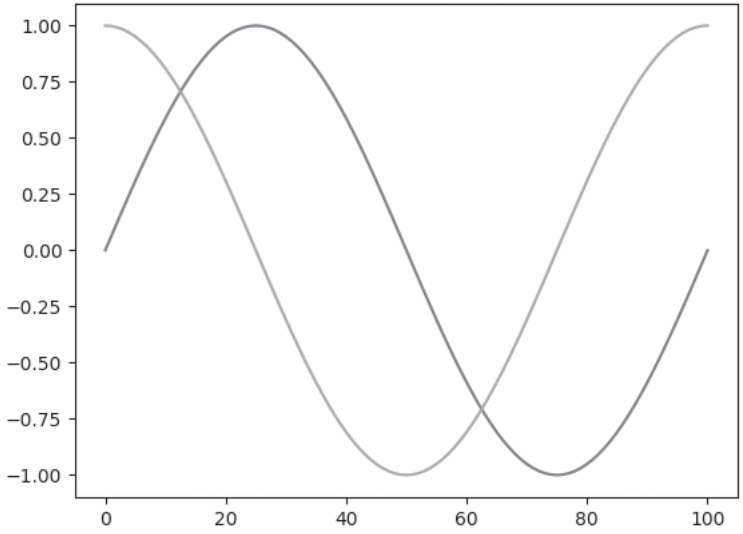

FIGURE 4.13 Trigonometric Functions.

DISPLAY IQ SCORES IN MATPLOTLIB

Listing 4.14 displays the contents of `iq-scores.py` that illustrates how to plot a histogram that displays IQ scores (based on a normal distribution).

LISTING 4.14: iq-scores.py

```
import numpy as np
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 4.14 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 4.14 displays a histogram whose shape is based on the code in Listing 4.13.

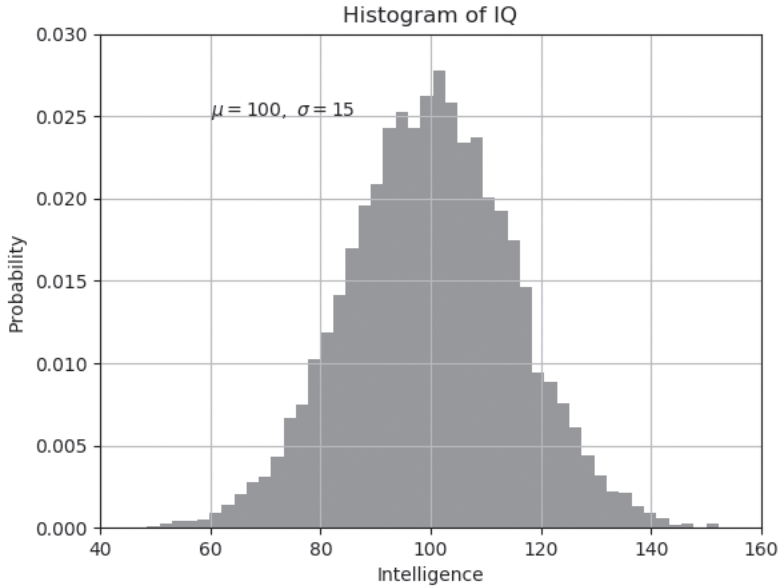

FIGURE 4.14 A Histogram to Display IQ Scores.

PLOT A BEST-FITTING LINE IN MATPLOTLIB

Listing 4.15 displays the contents of `plot-best-fit2.py` that illustrates how to plot a best-fitting line in `Matplotlib`.

LISTING 4.15: plot-best-fit2.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 4.15 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 scatter plot of the points `xs` and `ys`, followed by the `plot()` API that displays the best-fitting line. Figure 4.15 displays a simple line based on the code in Listing 4.15.

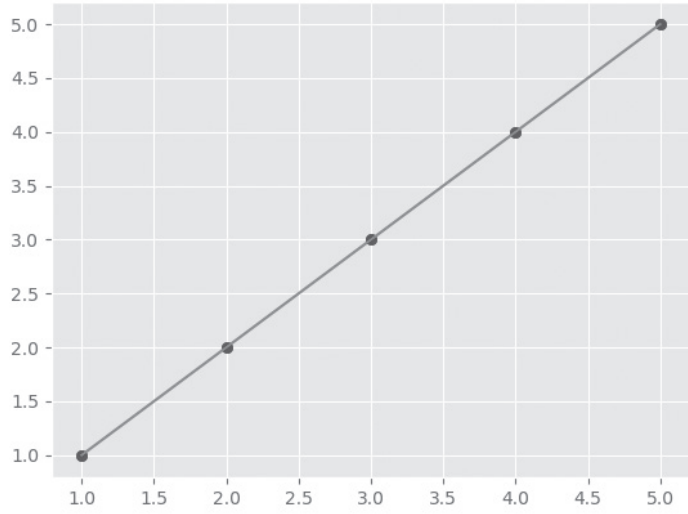

FIGURE 4.15 A Best-Fitting Line for a 2D Dataset.

LINEAR REGRESSION WITH NUMPY AND MATPLOTLIB

Listing 4.16 displays the contents of `plain-linreg1.py` that illustrates how to perform linear regression with `NumPy` and plot the result in `Matplotlib`.

LISTING 4.16: 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]

costs = []

#Step 1: Parameter initialization
W = 0.45
b = 0.75

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

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

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

  if i%10 == 0:
    print("Cost after iteration ", i,": ", cost)

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

Listing 4.16 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 scatter plot of the points `xs` and `ys`, followed by the `plot()` API that displays the best-fitting line. The text output from launching Listing 4.16 is here:

```
('Cost after iteration ', 10, ': ', 0.04114630674619492)
('Cost after iteration ', 20, ': ', 0.026706242729839392)
('Cost after iteration ', 30, ': ', 0.024738889446900423)
('Cost after iteration ', 40, ': ', 0.023850565034634254)
('Cost after iteration ', 50, ': ', 0.0231499048706651)
('Cost after iteration ', 60, ': ', 0.02255361434242207)
('Cost after iteration ', 70, ': ', 0.0220425055291673)
('Cost after iteration ', 80, ': ', 0.021604128492245713)
('Cost after iteration ', 90, ': ', 0.021228111750568435)
('W = ', 0.47256473531193927, '& b = ', 0.19578262688662174)
```

Figure 4.16 displays a best-fitting line for a set of scatter points whose shape is based on the code in Listing 4.16.

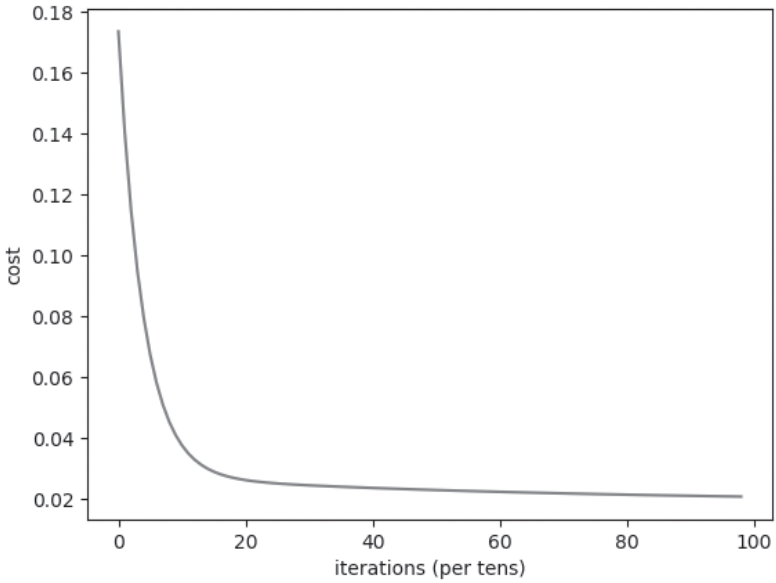

FIGURE 4.16 A Best-Fitting Line Segment.

This concludes the portion of the chapter pertaining to `NumPy` and `Matplotlib`. The next section introduces you to `scikit-learn`, which is another powerful Python-based library.

WHAT IS 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/

Machine Learning code samples often contain a combination of `Sklearn`, `NumPy`, `Pandas`, and `Matplotlib`.

`Sklearn` is well-suited for classification tasks as well as regression and clustering tasks. `Sklearn` supports a vast collection of ML algorithms, including linear regression, logistic regression, kNN ("K Nearest Neighbor"), kMeans, decision trees, random forests, MLPs (Multi-Layer Perceptrons), and SVMs (Support Vector Machines).

Moreover, `Sklearn` supports dimensionality reduction techniques such as PCA (Principal Component Analysis), "hyper parameter" tuning (this topic is discussed briefly in Chapter 5), methods for scaling data, and is suitable for preprocessing data, cross-validation, and so forth.

Keep in mind that Sklearn does not support Deep Learning models (which of course are supported in TensorFlow).

`Sklearn` also provides various built-in datasets, such as the `Digits` dataset, which is the topic of the next section.

THE DIGITS DATASET IN SKLEARN (1)

The Digits dataset in Sklearn comprises 1797 small 8x8 images; each image is a hand-written digit, which is also the case for the MNIST dataset. Listing 4.17 displays the contents of load-digits1.py that illustrates how to plot the Digits dataset.

LISTING 4.17: load-digits1.py

```
# Import 'datasets' from 'sklearn'
from sklearn import datasets

# Load in the 'digits' data
digits = datasets.load_digits()

# Print the 'digits' data
print(digits)
```

Listing 4.17 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.]],
```

Listing 4.18 displays the contents of predict-sklearn.py that illustrates how to perform a prediction based on a dataset that contains hard-coded values. Although the discussion of decision trees is beyond the scope of this book, Listing 4.18 demonstrates how easy it is to use them "out of the box" in Sklearn.

LISTING 4.18: predict-sklearn.py

```
from sklearn import tree

# step 1: collect training data
features = [[130,1], [140,1], [150,0], [170,0]]
labels   = [0, 0, 1, 1]

# step 2: train the classifier via a decision tree
clf = tree.DecisionTreeClassifier()

# "fit" learning algorithm ("find patterns in data")
clf = clf.fit(features, labels)
```

```
# step 3: make a prediction
print('a fruit of weight 160 =',clf.predict([[160,0]]),
                    'where 0 is an apple and 1 is an orange')
```

Listing 4.18 imports the `tree` class from `Sklearn` and then initializes the 2D Python array features with 2D points that specify the weight and the class for each 2D point. Specifically, the points [130,1] and [140,1] belong to class 0, whereas the points [150,0] and [170,0] belong to class 0. Since the weight value of 160 lies between 150 and 170, you no doubt expect the code to predict the class 0 for this weight. Indeed, class 0 is the predicted value for the class, as you can see in the output (shown in bold) from launching the code in Listing 4.18:

```
a fruit of weight 160 = [1] where 0 is an apple and 1 is an
                                                        orange
```

THE DIGITS DATASET IN SKLEARN (2)

Listing 4.19 displays the contents of `load-digits2.py` that illustrates how to plot the `Digits` dataset.

LISTING 4.19: 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 4.19 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 4.19 initializes the variable `fig` and invokes the method `imshow()` of the `plt` class in order to display a plot of the digits dataset.

Figure 4.17 displays a plot of the data in the `Digits` dataset based on the code in Listing 4.19.

THE DIGITS DATASET IN SKLEARN

Listing 4.20 displays the contents of `sklearn-digits.py` that illustrates how to access the `Digits` dataset in `Sklearn`.

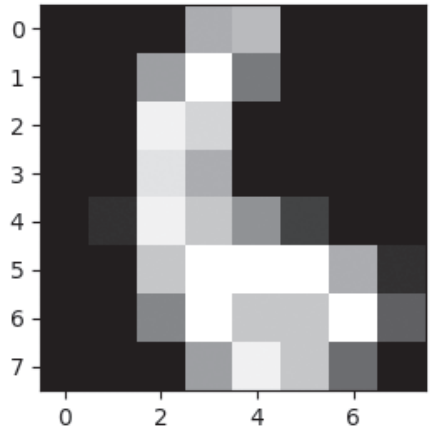

FIGURE 4.17 The Sklearn Digits Dataset.

LISTING 4.20: 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,
interpolation='nearest')
plt.show()
```

Listing 4.20 starts with one `import` statement followed by the variable `digits that` contains the `Digits` dataset. The output from Listing 4.21 is here:

```
digits shape: (1797, 8, 8)
data   shape: (1797, 64)
(samples,features): (1797, 64)
```

Figure 4.18 displays the images in the `Digits` dataset based on the code in Listing 4.20.

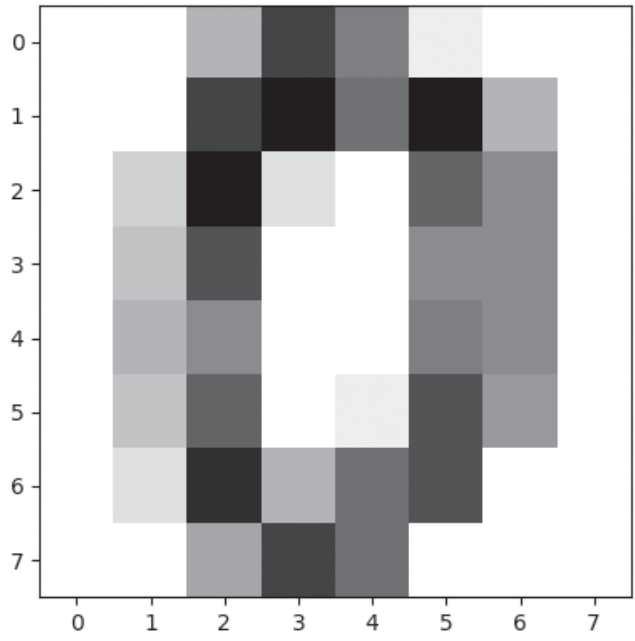

FIGURE 4.18 The Digits Dataset.

DISPLAYING IMAGES IN SKLEARN

Listing 4.21 displays the contents of `load-digits3.py` that illustrates how to display the `Digits` dataset in `Sklearn`.

LISTING 4.21: load-digits3.py

```
from sklearn import datasets
from sklearn.datasets import load_digits
from matplotlib import pyplot as plt

digits = load_digits()

# Figure size (width, height) in inches
fig = plt.figure(figsize=(6, 6))

# Adjust the subplots
fig.subplots_adjust(left=0, right=1, bottom=0, top=1,
                                    hspace=0.05, wspace=0.05)

# For each of the 64 images
for i in range(64):
  # Initialize the subplots: add a subplot in
  # the grid of 8 by 8, at the i+1-th position
  ax = fig.add_subplot(8, 8, i + 1, xticks=[], yticks=[])
```

```
  # Display an image at the i-th position
ax.imshow(digits.images[i],cmap=plt.cm.binary,interpolation
='nearest')

  # label the image with the target value
  ax.text(0, 7, str(digits.target[i]))

# Show the plot
plt.show()
```

Listing 4.21 contains various `import` statements and then initializes the variable `digits` with the contents of the `Digits` dataset. The next portion of Listing 4.21 contains code for initializing the dimensions and attributes for the actual plot, followed by a `for` loop that adds 64 images in an 8x8 rectangular grid. These images are taken from the `digits` variable. For instance, the ith image in the variable digits is `digits.images[i]`. Each image is displayed with its target value.

Launch the code in Listing 4.21 and you will see the plotted image as shown in Figure 4.19.

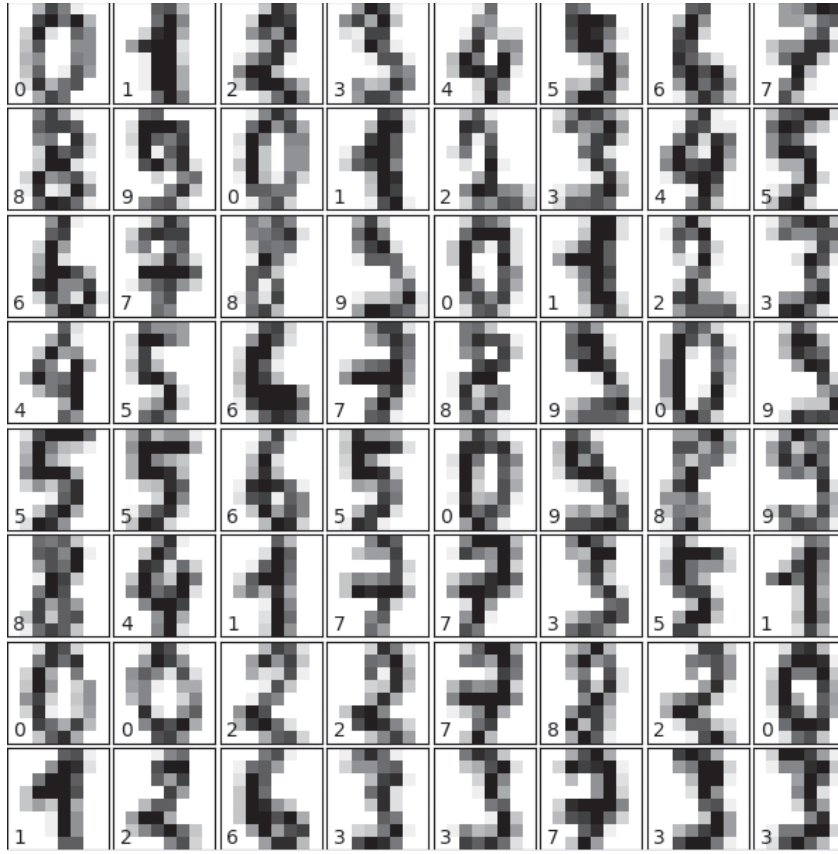

FIGURE 4.19 The Digits Dataset.

THE IRIS DATASET IN SKLEARN (OPTIONAL)

The `Iris` dataset in `Sklearn` consists of the lengths of three different types of `Iris`-based petals and sepals: Setosa, Versicolour, 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 4.22 displays the contents of `sklearn-iris.py` that illustrates how to display detailed information about the `Iris` dataset.

LISTING 4.22: sklearn-iris.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 4.22 starts with an `import` statement followed by the variables `iris` and `data`, where the latter contains the `Iris` dataset. The first half of Listing 4.22 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 4.22 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 4.22 iterates generates a histogram that displays some of the images in the `Iris` dataset. The output from Listing 4.22 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 4.20 displays the images in the `Iris` dataset based on the code in Listing 4.22.

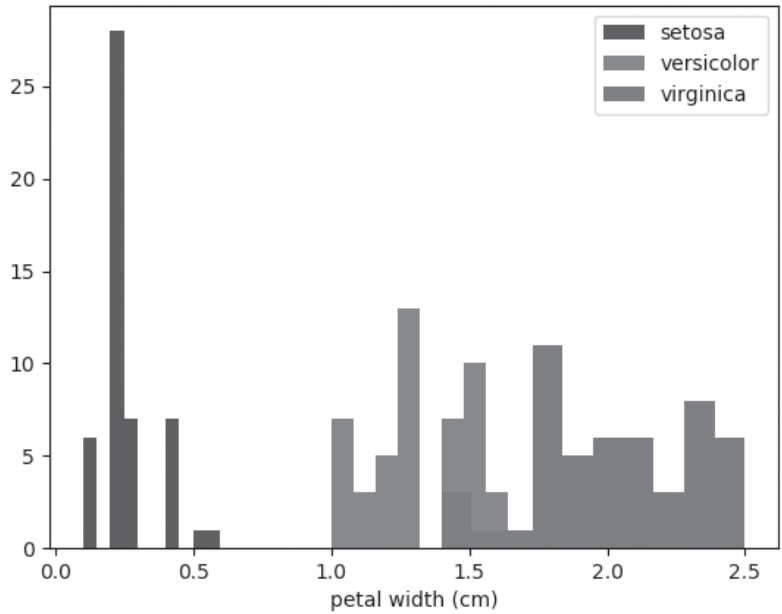

FIGURE 4.20 The Iris Dataset.

THE `OLIVETTI 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 4.23, the `sklearn.datasets.fetch_olivetti_faces` function is the data fetching and caching function that downloads the data archive from AT&T.

Listing 4.23 displays the contents of `sklearn-faces.py` that displays the contents of the `Faces` dataset in `Sklearn`.

LISTING 4.23: 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 4.23 starts with `import` statements and then initializes the variable `faces` with the contents of the `Olivetti Faces` dataset. The next portion of Listing 4.23 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 4.23 and you will see the plotted image as shown in Figure 4.21.

FIGURE 4.21 The faces Dataset.

Now that you have seen some of the `Sklearn` datasets, along with code samples for displaying their contents, let us turn our attention to linear regression in `Sklearn`, as discussed in the next section.

SIMPLE LINEAR REGRESSION IN SKLEARN (1)

Listing 4.24 displays the contents of `sklearn-simple-lr.py` that illustrates how to perform linear regression in `Sklearn` with a set of points.

LISTING 4.24: sklearn-linreg.py

```
import warnings
warnings.filterwarnings(action="ignore", module="scipy",
message="^internal gelsd")

from sklearn import linear_model

reg = linear_model.LinearRegression()
reg.fit ([[0, 0], [1, 1], [2, 2]], [0, 1, 2])
print(reg.coef_)
```

Listing 4.24 starts with two statements (shown in bold) that suppress warning messages involving `gelsd`. You can comment out these two lines to see if a warning message is displayed.

The next portion of Listing 4.24 initializes the variable `reg` as an instance of the `LinearRegression` class, and then invokes the `fit()` method in order to train the model with the hard-coded list of values. The `print()` statement displays the result of the training, which is a pair of numbers that correspond to the slope and y-intercept of the best-fitting line. The output from Listing 4.24 is here:

```
[0.5 0.5]
```

Launch Listing 4.24 and you will see the plotted image as shown in Figure 4.22.

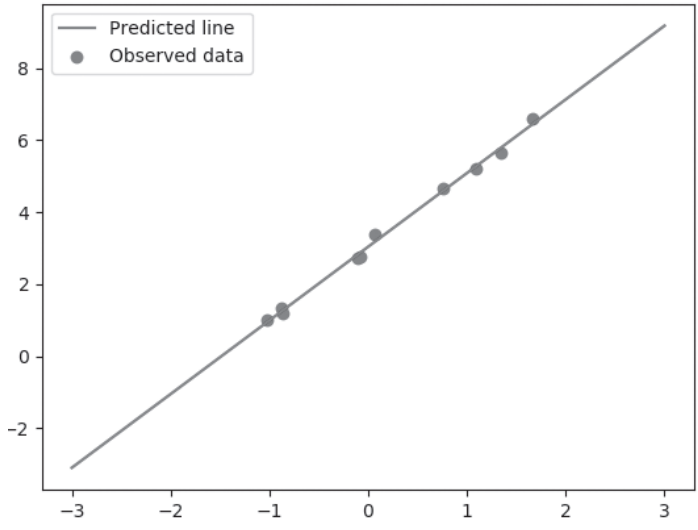

FIGURE 4.22 Simple Linear Regression.

LINEAR REGRESSION IN SKLEARN (2)

Listing 4.25 shows you show you how to perform linear regression in order to find the best-fitting line to a set of points in the plane. Listing 4.24 extends Listing 4.23 by using a linear equation instead of a set of a hard-coded set of points, and then plots the points and the best-fitting line.

Listing 4.25 displays the contents of `sklearn-linreg.py` that illustrates how to perform linear regression in `Sklearn` with a random set of points.

LISTING 4.25: sklearn-linreg.py

```
from sklearn.linear_model import LinearRegression
import Matplotlib.pyplot as plt
import numpy as np

x = np.random.randn(10,1)
y = 2*x + 3 + 0.1*np.random.randn(10,1)

model = LinearRegression()
model.fit(x,y)

print('slope:',model.coef_)
print('inter:',model.intercept_)
print('predict:',model.predict(20))

x_test = np.linspace(-3,3)
y_pred = model.predict(x_test[:,None])

plt.scatter(x,y)
plt.plot(x_test,y_pred,'r')
plt.legend(['Predicted line','Observed data'])
plt.show()
```

Listing 4.25 contains a new `import` statement that enables us to use the `LinearRegression` class in `Sklearn`. Next, the `NumPy` variable `x` is initialized with a random set of numbers, followed by the `NumPy` variable `y` whose values are based on the variable `x`. Notice that each value in `y` is offset with a small randomized a number so that we have a "fuzzy" approximation to a line (you have seen this technique used in other code samples).

Next, the variable `model` is instantiated as an instance of `LinearRegression`, followed by the invocation of the `fit()` method in order to train this model. The `NumPy` variable `x_test` is initialized via the `NumPy linspace()` API and the `NumPy` variable `y_pred` is initialized with the predicted values that are calculated via the `predict()` API.

The last block of code in Listing 4.25 displays a scatter plot of points derived from x and y, followed by a linear plot with a legend.

Figure 4.23 displays the images based on the generated output from Listing 4.25.

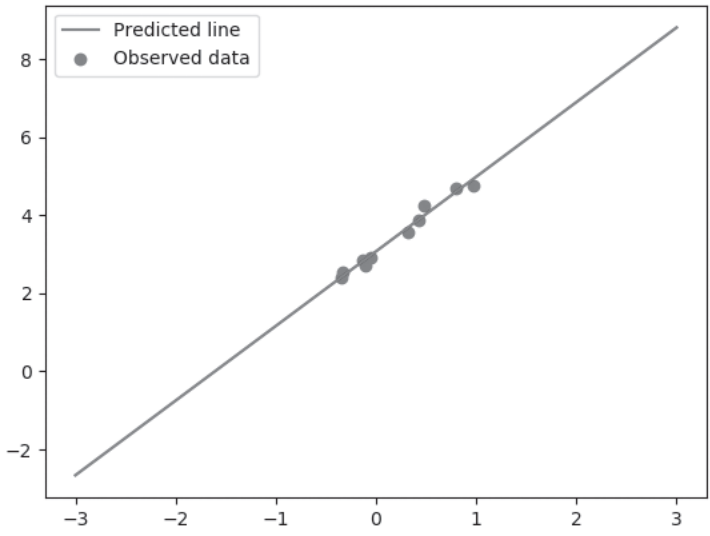

FIGURE 4.23 A Scatter Plot and a Best-Fitting Line.

LINEAR REGRESSION AND REGULARIZATION IN SKLEARN

The Appendix discusses overfitting, which can occur when you train a model, and also discusses regularization techniques that impose a "penalty" on the least squares solution in order to reduce the overfitting in a model. Two well-known techniques are L1 and L2 regularization.

`Sklearn` supports `Ridge` regression (essentially the same as L2 regression) to address the overfitting in a model, which you can easily specify in the appropriate API.

Listing 4.26 displays the contents of `sklearn-ridge-linreg.py` that illustrates how to include `Ridge` regression in a linear model while performing linear regression in `Sklearn` with a random set of points.

LISTING 4.26: sklearn-ridge-linreg.py

```
from sklearn.linear_model import LinearRegression
import warnings
warnings.filterwarnings(action="ignore", module="scipy",
message="^internal gelsd")

from sklearn import linear_model

# previous: reg = linear_model.LinearRegression()
reg = linear_model.Ridge (alpha = .5)
reg.fit ([[0, 0], [0, 0], [1, 1]], [0, .1, 1])

print("Coefficients:",reg.coef_)
print("Intercept:   ",reg.intercept_)
```

Listing 4.26 shows a commented out code snippet from the Python script `sklearn-simple-lr.py`, followed by the new code snippet (shown in bold) that provides the `Ridge`-related functionality. The output from Listing 4.26 is here:

```
Coefficients: [0.34545455 0.34545455]
Intercept:    0.13636363636363638
```

In addition to support for linear regression and L2 regularization, another nice feature of `Sklearn` is its support for logistic regression, which is the topic of a section later in this chapter.

WHAT IS LOGISTIC REGRESSION?

Logistic regression is essentially the result of "applying" the `sigmoid` activation function to linear regression in order to perform binary classification. Logistic regression is useful in a variety of unrelated fields. Such fields include machine learning, various medical fields, and social sciences. Logistic regression can be used to predict the risk of developing a given disease, based on various observed characteristics of the patient. Other fields that use logistic regression include engineering, marketing, and economics.

Logistic regression can be binomial (only two outcomes for a dependent variable), multinomial (three or more outcomes for a dependent variable), or ordinal (dependent variables are ordered).

For instance, suppose that a dataset consists of data that belong either to class A or to class B. If you are given a new data point, logistic regression predicts whether that new data point belongs to class A or to class B. By contrast, linear regression predicts a numeric value, such as the next-day value of a stock.

Despite its name, logistic regression is a classification technique and not a regression technique.

SKLEARN AND LOGISTIC REGRESSION

As you might have already surmised, `Sklearn` supports both linear regression and logistic regression. Listing 4.27 displays the contents of `logistic-regression-iris.py` that illustrates how to use `Sklearn` and logistic regression with the `Iris` dataset.

LISTING 4.27: logistic-regression-iris.py

```
from sklearn import datasets
from sklearn import metrics
from sklearn.linear_model import LogisticRegression

# load the iris datasets
dataset = datasets.load_iris()

# fit a logistic regression model to the data
```

```
model = LogisticRegression()
model.fit(dataset.data, dataset.target)
print(model)

# make predictions
expected = dataset.target
predicted = model.predict(dataset.data)

# summarize the fit of the model
print("classification report:")
print(metrics.classification_report(expected, predicted))
print("confusion matrix:")
print(metrics.confusion_matrix(expected, predicted))
```

Listing 4.27 initializes the variable `dataset` with the contents of the `Iris` dataset using code that is similar to previous code samples. Next, the variable `model` is initialized as an instance of the `LogisticRegression` class that is available in `Sklearn`, and then the `fit()` method of the `model` model is invoked in order to fit the data and the target in the `dataset` variable.

The next portion initializes the variables expected (i.e., the expected predictions) with the actual predictions in the variable predicted, after which we can generate a classification report and also display the confusion matrix.

In case you are wondering, the confusion matrix displays the number of predictions for a given class. For instance, the value in position (1,1) of the confusion matrix displays the number of items that were correctly predicted to be in class 1. However, the value in position (1,2) displays the number of items that were incorrectly predicted to belong to class 1 but in reality they belong to class 2. In general, if A is an nxn confusion matrix, the diagonal values display the number of correct predictions, whereas the number in a non-diagonal entry (i, j) displays the number of items that belong to class i, but whose prediction is that they belong to class j.

With all the preceding points in mind, here is the output from launching the code in Listing 4.27:

```
classification report:
             precision    recall  f1-score   support

          0       1.00      1.00      1.00        50
          1       0.98      0.90      0.94        50
          2       0.91      0.98      0.94        50

avg / total       0.96      0.96      0.96       150
confusion matrix:
[[50  0  0]
 [ 0 45  5]
 [ 0  1 49]]
```

This concludes the portion of the chapter pertaining to `Sklearn`. Now let us 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 the 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 4.28 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 4.28: seaborn-tips.py

```
import seaborn as sns
df = sns.load_dataset("tips")
print(df.head())
```

Listing 4.28 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 4.28 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 4.29 displays the contents of `seaborn-iris.py` that illustrates how to plot the `Iris` dataset.

LISTING 4.29: 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 4.29 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`, the vertical axis labeled `petal_length`, and the displayed points are from the `Iris` dataset.

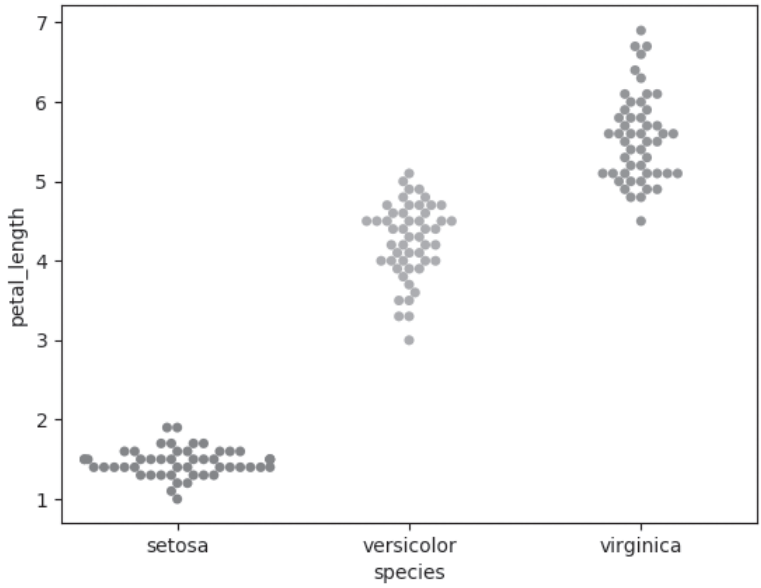

FIGURE 4.24 The Iris Dataset.

Figure 4.24 displays the images in the `Iris` dataset based on the code in Listing 4.29.

THE TITANIC DATASET IN SEABORN

Listing 4.30 displays the contents of `seaborn-titanic-plot.py` that illustrates how to plot the `Titanic` dataset.

LISTING 4.30: 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 4.30 contains the same `import` statements as Listing 4.22, 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 4.25 displays a plot of the data in the `Titanic` dataset based on the code in Listing 4.30.

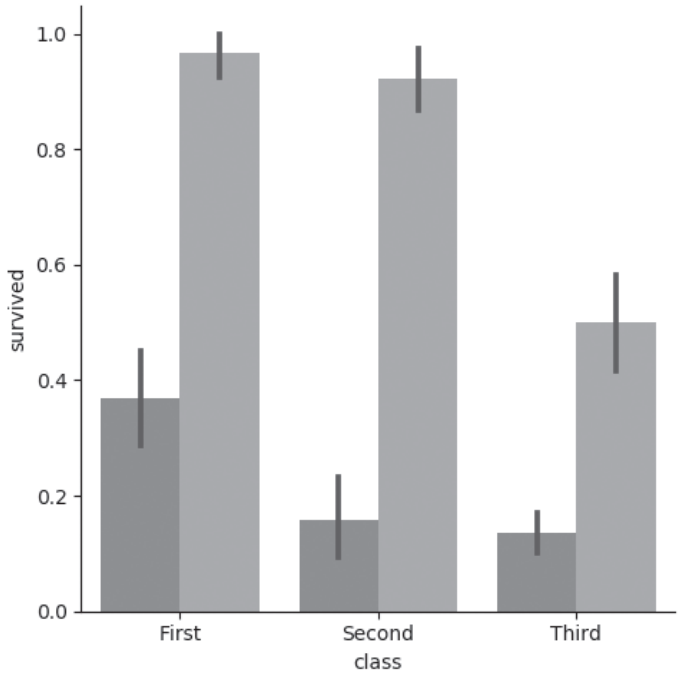

FIGURE 4.25 A Histogram of the Titanic Dataset.

EXTRACTING DATA FROM TITANIC DATASET IN SEABORN

Listing 4.31 displays the contents of `seaborn-titanic.py` that illustrates how to extract subsets of data from the `Titanic` dataset.

LISTING 4.31: 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 4.31 contains the same `import` statements as Listing 4.30, and then initializes the variable `titanic` with the contents of the built-in `Titanic` dataset. The next portion of Listing 4.31 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 4.31 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 TITANIC DATASET IN SEABORN

Listing 4.32 displays the contents of `seaborn-titanic2.py` that illustrates how to extract subsets of data from the `Titanic` dataset.

LISTING 4.32: 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 4.32 contains the same `import` statements as Listing 4.31, 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 displays the age and fare of the passengers with index 4 and index 5 in the dataset.

The final portion of Listing 4.32 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 did not survive the accident. The entire output from Listing 4.32 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 4.33 displays the contents of `pandas-seaborn.py` that illustrates how to display a `Pandas` dataset in `Seaborn`.

LISTING 4.33: 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 4.33 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 4.33 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 4.33 is here:

```
top five elements:
    x   y
0  52  34
1  31  47
2  23  18
3  34  70
4  71   1
```

Figure 4.26 displays a plot of the data in the `Titanic` dataset based on the code in Listing 4.33.

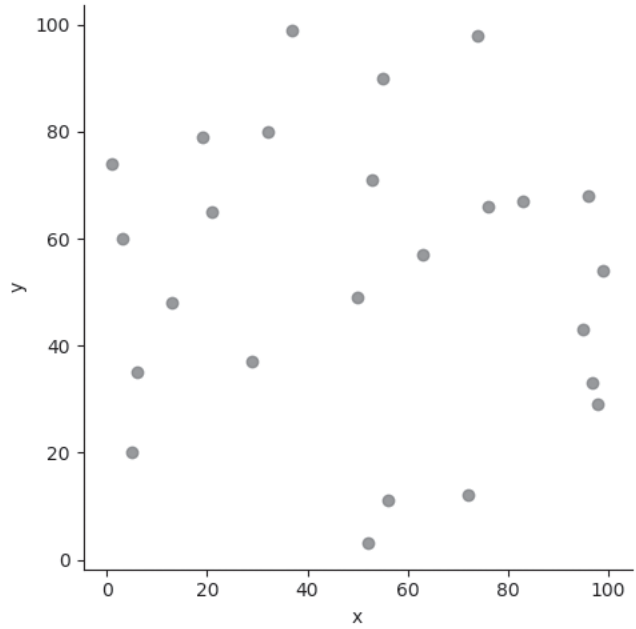

FIGURE 4.26 A Pandas Dataframe Displayed via Seaborn.

SUMMARY

This chapter started with some high-level information about Linear Algebra, which is important for further learning (and beyond the scope of this book). Then you learned about 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 `Iris` and `Digits` datasets, and how to process images. In addition, you saw how to perform Linear Regression with `Sklearn`.

Finally, 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`.

CHAPTER 5

INTRODUCTION TO TENSORFLOW

This chapter provides a quick introduction to various features of TensorFlow, and some of the TensorFlow tools and projects that are included in the TensorFlow "family." The intent of this chapter is to provide relevant TensorFlow information, how to visualize TensorFlow graphs via TensorBoard, how to invoke TensorFlow code in a browser via Jupyter notebooks, and where to train neural networks with free GPU support (Google Colaboratory). The material in this chapter will prepare you for Chapter 2, which provides a "foundation" of commonly used TensorFlow APIs (illustrated via short code samples) that are also in code samples in the remaining chapters of this book.

There are a few points to keep in mind before you read this chapter. First, in this book TensorFlow refers to TensorFlow 1.x, unless explicitly stated otherwise. For expediency, you will often see the acronym TF used instead of TensorFlow. In addition, the historical details regarding TensorFlow are minimized in order to provide you with a decent set of TensorFlow code samples.

Second, the code samples in this book were tested on a Macbook Pro (12.5.3) with Python 3.5.1 and TensorFlow 1.12, which is currently the latest version of TensorFlow 1.x. When TensorFlow 2 is released in 2019, all TensorFlow 1.x releases will become legacy code. However, Google will probably support TensorFlow 1.x for at least another year after the release of TensorFlow 2. At that point there will not be any new code development for TF 1.x (except for security-related updates).

The first part of this chapter briefly discusses some TensorFlow features and some of the tools that are part of the TensorFlow "family." The second section of this chapter shows you how to write TensorFlow code that contains various combinations of TF constants, TF placeholders, and TF variables. After reading these code samples, you will learn about the differences between TF placeholders and TF variables, which tend to confuse people who are new to TensorFlow.

The third section of this chapter shows you how to perform arithmetic operations in TensorFlow, how to use various built-in functions, how to calculate trigonometric values, `for` loops, `while` loops, and how to calculate exponential values.

The fourth section contains TF code samples that perform various operations on arrays, such as creating an identity matrix, a constant matrix, a random uniform matrix, and a truncated normal matrix. This section also shows you how to multiply TensorFlow arrays and how to convert Python arrays to TensorFlow arrays.

The fourth section shows you how to save a TensorFlow graph so that you can view its contents in a browser using the built-in TensorBoard utility that is launched from the command line. The final section introduces you to Google Colaboratory, which is a fully online Jupyter-based environment that offers 12 hours of daily GPU usage for free.

WHAT IS TENSORFLOW?

TensorFlow is an open source framework from Google that was released in November 2015. The TensorFlow framework is for machine learning and deep learning. TensorFlow evolved from Google Brain and is available through an Apache license. In this book the TensorFlow code samples use Python 3.x. TensorFlow also supports a variety of programming languages and hardware platforms. Here is a short list of some TensorFlow features:

Support for Python, Java, C++
Desktop, server, mobile device (TF Lite)
CPU/GPU/TPU support
Linux and Mac OS X support
VM for Windows

Navigate to the TensorFlow home page, where you will find links to many resources for TensorFlow:
https://www.tensorflow.org

Install TensorFlow by issuing the following command from the command line:

```
pip install tensorflow
```

If you want to upgrade TensorFlow to the latest version, issue the following command from the command line:

```
pip install --upgrade tensorflow
```

TensorFlow Architecture (High View)

TensorFlow consists of two main components: 1) a graph protocol buffer, and 2) a runtime to execute (distributed) graph that is analogous to Python code and the Python interpreter. TensorFlow is written in C++ and supports

various graph-based operations involving primitive values and so-called tensors (discussed later).

TensorFlow creates a "computation" graph for numerical computation and data flow, with the notion that "everything is a graph." Graph and data visualization is handled via TensorBoard (discussed later) that is included as part of TensorFlow. As you will see in the code samples in this book, TensorFlow APIs are available in Python and can therefore be embedded in Python scripts.

The default execution mode for TF 1.x is *deferred execution*, whereas TF 2 uses *eager execution* (think "immediate mode"). Although TF 1.4 introduced eager execution, the vast majority of TF 1.x code samples that you will find online use deferred execution. TensorFlow supports arithmetic operations on tensors (i.e., multidimensional arrays with enhancements) as well as conditional logic and while loops.

TensorFlow provides a "checkpoint" API to save TensorFlow graphs and restore them at a later point. In addition, you can view saved graphs in TensorBoard, which is a very useful data visualization tool that is bundled with TensorFlow.

TensorFlow Features

The following list contains various high-level features of TensorFlow:

Distributed computation
R/D for developing new ML algorithms
A REPL environment
Take models from training to production
Large-scale distributed models
TF graph computed on different machines
Models for mobile

Most of the preceding items are self-explanatory. The TensorFlow `REPL` (read-eval-print-loop) is available through the Python `REPL`, which is accessible by opening a command shell and then typing the following command:

```
python
```

As a simple illustration of accessing TensorFlow-related functionality in the Python `REPL`, import the TensorFlow library as follows:

```
>>> import tensorflow as tf
```

Now check the version of TensorFlow that is installed on your machine with this command:

```
>>> print('TF version:',tf.__version__)
```

The output of the preceding code snippet is shown here (the number that you see depends on which version of TensorFlow you installed):

```
TF version: 1.12.0
```

Although the Python REPL is useful for short code blocks, it is simpler to place the TensorFlow code samples in this book inside Python scripts that you can launch with the Python executable.

TensorFlow Use Cases

TensorFlow is designed to solve many use cases, some of which are listed here:

Image recognition
Computer vision
Voice/sound recognition
Time series analysis
Language detection
Language translation
Text-based processing
Handwriting recognition

In case you did not already know, the preceding list also includes common use cases that are suitable for Machine Learning as well as Deep Learning.

OTHER TENSORFLOW-BASED TOOLKITS

TensorFlow has the following associated toolkits:

TensorBoard (included as part of TensorFlow)
TensorFlow Serving (hosting)
TensorFlow Lite (for mobile)
tensorflow.js (for Web pages and NodeJS)

The TensorFlow distribution contains TensorBoard, which is a graph visualization tool that runs in a browser. For example, type the following command that accesses a saved TF graph in the subdirectory /tmp/abc:

```
tensorboard –logdir=/tmp/abc
```

Next, open a browser session and navigate to this URL:

```
localhost:6006
```

Now you will see a visualization of the TensorFlow graph that was saved as a file in the directory /tmp/abc.

TensorFlow Serving is a flexible, high-performance serving system for machine learning models designed for production environments. TensorFlow Serving makes it easy to deploy new algorithms and experiments, while keeping the same server architecture and APIs. More information is here:
https://www.tensorflow.org/serving/

TensorFlow Lite was created for developing mobile applications (both Android and iOS). TensorFlow Lite supersedes TensorFlow Mobile, which

was an earlier SDK for developing mobile applications. TensorFlow Lite is a lightweight solution for mobile and embedded devices. It enables on-device machine learning inference with low latency and a small binary size. TensorFlow Lite also supports hardware acceleration with the Android Neural Networks API. More information about TensorFlow Lite is here:
https://www.tensorflow.org/lite/

A more recent addition is `tensorflow.js` that provides JavaScript APIs to access TensorFlow in a Web page. The `tensorflow.js` toolkit was previously called `deeplearning.js`. In addition, tensorflow.js can be used with NodeJS. More information about `tensorflow.js` is here:
https://js.tensorflow.org/

WHAT ABOUT TENSORFLOW 2?

TensorFlow 2 will probably be released some time in the spring of 2019, at which point TensorFlow 1.x will become legacy code (as mentioned in the introduction). One of the most significant changes involves "eager" execution as the default mode instead of "deferred" execution mode.

Why use TF 1.x Instead of TF 2?

There are several reasons for staying with TF 1.x instead of switching to TF 2. First, as this book goes to print, TF 2 has not been released. Second (and perhaps even more important), currently there is a very small percentage of online TensorFlow code samples that use eager execution. Fortunately, Google will provide tools for converting TF 1.x code to TF 2 that will assist in many situations, i.e., when there is a straightforward conversion from TF 1.x APIs to TF 2 APIs. When those tools cannot perform the conversion, there will probably be online tutorials (perhaps from Google?) that will explain how to make the conversion.

Third, there is a massive code base that uses TF 1.x, which includes many customers and even Google itself. The conversion process from TF 1.x to TF 2 will take a lot of time because some (probably large) companies will be slow to make the transition. As an analogy, consider the migration from Python 2.x to Python 3.x: there are still many places that continue using Python 2.x, even though Python 3.x was released several years ago.

There is one possible exception: if you are "brand new" to TensorFlow and you do not have an urgent need to learn TensorFlow, then it might be less work for you to wait until TF 2 is officially released.

WHAT IS A TENSORFLOW TENSOR?

In simplified terms, a TF tensor is an n-dimensional array that is similar to a NumPy `ndarray`. A TF tensor is defined by its dimensionality, as summarized here:

```
scalar number:        a zeroth-order tensor
vector:               a first-order tensor
```

```
matrix:              a second-order tensor
3-dimensional array: a 3rd order tensor
```

As you will see in subsequent sections, TensorFlow supports various data types and primitive types. In addition to constants and variables, TensorFlow provides a data type called a "placeholder," which acts as a buffer that holds transient data. Use TF variables for things that need to be trained, such as the slope `m` and intercept `b` of a best-fitting line in the plane.

TensorFlow Data Types

TensorFlow supports the following data types:

- tf.float32
- tf.float64
- tf.int8
- tf.int16
- tf.int32
- tf.int64
- tf.uint8
- tf.string
- tf.bool

The data types in the preceding list are self-explanatory: two floating point types, four integer types, one unsigned integer type, one string type, and one Boolean type. As you can see, there is a 32-bit and a 64-bit floating point type, and integer types that range from 8-bit through 64-bit.

TensorFlow Primitive Types

TensorFlow supports the following primitive types (arrays are discussed later):

TF Constants
TF Placeholders
TF Variables

A TensorFlow *constant* is an immutable value, and a simple example is shown here:

```
aconst = tf.constant(3.0)
```

A TensorFlow *placeholder* allocates space for data, and it's somewhat analogous to a "buffer" in other languages such as C. The really nice aspect of TF placeholders is that you do *not* need to specify their shape: they "automagically" assume the shape of the data that is assigned to them. A simple example of a TensorFlow placeholder is shown here:

```
a = tf.placeholder("float")
b = tf.placeholder("float")
c = tf.multiply(a,b)
```

As you can see in the preceding code block, `c` is defined as the product of `a` and `b`, which do not have a value. Use a `feed_dict` to assign values to placeholder, as shown here:

```
a = tf.placeholder("float")
b = tf.placeholder("float")
c = tf.multiply(a,b)

# assign values to a and b:
feed_dict = {a:2, b:3}
```

The calculation of `c` itself is shown later in the section that discusses TF sessions.

A TensorFlow *variable* is a "trainable value" in a TensorFlow graph. For example, the slope m and y-intercept b of a best-fitting line in the plane are two examples of trainable values. Some examples of TF variables are shown here:

```
b = tf.Variable(3, name="b")
x = tf.Variable(2, name="x")
z = tf.Variable(5*x, name="z")

W = tf.Variable(20)
lm = tf.Variable(W*x + b, name="lm")
```

Notice that `b`, `x`, and `W` are assigned with numeric initial values, whereas `z` and `W` are defined as expressions. Specifically, the value of the variable `z` depends on the value of `x` (which equals 2), and the value of the variable `lm` depends on the values of `W`, `x`, and `b`. Both `z` and `lm` are evaluated inside a TensorFlow "session" block, as shown in a later section.

TENSORFLOW GRAPHS

Whenever you define TensorFlow code in a Python script, you will often use some combination of TF data types and other TF constructs. Before executing TensorFlow code, TensorFlow generates a graph structure that is based on the constants, placeholders, and variables that you have defined. A typical TensorFlow graph consists of the following:

Graph: graph of operations (DAG)
Sessions: contains Graph(s)
lazy execution (default)
operations in parallel (default)
Nodes: operators/variables/constants
Edges: tensors

TensorFlow graphs are split into subgraphs and executed in parallel (or multiple CPUs)

The chapters in this book contain an assortment of TensorFlow code samples written in Python illustrate how to perform arithmetic operations, calculate trigonometric values, and how to use eager execution.

TF go faster: *https://www.youtube.com/watch?v=BoufOxOjyj8&feature=youtu.be*

The TensorFlow Version Number

Listing 5.1 displays the contents of `tf-version.py` that illustrates how to find the version number of TensorFlow that is installed on your machine.

LISTING 5.1: tf-version.py

```
import tensorflow as tf

print('TF version:',tf.__version__)
```

Listing 5.1 contains an `import` statement and one `print()` statement that displays the installed version of TensorFlow. The output from Listing 5.1 is here:

```
('TF version:', '1.12.0')
```

Listing 5.1 consists of two statements: an `import` statement for TensorFlow, followed by a `print()` statement that displays the version of TensorFlow that is installed on your machine. The output from Listing 5.1 is here (the number depends on the version of TensorFlow that you installed on your machine):

```
TF version: 1.12.0
```

A TENSORFLOW GRAPH WITH `TF.SESSION()`

A Tensorflow graph in TF 1.x consists of two parts: TensorFlow code in the first portion of a Python script, followed by an instance of a `tf.Session()` object (often abbreviated as `sess`). In order to obtain values from TF tensors, you invoke the `sess.run()` method and specify a TF tensor as an argument to the `sess.run()` method. By contrast, TF 2 does not require an instance of `tf.Session()`, and it's more "Pythonesque" than TF 1.x.

In extremely simple cases of TF 1.x code, such as displaying the version number of TensorFlow, you do not need a `tf.Session()` object. The next section contains a very simple program that consists of a mere two lines of code to display the version of TensorFlow that is installed on your machine.

Listing 5.2 displays the contents of `tf-session.py` that illustrates how to use the `tf.Session()` object in a TF graph.

LISTING 5.2: tf-session.py

```
import tensorflow as tf

aconst = tf.constant(3.0)
print(aconst)
# output: Tensor("Const:0", shape=(), dtype=float32)

sess = tf.Session()
print(sess.run(aconst))
# output: 3.0

sess.close()
# => there's a better way…
```

Listing 5.2 starts with an `import` statement, the definition of the TF constant `aconst`, and one `print()` statement that displays the metadata for `aconst`. The next portion of Listing 5.2 initializes the variable `sess` as an instance of the `TF.Session` class, followed by a `print()` statement that display the value of the variable `aconst`. The output from Listing 5.2 is here:

```
3.0
```

Listing 5.3 displays the contents of `tf-const2.py` that illustrates how to use the `tf.Session()` object in a TF graph.

LISTING 5.3: tf-session2.py

```
import tensorflow as tf

aconst = tf.constant(3.0)
print(aconst)

# Automatically close "sess"
with tf.Session() as sess:
  print(sess.run(aconst))
```

Listing 5.3 contains almost the same code as Listing 5.2; the difference is that `with` code snippet, which does not require explicitly closing the TF session. The output from Listing 5.3 is here:

```
3.0
```

PLACEHOLDERS AND `FEED_DICT` IN A TF SESSION

Listing 5.4 displays the contents of `tf-ph-feeddict.py` that illustrates how to compute values involving TF placeholders in with code block in TensorFlow.

LISTING 5.4: tf-ph-feeddict.py

```
import tensorflow as tf

a = tf.placeholder("float")
b = tf.placeholder("float")
c = tf.multiply(a,b)

# initialize a and b:
feed_dict = {a:2, b:3}

# multiply a and b:
with tf.Session() as sess:
  print(sess.run(c, feed_dict))
```

Listing 5.4 defines two TF placeholders `a` and `b`, followed by the TF variable `c` that is the product of `a` and `b`. The output of Listing 5.4 is 6, which is the product of the values assigned to the TF placeholders `a` and `b`.

There is another important detail about TF variables and how they are initialized in TensorFlow, which is the subject of the next section.

CONSTANTS AND VARIABLES IN A TF SESSION

Listing 5.5 displays the contents of `tf-variables-init.py` that illustrates how to compute values involving TF placeholders in a `with` code block in TensorFlow.

LISTING 5.5: tf-variables-init.py

```
import tensorflow as tf

x = tf.constant(5,name="x")
y = tf.constant(8,name="y")
z = tf.Variable(2*x+3*y, name="z")

init = tf.global_variables_initializer()

with tf.Session() as session:
  session.run(init)
  print 'z = ',session.run(z) # =>  z = 34
```

The two code snippets shown in bold in Listing 5.5 are required (you can replace `init` with a different string) whenever a `with` code block contains a TF variable, such as the variable `z` in Listing 5.5. The output of Listing 5.5 is 34, which is the result of computing `2*x + 3*y`.

This concludes the quick tour involving TensorFlow code that contains various combinations of TF constants, TF placeholders, and TF variables. The next few sections delve into more details regarding the TF primitive types that you saw in the preceding sections.

CONSTANTS IN TENSORFLOW (REVISITED)

Here is a short list of some properties of TensorFlow constants:

- initialized during their definition
- cannot change its value ("immutable")
- can specify its name (optional)
- the type is required (ex: tf.float32)
- assigned value only once (can use `feed_dict`)
- are not modified during training

Some examples of TensorFlow constants:

```
d = tf.constant("3", name="a")
e = tf.constant([5,5,5], tf.float32)
```

Listing 5.6 displays the contents of `tf-constants.py` that illustrates how to assign and print the values of some TensorFlow constants.

LISTING 5.6: tf-constants.py

```
import tensorflow as tf

scalar = tf.constant(10)
vector = tf.constant([1,2,3,4,5])
matrix = tf.constant([[1,2,3],[4,5,6]])
cube   = tf.constant([[[1],[2],[3]],[[4],[5],
                                     [6]],[[7],[8],[9]]])

print(scalar.get_shape())
print(vector.get_shape())
print(matrix.get_shape())
print(cube.get_shape())
```

Listing 5.6 contains four `tf.constant()` statements that define TF tensors of dimension 0, 1, 2, and 3, respectively. The second part of Listing 5.6 contains four `print()` statements that display the shape of the four TF constants that are defined in the first section of Listing 5.6. The output from Listing 5.6 is here:

```
()
(5,)
(2, 3)
(3, 3, 1)
```

The tf.rank() API

The *rank* of a TensorFlow tensor is the dimensionality of the tensor, whereas the *shape* (discussed in the next section) of a tensor is the number of

elements in each dimension. Listing 5.7 displays the contents of `tf-rank.py` that illustrates how to find the rank of TensorFlow tensors.

LISTING 5.7: tf-rank.py

```
import tensorflow as tf

# constant:
aconst = tf.constant(3.0)
print(aconst)

# 2x3 constant matrix
B = tf.fill([2,3], 5.0)

with tf.Session() as sess:
  print('aconst:',sess.run(tf.rank(aconst)))
  print('rank B:',sess.run(tf.rank(B)))
```

Listing 5.7 contains familiar code for defining the TF constant `aconst`, followed by the TF tensor `B` that is a 2x3 tensor in which every element has the value 5. The next block of code is a with code block that prints the values of `aconst` and the shape of `B`. The output from Listing 5.7 is here:

```
Tensor("Const:0", shape=(), dtype=float32)
aconst: 3.0
rank B: 2
```

The Shape of a TF Tensor

The *shape* of a TensorFlow tensor is the number of elements in each dimension of a given tensor.

Listing 5.8 displays the contents of `tf-shape.py` that illustrates how to find the shape of TensorFlow tensors.

LISTING 5.8: tf-shape.py

```
import tensorflow as tf

# constant:
aconst = tf.constant(3.0)
print(aconst)

# 2x3 constant matrix
B = tf.fill([2,3], 5.0)

with tf.Session() as sess:
  print('aconst:',sess.run(tf.rank(aconst)))
  print('rank B:',sess.run(tf.rank(B)))
```

Listing 5.8 contains the usual `import` statement, followed by the definition of the TF constant `aconst`. Next, the TF variable B is initialized as a 1x2

vector that is initialized with the value 5.0. Next, a `with` code block contains the code to display the values of `aconst` and the rank of `B`. The output from Listing 5.8 is here:

```
Tensor("Const:0", shape=(), dtype=float32)
aconst: 3.0
rank B: 2
```

PLACEHOLDERS IN TENSORFLOW (REVISITED)

TensorFlow placeholders have the following features:

- they only allocate memory for future use
- they can have variable size
- used for inputting data "external" to graph
- good for unknown data size
- placeholders are from functions (not TF class instances)

The following examples of TensorFlow placeholders illustrate how to specify unconstrained shapes and higher-dimensional tensors:

```
a = tf.placeholder("float")
b = tf.placeholder(tf.int32, name='b')
c = tf.placeholder(tf.float32, shape=[3])

# Unconstrained shape:
w = tf.placeholder(tf.float32)

# Matrix of unconstrained size:
x = tf.placeholder(tf.float32, shape=[None, None])

# Matrix with 32 columns:
y = tf.placeholder(tf.float32, shape=[None, 32])

# 128x32-element matrix:
z = tf.placeholder(tf.float32, shape=[128, 32])
```

The following examples of TensorFlow placeholders illustrate how to specify constrained shapes and higher-dimensional tensors:

```
# placeholder with shape:
x = tf.placeholder(tf.float32,(3,4))

# feed any matrix with 4 columns and any number of rows at
                                                    run time:
x = tf.placeholder(tf.float32, shape=(None,4))

# placeholder X with unspecified number of
# rows of shape (128, 128, 3) of type float32:
X = tf.placeholder(tf.float32, shape=[None, 128, 128, 3],
                                                    name="X")
```

TF PLACEHOLDERS AND FEED_DICT

Listing 5.9 displays the contents of `tf-feeddict-values.py` that illustrates how to specify values in a `feed_dict` in a TF graph.

LISTING 5.9: tf-feeddict-values.py

```
import tensorflow as tf

x = tf.placeholder("float", None)
y = x * 2

with tf.Session() as session:
  result = session.run(y, feed_dict={x: [1, 2, 3]})
  print('y:',result)
```

Listing 5.9 defines the TF placeholder `x` as a float data type, followed by the variable `y` that is twice the value of `x`. The `with` code block calculates the value of `y` by specifying an array of values for `feed_dict`. If you expected merely a scalar value for `x`, this example illustrates the fact that a TF placeholder can take whatever shape is required (i.e., not just a scalar value). The output from Listing 5.9 is here:

```
('y:', array([2., 4., 6.], dtype=float32))
```

Listing 5.10 displays the contents of `tf-feeddict-values2.py` that illustrates how to use `feed_dict` to supply values to a placeholder.

LISTING 5.10: tf-feeddict-values2.py

```
import tensorflow as tf

x = tf.placeholder("float", [None, 3])
y = x * 2
z = x ** 3

with tf.Session() as session:
  x_data = [[1, 2, 3],
            [4, 5, 6],]

  result1 = session.run(y, feed_dict={x: x_data})
  print('y:',result1.eval())

  result2 = session.run(z, feed_dict={x: x_data})
  print('z:',result2.eval())
```

Listing 5.10 defines the TF placeholder `x` as a `float` data type with an arbitrary number of rows and three columns. The variables `y` and `z` are defined as twice the value of `x` and `x` cubed, respectively.

The `with` code block defines the 2x3 array variable `x_data` with integer values, after which `result1` is calculated by evaluating y with `x_data` for

feed_dict. Similarly, result2 is calculated by evaluating z with x_data for feed_dict. The output from Listing 5.10 is here:

```
y: [[ 2.  4.  6.]
 [ 8. 10. 12.]]
z: [[  1.   8.  27.]
 [ 64. 125. 216.]]
```

VARIABLES IN TENSORFLOW (REVISITED)

In addition to constants and placeholders that are described earlier in this chapter, TensorFlow supports variables. Variables can be updated during backward error propagation (also called "backprop" that is discussed later). TF variables can also be saved in a graph and then restored at a later point in time.

TensorFlow also provides the method tf.assign() in order to modify values of TF variables. The following list contains some properties of TensorFlow variables:

- initial value is optional
- must be initialized before graph execution
- updated during training
- constantly recomputed
- they hold values for weights and biases
- in-memory buffer (saved/restored from disk)

Here are some examples of TensorFlow variables:

```
b = tf.Variable(3, name='b')
x = tf.Variable(2, name='x')
z = tf.Variable(5*x, name="z")

W = tf.Variable(20)
lm = tf.Variable(W*x + b, name="lm")
```

Notice that the variables b, x, and W specify constant values, whereas the variables z and lm specify expressions that are defined in terms of other variables. If you are familiar with linear regression, you undoubtedly noticed that the variable lm defines a line in the Euclidean plane.

Other properties of TensorFlow variables are listed below:

- a tensor that is updateable via operations
- exist outside the context of session.run
- like a "regular" variable
- hold the learned model parameters
- variables can be shared (or non-trainable)
- used for storing/maintaining state
- internally stores a persistent tensor

- modifications are visible across multiple `tf.Sessions`
- you can read/modify the values of the tensor
- multiple workers see the same values for `tf.Variables`
- the best way to represent shared, persistent state manipulated by your program

TF Variables versus TF Tensors

Keep in mind the following distinction between TF variables and TF tensors:

TF *variables* represent your model's trainable parameters (ex: weights and biases of a neural network), whereas TF *tensors* represents the data fed into your model and the intermediate representations of that data as it passes through your model.

Initializing Variables in TensorFlow Graphs

Earlier in this chapter you learned that variables are initialized in a TensorFlow session. Doing so will initialize variables with their computed values, such as the `lm` variable that is defined in terms of the variables `W` and `x`.

The TensorFlow method `tf.global_variables_initializer()` explicitly causes the initialization of all variables in your code, and it works in asynchronous mode.

TensorFlow Graph Execution

In all but trivial cases, a TF graph involves the following sequence of steps:

- Build (define) a TF graph
- Initialize a tf.Session()
- Feed data into the graph
- Execute the graph
- Get some output

In subsequent chapters you will see TensorFlow code samples that illustrate each of the steps in the preceding list.

ARITHMETIC OPERATIONS IN TF GRAPHS

Listing 5.11 displays the contents of `tf-arithmetic.py` that illustrates how to perform arithmetic operations in a TF graph.

LISTING 5.11: tf-arithmetic.py

```
import tensorflow as tf

a = tf.add(4, 2)
b = tf.subtract(8, 6)
```

```
c = tf.multiply(a, 3)
d = tf.div(a, 6)

with tf.Session() as sess:
  print(sess.run(a)) # 6
  print(sess.run(b)) # 2
  print(sess.run(c)) # 18
  print(sess.run(d)) # 1
```

Listing 5.11 contains straightforward code for computing the sum, difference, product, and quotient via the `tf.add()`, `tf.subtract()`, `tf.multiply()`, and the `tf.div()` APIs, respectively. The with code block displays the result of invoking those APIs, and the output from Listing 5.11 is here:

```
6
2
18
1
```

TF GRAPHS AND BUILT-IN FUNCTIONS

Listing 5.12 displays the contents of `tf-math-ops.py` that illustrates how to perform arithmetic operations in a TF graph.

LISTING 5.12: tf-math-ops.py

```
import tensorflow as tf

PI = 3.141592
sess = tf.Session()

print(sess.run(tf.div(12,8)))
print(sess.run(tf.floordiv(20.0,8.0)))
print(sess.run(tf.sin(PI)))
print(sess.run(tf.cos(PI)))
print(sess.run(tf.div(tf.sin(PI/4.), tf.cos(PI/4.))))
```

Listing 5.12 contains a hard-coded approximation for `PI`, an instance of `tf.Session()`, and four `print()` statements that display various arithmetic results. Note in particular the third output value is a very small number (the correct value is zero). The output from Listing 5.12 is here:

```
1
2.0
6.27833e-07
-1.0
1.0
```

Listing 5.13 displays the contents of `tf-math-ops-pi.py` that illustrates how to perform arithmetic operations in a TF graph.

LISTING 5.13: tf-math-ops-pi.py

```
import tensorflow as tf
import math as m

PI = tf.constant(m.pi)

sess = tf.Session()

print(sess.run(tf.div(12,8)))
print(sess.run(tf.floordiv(20.0,8.0)))
print(sess.run(tf.sin(PI)))
print(sess.run(tf.cos(PI)))
print(sess.run(tf.div(tf.sin(PI/4.), tf.cos(PI/4.))))
```

Listing 5.13 contains the `tf.floordiv()`, `tf.sin()`, and `tf.cos()` APIs that calculate the floor of the quotient, the sine of a number (in radians), and the cosine of a number (in radians). This time the approximated value is one decimal place closer to the correct value of zero. The output from Listing 5.13 is here:

```
1
2.0
#OLD: 6.27833e-07
-8.742278e-08
-1.0
1.0
```

CALCULATING TRIGONOMETRIC VALUES IN TF

Listing 5.14 displays the contents of `tf-trig-values.py` that illustrates how to perform arithmetic operations in a TF graph.

LISTING 5.14: tf-trig-values.py

```
import tensorflow as tf

import numpy as np
import math as m
PI = tf.constant(m.pi)

a = tf.cos(PI/3.)
b = tf.sin(PI/3.)
c = 1.0/a # sec(60)
d = 1.0/tf.tan(PI/3.) # cot(60)

with tf.Session() as sess:
  print('a:', sess.run(a))
  print('b:', sess.run(b))
  print('c:', sess.run(c))
  print('d:', sess.run(d))
```

Listing 5.14 is straightforward; there are several of the same TF APIs that you saw in Listing 5.13. In addition, Listing 5.14 contains the `tf.tan()` API, which computes the tangent of a number (in radians). The output from Listing 5.14 is here:

```
a: 0.49999997
b: 0.86602545
c: 2.0000002
d: 0.57735026
```

CALCULATING EXPONENTIAL VALUES IN TF

Listing 5.15 displays the contents of `tf-exp-values.py` that illustrates how to perform exponential operations in a TF graph.

LISTING 5.15: tf-exp-values.py

```
import tensorflow as tf

a  = tf.exp(1.0)
b  = tf.exp(-2.0)
s1 = tf.sigmoid(2.0)
s2 = 1.0/(1.0 + b)
t2 = tf.tanh(2.0)

with tf.Session() as sess:
  print('a: ', sess.run(a))
  print('b: ', sess.run(b))
  print('s1:', sess.run(s1))
  print('s2:', sess.run(s2))
  print('t2:', sess.run(t2))
```

Listing 5.15 starts with the TF APIs `tf.exp()`, `tf.sigmoid()`, and `tf.tanh()` that compute the exponential value of a number, the sigmoid value of a number, and the hyperbolic tangent of a number, respectively. The second portion of Listing 5.15 is a `with` code block that displays the values of the initialized values. The output from Listing 5.15 is here:

```
a:  2.7182817
b:  0.13533528
s1: 0.880797
s2: 0.880797
t2: 0.9640276
```

The next section contains a simple example of using a `for` loop in TF, followed by another example that uses a `for` loop in TF.

USING FOR LOOPS IN TF

Listing 5.16 displays the contents of `tf-forloop1.py` that illustrates how to use a `for` loop in a TF graph.

LISTING 5.16: tf-forloop1.py

```
import tensorflow as tf

x = tf.Variable(0, name='x')

init = tf.global_variables_initializer()

with tf.Session() as session:
  session.run(init)
  for i in range(5):
    x = x + 1
    print(session.run(x))
```

Listing 5.16 initializes the TF variable `x` with the value 0 and then defines the `model` variable that is used for initializing the variables in this code sample (which is just the variable `x`). The `with` code block contains a loop that iterates through the values 1 through 5 inclusive. During each iteration of the loop, the variable `x` is incremented by 1 and its value is printed. The output from Listing 5.16 is here:

```
1
2
3
4
5
```

The next section contains an example of a `for` loop in TensorFlow using eager execution.

USING FOR LOOPS WITH EAGER EXECUTION IN TF

Listing 5.17 displays the contents of `tf-forloop2.py` that illustrates how to use a `for` loop in a TF graph.

LISTING 5.17: tf-forloop2.py

```
import tensorflow as tf

import tensorflow.contrib.eager as tfe
tfe.enable_eager_execution()

#x = tf.Variable(0, name='x')
x = tf.contrib.eager.Variable(0, name='x')

for i in range(5):
    print(x)
    x = x + 1
```

Listing 5.17 contains code that is almost the same as Listing 5.16. The new code is shown in bold, starting with an `import` statement that enables us to enable eager execution (via the second statement in bold).

In addition, the definition of the TF variable is modified slightly to reference the `Variable` class in the `tf.contrib.eager` package instead of the `tf` package. The output from Listing 5.17 is here:

```
<tf.Variable 'x:0' shape=() dtype=int32, numpy=0>
tf.Tensor(1, shape=(), dtype=int32)
tf.Tensor(2, shape=(), dtype=int32)
tf.Tensor(3, shape=(), dtype=int32)
tf.Tensor(4, shape=(), dtype=int32)
```

USING WHILE LOOPS WITH EAGER EXECUTION IN TF

Listing 5.18 displays the contents of `tf-while-eager2.py` that illustrates how to use a `for` loop in a TF graph.

LISTING 5.18: tf-while-eager2.py

```
import tensorflow as tf
import tensorflow.contrib.eager as tfe

tfe.enable_eager_execution()

a = tf.constant(12)

while not tf.equal(a, 1):
  if tf.equal(a % 2, 0):
    a = a / 2
  else:
    a = 3 * a + 1
  print(a)
```

Listing 5.18 contains the required pair of statements to invoke eager execution, followed by the TF constant `a` whose value is 12. The next portion of Listing 5.18 is a `while` loop that contains an if/else statement. If the value of `a` is even, then `a` is replaced by half its value. If `a` is odd, then its value is tripled and incremented by 1.

Did you notice that eager execution does not require a `with` code block in Listing 5.18? In addition, `sess.run()` statements are not required. Compared to the code samples that you have seen for deferred execution, eager execution involves a simpler syntax. The output from Listing 5.18 is here:

```
tf.Tensor(6.0, shape=(), dtype=float64)
tf.Tensor(3.0, shape=(), dtype=float64)
tf.Tensor(10.0,shape=(), dtype=float64)
tf.Tensor(5.0, shape=(), dtype=float64)
tf.Tensor(16.0,shape=(), dtype=float64)
tf.Tensor(8.0, shape=(), dtype=float64)
tf.Tensor(4.0, shape=(), dtype=float64)
tf.Tensor(2.0, shape=(), dtype=float64)
tf.Tensor(1.0, shape=(), dtype=float64)
```

As you can see, Listing 5.18 works correctly in eager mode because `a` is defined as a TF constant instead of a variable, so the issues in the previous code sample that contains a loop does not occur in this code sample.

THE TF.LESS() API IN A WHILE LOOP

Listing 5.19 displays the contents of `tf-while-less.py` that illustrates how to use a `while` loop in a TF graph.

LISTING 5.19: tf-while-less.py

```
import tensorflow as tf

x = tf.Variable(0., name='x')
threshold = tf.constant(5.)

model = tf.global_variables_initializer()

with tf.Session() as session:
  session.run(model)
  while session.run(tf.less(x, threshold)):
    x = x + 1
    x_value = session.run(x)
    print(x_value)
```

Listing 5.19 defines the TF variable `x` with the value 0, followed by the TF constant `threshold` whose value is 5. The variable `model` is defined as you have seen previously, followed by the `with` code block. Notice that the `while` loop inside the `with` code block has a more complex condition than the `with` code block that you saw previously in this chapter. The `while` loop increments `x`, assigns the result to `x_value`, and then prints the value of `x_value` as long as the value of `x` is less than the value of threshold. The output from Listing 5.19 is here:

```
1.0
2.0
3.0
4.0
5.0
```

As you can see, Listing 5.19 works correctly in eager mode.

THE TF ONE_HOT() API

Listing 5.20 displays the contents of `tf-onehot2.py` that illustrates how to use the TF `one_hot()` API with an array. Chapter 2 contains more information about "one hot" encoding, which you will encounter when you train neural networks.

LISTING 5.20: tf-onehot2.py

```
import tensorflow as tf
```

```
# Generate one-hot array using idx
idx = tf.get_variable("idx", initializer=tf.constant([2, 0,
                                                      -1, 0]))

target = tf.one_hot(idx, 3, 2, 0)

init = tf.global_variables_initializer()

with tf.Session() as sess:
  sess.run(init)
  print(sess.run(target))
```

Listing 5.20 starts by defining the variable `idx` that is initialized as a TF constant that is a one-dimensional TF tensor with four integer values. Notice that the second and third elements in the TF tensor are equal, which means that their one-hot encoding will be the same. The next portion of Listing 5.20 defines `target`, which will contain the one-hot encoded values for `idx`. Next, the `with` code block initializes the TF variable `idx` and then prints the contents of `target`, which is a 4x3 tensor. The output from Listing 5.20 is here:

```
[[0 0 2]
 [2 0 0]
 [0 0 0]
 [2 0 0]]
```

ARRAYS IN TENSORFLOW (1)

Listing 5.21 displays the contents of `tf-elem1.py` that illustrates how to define a TF array and access elements in that array.

LISTING 5.21: tf-elem1.py

```
import tensorflow as tf

sess = tf.Session()

arr1 = tf.constant([1,2])
print('arr1: ',sess.run(arr1))
print('[0]:  ',sess.run(arr1)[0])
print('[1]:  ',sess.run(arr1)[1])
```

Listing 5.21 contains the TF constant `arr1` that is initialized with the value [1,2]. The three `print()` statements display the value of `arr1`, the value of the element whose index is 0, and the value of the element whose index is 1. The output from Listing 5.21 is here:

```
arr1:  [1 2]
[0]:   1
[1]:   2
```

ARRAYS IN TENSORFLOW (2)

Listing 5.22 displays the contents of `tf-elem2.py` that illustrates how to define a TF array and access elements in that array.

LISTING 5.22: tf-elem2.py

```
import tensorflow as tf

sess = tf.Session()

arr2 = tf.constant([[1,2],[2,3]])
print('arr2:  ',sess.run(arr2))
print('[1]:   ',sess.run(arr2)[1])
print('[1,1]: ',sess.run(arr2)[1,1])
```

Listing 5.22 contains the TF constant `arr1` that is initialized with the value `[[1,2],[2,3]]`. The three `print()` statements display the value of `arr1`, the value of the element whose index is 1, and the value of the element whose index is `[1,1]`. The output from Listing 5.22 is here:

```
arr2:   [[1 2]
 [2 3]]
[1]:    [2 3]
[1,1]:  3
```

ARRAYS IN TENSORFLOW (3)

Listing 5.23 displays the contents of `tf-elem3.py` that illustrates how to define a TF array and access elements in that array.

LISTING 5.23: tf-elem3.py

```
import tensorflow as tf

sess = tf.Session()

arr3 = tf.constant([[[1,2],[2,3]],[[3,4],[5,6]]])
print('arr3:   ',sess.run(arr3))
print('[1]:    ',sess.run(arr3)[1])
print('[1,1]:  ',sess.run(arr3)[1,1])
print('[1,1,0]:',sess.run(arr3)[1,1,0])
```

Listing 5.23 contains the TF constant `arr1` that is initialized with the value `[[[1,2],[2,3]],[[3,4],[5,6]]]`. The four `print()` statements display the value of `arr1`, the value of the element whose index is 1, the value of the element whose index is `[1,1]`, and the value of the element whose index is `[1,1,0]`. The output from Listing 5.23 (adjusted slightly for display purposes) is here:

```
arr3:
 [[[1 2]
  [2 3]]

 [[3 4]
  [5 6]]]
[1]:
[[3 4]
 [5 6]]
[1,1]:   [5 6]
[1,1,0]: 5
```

MULTIPLYING TWO ARRAYS IN TF (CPU)

Listing 5.24 displays the contents of `tf-cpu.py` that illustrates how to define a TF array and access elements in that array.

LISTING 5.24: tf-cpu.py

```
import tensorflow as tf

with tf.Session() as sess:
  m1 = tf.constant([[3., 3.]])  # 1x2
  m2 = tf.constant([[2.],[2.]]) # 2x1
  p1 = tf.matmul(m1, m2)        # 1x1
  print('m1:',sess.run(m1))
  print('m2:',sess.run(m2))
  print('p1:',sess.run(p1))
```

Listing 5.24 contains two TF constants `m1` and m2 that are initialized with the value `[[3., 3.]]` and `[[2.], [2.]]`, respectively. Due to the nested square brackets, `m1` has shape `1x2`, whereas `m2` has shape `2x1`. Hence, the product of `m1` and `m2` has shape `(1,1)`.

The three `print()` statements display the value of `m1`, `m2`, and `p1`. The output from Listing 5.24 is here:

```
m1: [[3. 3.]]
m2: [[2.]
 [2.]]
p1: [[12.]]
```

MULTIPLYING TWO ARRAYS IN TF (GPU)

Listing 5.25 displays the contents of `tf-gpu.py` that illustrates how to define a TF array and access elements in that array.

LISTING 5.25: tf-gpu.py

```
import tensorflow as tf
```

```
with tf.Session() as sess:
  with tf.device("/gpu:1"):

  m1 = tf.constant([[3., 3.]])  # 1x2
  m2 = tf.constant([[2.],[2.]]) # 2x1
  p1 = tf.matmul(m1, m2)        # 1x1
  print('m1:',sess.run(m1))
  print('m2:',sess.run(m2))
  print('p1:',sess.run(p1))
```

Listing 5.25 contains the same code as Listing 5.24, with the exception of the second with code statement (shown in bold) that specifies the GPU where the TensorFlow code will be executed. Since the TensorFlow code is the same, the output is also the same. The only difference is that the code in Listing 5.25 is executed on a GPU instead of a CPU.

NOTE *Consider using Google Colaboratory that does not require specifying any GPUs, and also provides 12 free hours of GPU usage per day.*

CONVERT PYTHON ARRAYS TO TF ARRAY

Listing 5.26 displays the contents of `tf-convert-tensors.py` that illustrates how to convert a Python array to a TF array.

LISTING 5.26: tf-convert-tensors.py

```
import tensorflow as tf
import numpy as np

x_data = np.array([[1.,2.],[3.,4.]])
x = tf.convert_to_tensor(x_data, dtype=tf.float32)
print ('x1:',x)
sess = tf.Session()
print('x2:',sess.run(x))
```

Listing 5.26 is straightforward, starting with an `import` statement for TensorFlow and one for NumPy. Next, the `x_data` variable is a NumPy array, and `x` is a TF tensor that results from converting `x_data` to a TF tensor. The output from Listing 5.26 is here:

```
x1: Tensor("Const:0", shape=(2, 2), dtype=float32)
x2: [[1. 2.]
 [3. 4.]]
```

WHAT IS TENSORBOARD?

TensorBoard is a very powerful data and graph visualization tool that provides a great deal of useful information, as well as debugging support. Some

of the previous code samples contain code snippets for saving TensorFlow graphs, and this section provides some additional information.

Listing 5.27 displays the contents of `tf-save-data.py` that illustrates how to save a TF graph that can then be viewed in TensorBoard.

LISTING 5.27: tf-save-data.py

```
import tensorflow as tf

x = tf.constant(5,name="x")
y = tf.constant(8,name="y")
z = tf.Variable(2*x+3*y, name="z")
init = tf.global_variables_initializer()

with tf.Session() as session:
  writer = tf.summary.FileWriter("./tf_logs",session.graph)
  session.run(init)
  print('z = ',session.run(z)) # =>  z = 34

# launch tensorboard: tensorboard –logdir=./tf_logs
```

Listing 5.27 starts with code that you have seen in previous code samples. The second portion of Listing 5.27 contains a `with` code block that specifies the `tf_logs` subdirectory of the current directory as the location in which a TensorFlow graph will be saved. This subdirectory will be created if it does not already exist. Next the TF variable `z` is initialized, and then the result of that initialization is printed to standard output. In addition, the value of `z` is saved in an output file (located in the directory `tf_logs`) for the current TensorFlow graph. The output from Listing 5.27 is here:

```
34
```

Now open a command shell, navigate to the location of the TensorFlow code, and launch the following command to invoke TensorBoard (displayed in bold in Listing 5.27):

```
tensorboard –logdir=./tf_logs
```

Now open a browser session and navigate to `localhost:6006`, and you will see the contents of the TensorFlow graph.

Other tips and how-to information about TensorBoard are available here:

https://github.com/tensorflow/tensorboard/blob/master/README.md#my-tensorboard-isnt-showing-any-data-whats-wrong

GOOGLE COLABORATORY

GPU-based TensorFlow code is typically at least 15 times faster than CPU-based TensorFlow code. However, the cost of a good GPU can be a significant

factor. Keep in mind that while NVIDIA provides GPUs, those consumer-based GPUs are not optimized for multi-GPU support (which *is* supported by TensorFlow).

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 is 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 Colab include TensorFlow execution with GPUs, visualization using Matplotlib (discussed in Chapter 4), 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 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 do not have a suitable GPU on their local machine (which is probably the majority of users), and now they launch TensorFlow code to train neural networks in less than 20 or 30 minutes that would otherwise require multiple hours of CPU-based execution time.

Keep in mind the following details about Google Colaboratory. First, whenever you connect to a server in Google Colaboratory, you start what is known as a *session*. You can execute the code in a session with a CPU (the default), a GPU, or a TPU (which incurs a cost), 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.

OTHER CLOUD PLATFORMS

GCP (Google Cloud Platform) is a cloud-based service that enables you to train TensorFlow code in the cloud. GCP provides Deep Learning DL images (similar in concept to Amazon AMIs) that are available here:
https://cloud.google.com/deep-learning-vm/docs

The preceding link provides documentation, and also a link to DL images based on different technologies, including TensorFlow and PyTorch, with GPU and CPU versions of those images. Along with support for multiple versions of Python, you can work in a browser session or from the command line.

GCP SDK

Install GCloud SDK on a Mac-based laptop by downloading the software at this link:
https://cloud.google.com/sdk/docs/quickstart-macos

You will also receive $300 USD worth of credit (over one year) if you have never used Google cloud.

SUMMARY

This chapter introduced you to TensorFlow, its architecture, and some of the tools that are part of the TensorFlow "family." Then you learned how to create a TensorFlow graph containing TensorFlow constants, placeholders, and variables. You also learned how to perform TensorFlow arithmetic operations, along with some built-in functions.

Next, you learned how to calculate trigonometric values, how to use for loops, while loops, and how to calculate exponential values. You also saw how to perform various operations on arrays, such as creating an identity matrix and a constant matrix.

Finally, you learned how to save a TensorFlow graph and then view its contents in a browser using the built-in TensorBoard utility that is launched from the command line.

CHAPTER 6

TENSORFLOW DATASETS

This chapter contains various code samples that illustrate how to use TensorFlow `Datasets`, which support a rich set of operators that can simplify your TensorFlow code and enable you to process very large datasets (i.e., datasets that are too large to fit in memory). You will learn about operators (such as `filter()` and `map()`) that you can specify via "method chaining" as part of the definition of a TF dataset. In addition, you'll learn about TF `estimators` (in the `tf.estimator` namespace), TF `layers` (in the `tf.layers` namespace), and `TFRecords`.

Familiarity with lambda expressions (discussed later) and Reactive Programming will be very helpful for this chapter. In fact, this chapter will be very straightforward if you already have experience with `Observables` in `RxJS`, `RxAndroid`, `RxJava`, or some other environment that involves lazy execution.

The first part of this chapter briefly introduces you to TF `Datasets` and lambda expressions, along with some simple code samples. This section also introduces you to several types of `iterators` in TensorFlow, including "one-shot" iterators and "reiterable" iterators.

The second part of this chapter discusses `TextLineDatasets` that are very convenient for working with text files. The code samples in this section also use two of the four types of iterators that are discussed in the previous section.

The third part of this chapter discusses so-called "intermediate" or "lazy" operators, such as `filter()`, `flatmap()`, and `map()` operators. You can use *method chaining* in order to combine these operators, resulting in powerful code combinations that can significantly reduce the complexity of your TensorFlow code.

The final portion of the chapter briefly discusses TF `estimators`, which are implementations of various Machine Learning algorithms, as well as

TF layers that provide an assortment of classes for DNNs (Dense Neural Networks) and CNNs (Convolutional Neural Networks).

TENSORFLOW DATASETS

TensorFlow Datasets (accessible via the namespace tf.data.Dataset) are well suited for creating asynchronous and optimized data pipelines. In brief, the TF Dataset API loads data from the disk (both images and text), applies optimized transformations, creates batches, and sends the batches to the GPU. Hence, the TF Dataset API is good for better GPU utilization. In addition, use tf.functions in TF 2.0 to fully utilize dataset asynchronous prefetching/streaming features.

A TF Dataset acts as a container-like "wrapper" around a dataset, somewhat analogous to a Pandas DataFrame. A very large dataset can therefore involve a very large TF Dataset. A TF Dataset can also represent an input pipeline as a collection of elements (i.e., a nested structure of tensors), along with a "logical plan" of transformations that act on those elements. For example, you can define a TF Dataset that initially contains the lines of text in a text file, then extract the lines of text that start with a "#" character, and then display only the first three lines. Creating this pipeline is easy: create a TF Dataset and then chain the lazy operators filter() and take(), which is similar to an example that you will see later in this chapter.

Basic Steps for TensorFlow Datasets

Perform the following three steps in order to create and process the contents of a TF Dataset:

1. Import Data
2. Create an Iterator
3. Consume the Data

There are many ways to populate a TF Dataset with data that can be retrieved from multiple sources. For simplicity, the code samples in the first part of this chapter perform the following steps: first create a TF Dataset instance with an initialized NumPy array of data; second, define an iterator instance in order to iterate through the TF Dataset; and third, use the iterator to access the elements of the dataset (and in some cases, supply those elements to a TF model).

A Simple TensorFlow Dataset

Listing 6.1 displays the contents of tf-numpy-dataset.py that illustrates how to create a TF Dataset from a NumPy array of numbers. Although this code sample is minimalistic, it is the starting point in various other code samples in this chapter.

LISTING 6.1: tf-numpy-dataset.py

```
import tensorflow as tf
import numpy as np

x = np.arange(0, 10)

# make a dataset from a numpy array
dataset = tf.data.Dataset.from_tensor_slices(x)
```

Listing 6.1 contains two `import` statements and then initializes the variable `x` as a `NumPy` array with the integers from 0 through 9 inclusive. The variable `dataset` is initialized as a TF `Dataset` that is based on the contents of the variable `x`.

Note that nothing else happens in Listing 6.1; later you will see more meaningful code samples involving TF `Datasets`.

TENSORFLOW RAGGED CONSTANTS AND TENSORS (OPTIONAL)

The section is marked "optional" because the code samples require TF 2. If you launch the code samples using TF 1.x, you will see the following error message:

```
AttributeError: 'module' object has no attribute
                                                'RaggedTensor'
```

As you probably know, every element in a multidimensional array has the same dimensions. For example, a 2x3 array contains two rows and three columns: each row is a 1x3 vector, and each column is a 2x1 vector. As another example, a 2x3x4 array contains two 3x4 arrays (and the same logic applies to each 3x4 array).

On the other hand, a *ragged constant* is a set of elements that have different lengths. You can think of ragged constants as a generalization of "regular" datasets.

Listing 6.2 displays the contents of `tf-raggedtensors1.py` that illustrates how to define a ragged dataset and then iterate through its contents.

LISTING 6.2: tf-raggedtensors1.py

```
import tensorflow as tf

digits = tf.ragged.constant([[3, 1, 4, 1], [], [5, 9, 2],
                                                  [6], []])
words = tf.ragged.constant([["Bye", "now"], ["thank",
                                  "you", "again", "sir"]])

print(tf.add(digits, 3))
print(tf.reduce_mean(digits, axis=1))
print(tf.concat([digits, [[5, 3]]], axis=0))
print(tf.tile(digits, [1, 2]))
print(tf.strings.substr(words, 0, 2))
```

Listing 6.2 defines two ragged constants `digits` and `words` consisting of integers and strings, respectively. The remaining portion of Listing 6.2 consists of five `print()` statements that apply various operations to these two datasets and then display the results.

The first `print()` statement adds the value 3 to every number in the `digits` dataset, and the second `print()` statement computes the row-wise average of the elements of the `digits` dataset because `axis=1` (whereas `axis=0` performs column-wise operations).

The third `print()` statement appends the element `[[5,3]]` to the `digits` dataset, and performs this operation in a column-wise fashion (because `axis=0`). The fourth `print()` statement "doubles" each non-empty element of the `digits` dataset. Finally, the fifth `print()` statement extracts the first two characters from every string in the `words` dataset.

The output from launching the code in Listing 6.2 is here:

```
<tf.RaggedTensor [[6, 4, 7, 4], [], [8, 12, 5], [9], []]>
tf.Tensor([2.25               nan 5.33333333 6.
                             nan], shape=(5,), dtype=float64)
<tf.RaggedTensor [[3, 1, 4, 1], [], [5, 9, 2], [6], [], [5,
                                                        3]]>
<tf.RaggedTensor [[3, 1, 4, 1, 3, 1, 4, 1], [], [5, 9, 2,
                                       5, 9, 2], [6, 6], []]>
<tf.RaggedTensor [[b'By', b'no'], [b'th', b'yo', b'ag',
                                                  b'si']]>
```

Listing 6.3 displays the contents of `tf-ragged-tensors.py` that illustrates how to define a ragged tensor in TensorFlow.

LISTING 6.3: tf-ragged-tensors.py

```
import tensorflow as tf

x1 = tf.RaggedTensor.from_row_splits(
        values=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
        row_splits=[0, 5, 10])
print("x1:",x1)

x2 = tf.RaggedTensor.from_row_splits(
        values=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
        row_splits=[0, 4, 7, 10])
print("x2:",x2)

x3 = tf.RaggedTensor.from_row_splits(
        values=[1, 2, 3, 4, 5, 6, 7, 8],
        row_splits=[0, 4, 4, 7, 8, 8])
print("x3:",x3)
```

Listing 6.3 defines the TF ragged tensors `x1`, `x2`, and `x3` that are based on the integers from 0 to 10 inclusive. The `values` parameter specifies a set of values that will be "split" into a set of vectors, using the numbers in the `row_splits` parameter for the start index and the end index of each vector.

For example, x1 specifies row_splits with the value [0,5,10], which determines two vectors: the vector whose values are from index 0 through index 4 of x1, and the vector whose values are from index 5 through index 9 of x1. The contents of those two vectors are [1, 2, 3, 4, 5] and [6, 7, 8, 9, 10] , respectively (see the output below).

As another example, x2 specifies row_splits with the value [0,4,7,10], which determines three vectors: the vector whose values are from index 0 through index 3 of x1, the vector whose values are from index 4 through index 6 of x1, and the vector whose values are from index 7 through index 9 of x1. The contents of those two vectors are [1,2,3,4], [5,6,7], and [8, 9, 10], respectively (see the output below).

You can perform a similar analysis for x3, keeping in mind that the vector whose start index and end index are [4,4] is an empty vector. The output from launching the code in Listing 6.3 is here:

```
x1: <tf.RaggedTensor [[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]]>
x2: <tf.RaggedTensor [[1, 2, 3, 4], [5, 6, 7], [8, 9, 10]]>
x3: <tf.RaggedTensor [[1, 2, 3, 4], [], [5, 6, 7], [8], []]>
```

If you want to generate a list of values, invoke the to_list() operator. For instance, suppose you define x4 as follows:

```
x4 = tf.RaggedTensor.from_row_splits(
        values=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
        row_splits=[0, 5, 10]).to_list()
print("x4:",x4)
```

The output from the preceding code snippet is here (which you can compare with the output for x1 in the preceding output block):

```
x4: [[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]]
```

You can also create higher-dimensional ragged tensors in TF. For example, the following code snippet creates a two-dimensional ragged tensor in TF:

```
RaggedTensor.from_nested_row_splits(
     flat_values=[3,1,4,1,5,9,2,6],
     nested_row_splits=([0,3,3,5], [0,4,4,7,8,8])).to_list()
```

The preceding code snippet generates the following output:

```
[[[3, 1, 4, 1], [], [5, 9, 2]], [], [[6], []]]
```

WHAT ARE LAMBDA EXPRESSIONS?

In brief, a *lambda expression* is an anonymous function. Use lambda expressions to define local functions that can be passed as arguments or returned as the value of function calls.

Informally, a lambda expression takes an input variable and performs some type of operation (specified by you) on that variable. For example,

here is a "bare bones" lambda expression that adds the number 1 to an input variable x:

```
lambda x: x + 1
```

You can use the preceding lambda expression in a valid TensorFlow code snippet, as shown here (dx is a TensorFlow Dataset that is defined elsewhere):

```
dx.map(lambda x: x + 1)
```

Even if you are unfamiliar with TensorFlow Datasets or the map() operator, you can still understand the preceding code snippet.

The next section contains a complete TensorFlow code samples that illustrates how to use the preceding lambda expression.

A LAMBDA EXPRESSION IN TENSORFLOW

Listing 6.4 displays the contents of tf-plusone.py that illustrates how to use a lambda expression to add the number 1 to the elements of a TF Dataset.

LISTING 6.4: tf-plusone.py

```
import tensorflow as tf
import numpy as np

x = np.arange(0, 10)

# create dataset object from Numpy array
dx = tf.data.Dataset.from_tensor_slices(x)

dx.map(lambda x: x + 1)

# create a one-shot iterator
iterator = dx.make_one_shot_iterator()

# extract an element
next_element = iterator.get_next()

with tf.Session() as sess:
  for i in range(10):
    val = sess.run(next_element)
    print("val:",val)
```

Listing 6.4 initializes the variable x as NumPy array consisting of the integers from 0 through 9 inclusive. Next, the variable dx is initialized as a TF Dataset that is created from the contents of the variable x. Notice how dx.map() then defines a lambda expression that adds one to each input value, which consists of the integers from 0 to 9 in this example.

The next code snippet defines the variable `iterator` as a "one-shot" iterator, followed by the variable `next_element` that is populated with the first integer in `dx` (which is the integer 0).

The final portion of Listing 6.4 iterates through the element of `dx` and displays their values. The output from launching the code in Listing 6.4 is here:

```
('val:', 0)
('val:', 1)
('val:', 2)
('val:', 3)
('val:', 4)
('val:', 5)
('val:', 6)
('val:', 7)
('val:', 8)
('val:', 9)
```

WHAT ARE ITERATORS?

An iterator is similar to a "cursor" in other languages. You can think of an iterator as something that "points" to an item in a dataset. By way of analogy, if you have a linked list of items, an iterator is analogous to a pointer that "points" to the first element in the list, and each time you move the pointer to the next item in the list, you are "advancing" the iterator.

Working with datasets and iterators involves the following steps:

1. create a dataset
2. create an iterator (see next section)
3. "point" the iterator to the dataset
4. print the contents of the current item
5. "advance" the iterator to the next item
6. go to step 4 if there are more items

Notice that Step 6 specifies "if there are more items," which you can handle via a `try/except` block (shown later in this chapter) when the iterator goes beyond the last item in the dataset. This technique is very useful because it obviates the need to know the number of items in a dataset. TensorFlow provides several types of iterators, as discussed in the next section.

TensorFlow Iterators

TensorFlow supports four types of iterators, as listed here:

1. One-shot
2. Initializable
3. Reinitializable
4. Feedable

A *one-shot iterator* can iterate only once through a dataset. After we reach the end of the dataset, the iterator will no longer yield elements; instead, it will raise an `Exception`. For example, if `dx` is an instance of `tf.data.Dataset`, then the following code snippet defines a one-shot iterator:

```
iterator = dx.make_one_shot_iterator()
```

An *initializable iterator* can be dynamically updated: invoke its initializer operation and pass new data via the parameter `feed_dict`. If `dx` is an instance of `tf.data.Dataset`, then the following code snippet defines a reusable iterator:

```
iterator = dx.make_initializable_iterator()
```

A *reinitializable iterator* can be initialized from a different `Dataset`. This type of iterator is very useful when for training datasets that require some additional transformation, such as shuffling its contents.

A *feedable iterator* allows you to select from different iterators: this type of iterator is essentially a "selector" to select an iterator from a collection of iterators.

The code samples in this chapter involve either one-shot iterators or `initializable` iterators. The next section contains a code sample that illustrates how to define a TF `Dataset` and a reusable iterator in TensorFlow.

TENSORFLOW REUSABLE ITERATORS

Listing 6.5 displays the contents of `tf-init-iterator.py` that illustrates how to define a reusable iterator in TensorFlow.

LISTING 6.5: tf-init-iterator.py

```
import tensorflow as tf
import numpy as np

x = np.arange(0, 10)
dx = tf.data.Dataset.from_tensor_slices(x)

# create an initializable iterator
iterator = dx.make_initializable_iterator()

# extract an element
next_element = iterator.get_next()

with tf.Session() as sess:
  sess.run(iterator.initializer)
  for i in range(15):
    val = sess.run(next_element)
    print(val)
    if i % 9 == 0 and i > 0:
      sess.run(iterator.initializer)
```

Listing 6.5 initializes the variable x as NumPy array consisting of the integers from 0 through 9 inclusive. Next, the variable dx is initialized as a TF Dataset that is created from the contents of the variable x. The next code snippet defines the variable iterator as an "initializable" iterator, followed by the variable next_element that is populated with the first integer in dx (which is the integer 0).

The final portion of Listing 6.5 involves a for loop that iterates through the element of dx and displays their values. Notice that the loop iterates through 15 values, whereas dx contains only 10 elements. What will happen?

The answer lies in the conditional logic in the for loop: when the variable i equals 9, the iterator variable is "reset" to its initial value, thereby preventing an error. You can test this situation by "commenting out" the if statement, and when you launch the code you will see the following error message:

```
OutOfRangeError (see above for traceback): End of sequence
[[node IteratorGetNext (defined at tf-init-iterator.py:11) ]]
```

Reset the code in Listing 6.5 to its original contents and you will see the following output when you launch the code in Listing 6.5 here:

```
0
1
2
3
4
5
6
7
8
9
0
1
2
3
4
```

THE TENSORFLOW FILTER() OPERATOR

Listing 6.6 displays the contents of tf-filter.py that illustrates how to use the filter() operator in TensorFlow with a "one-shot" iterator.

LISTING 6.6: tf-filter.py

```
import tensorflow as tf
import numpy as np

#def filter_fn(x):
#  return tf.reshape(tf.not_equal(x % 2, 1), [])

x = np.array([1,2,3,4,5,6,7,8,9,10])
```

```
ds = tf.data.Dataset.from_tensor_slices(x)
ds = ds.filter(lambda x: tf.reshape(tf.not_equal(x%2,1), []))
#ds = ds.filter(filter_fn)

iterator = ds.make_one_shot_iterator()
next_element = iterator.get_next()

with tf.Session() as sess:
  try:
    while True:
      value = sess.run(next_element)
      print("value:",value)
  except tf.errors.OutOfRangeError:
    pass
      sess.run(iterator.initializer)
```

Listing 6.6 initializes the variable `x` as a `NumPy` array consisting of the integers from 1 through 10 inclusive. Next, the variable `ds` is initialized as a TF `Dataset` that is created from the contents of the variable `x`. The next code snippet invokes the `filter()` operator, inside of which a lambda expression returns even numbers because of this expression:

```
tf.not_equal(x%2,1)
```

The next code snippet defines the variable `iterator` as a "one-shot" iterator, followed by the variable `next_element` that is populated with the first integer in `dx` (which is the integer 0).

The final portion of Listing 6.6 contains a `try/except` block, inside of which there is a `while True` code block that iterates through the elements of the dataset until an `tf.errors.OutOfRangeError` error is reached. When this error occurs, the code exits the loop gracefully. The output from launching the code in Listing 6.6 is here:

```
('value:', 2)
('value:', 4)
('value:', 6)
('value:', 8)
('value:', 10)
```

TensorFlow also supports a `TextLineDataset` that is useful for processing the contents of text files. The next several sections contain code samples that show you how to define a `TextLineDataset` and perform simple operations.

TENSORFLOW TEXTLINEDATASET (1)

Listing 6.7 displays the contents of `file.txt` that is referenced in the code in Listing 6.8.

LISTING 6.7: file.txt

```
this is file line #1
this is file line #2
this is file line #3
this is file line #4
this is file line #5
```

Listing 6.8 displays the contents of `tf-textlinedataset1.py` that illustrates another way to define a `TextLineDataset` in TensorFlow.

LISTING 6.8: tf-textlinedataset1.py

```
import tensorflow as tf
import numpy as np

dataset = tf.data.TextLineDataset("file.txt")
dataset = dataset.map(lambda string: tf.string_
                                     split([string]).values)

iterator = dataset.make_initializable_iterator()
next_element = iterator.get_next()
init_op = iterator.initializer

with tf.Session() as sess:
  sess.run(init_op)
  print(sess.run(next_element))
```

Listing 6.8 initializes the variable dataset as a TF `Dataset` that contains the contents of the text file `file.txt`. The next code snippet defines the variable `iterator` as an "initializable" iterator, followed by the variable `next_element` that is populated with the first line of text in `dataset`.

The final portion of Listing 6.8 prints one "tokenized" line of text from the file `file.txt`, as shown here:

```
['this' 'is' 'file' 'line' '#1']
```

TENSORFLOW TEXTLINEDATASET (2)

Listing 6.9 displays the contents of `tf-textlinedataset2.py` that illustrates another way to define a `TextLineDataset` in TensorFlow.

LISTING 6.9: tf-textlinedataset2.py

```
import tensorflow as tf
import numpy as np

dataset = tf.data.TextLineDataset("file.txt")
dataset = dataset.map(lambda string: tf.string_
                                     split([string]).values)

iterator = dataset.make_one_shot_iterator()
next_element = iterator.get_next()
```

```
with tf.Session() as sess:
  print(sess.run(next_element))
  print(sess.run(next_element))
  print(sess.run(next_element))
```

Listing 6.9 initializes the variable `dataset` as a TF `Dataset` that contains the contents of the text file `file.txt`. The next code snippet defines a lambda expression that tokenizes each line of text in the file `file.txt`.

The next code snippet defines the variable `iterator` as an "initializable" iterator, followed by the variable `next_element` that is populated with the first line of text in the file `file.txt`.

The final portion of Listing 6.9 prints three "tokenized" lines of text from the file `file.txt`, as shown here:

```
['this' 'is' 'file' 'line' '#1']
['this' 'is' 'file' 'line' '#2']
['this' 'is' 'file' 'line' '#3']
```

TENSORFLOW TEXTLINEDATASET (3)

Listing 6.10 displays the contents of `tf-textlinedataset3.py` that illustrates a third way to define a `TextLineDataset` in TensorFlow.

LISTING 6.10: tf-textlinedataset3.py

```
import tensorflow as tf
import itertools

dataset = tf.data.TextLineDataset("file.txt")
dataset = dataset.map(lambda string: tf.string_
                                     split([string]).values)

iterator = dataset.make_one_shot_iterator()
next_element = iterator.get_next()

with tf.Session() as sess:
  for i in range(5):
    print(sess.run(next_element))
```

Listing 6.10 initializes the variable `dataset` as a TF `Dataset` that contains the contents of the text file `file.txt`. The next code snippet defines a lambda expression that tokenizes each line of text in the file `file.txt`.

The next code snippet defines the variable `iterator` as a "one-shot" iterator, followed by the variable `next_element` that is populated with the first line of text in the `dataset` variable.

The final portion of Listing 6.10 contains a `for` loop that prints five "tokenized" lines of text from the file `file.txt`, as shown here:

```
['this' 'is' 'file' 'line' '#1']
['this' 'is' 'file' 'line' '#2']
['this' 'is' 'file' 'line' '#3']
```

```
['this' 'is' 'file' 'line' '#4']
['this' 'is' 'file' 'line' '#5']
```

The companion disc also contains the files `tf-textlinedataset4.py` and `tf-textlinedataset5.py` that use `tf.data.TextLineDataset` and lazy operators.

THE TENSORFLOW BATCH() OPERATOR (1)

Listing 6.11 displays the contents of `tf-batch1.py` that illustrates how to use the `batch()` operator in TensorFlow with a reusable iterator.

LISTING 6.11: tf-batch1.py

```
import tensorflow as tf
import numpy as np

x = np.arange(0, 33)
dx = tf.data.Dataset.from_tensor_slices(x).batch(3)
iterator = dx.make_initializable_iterator()

with tf.Session() as sess:
  sess.run(iterator.initializer)
  for i in range(15):
    next_element = iterator.get_next()
    val = sess.run(next_element)
    print(val)
    if (i + 1) % (10 // 3) == 0 and i > 0:
      sess.run(iterator.initializer)
```

Listing 6.11 initializes the variable `x` as `NumPy` array consisting of the integers from 0 through 32 inclusive. Next, the variable `dx` is initialized as a TF `Dataset` that is created from the contents of the variable `x`. Notice that the definition of `x` involves method chaining by "tacking on" the `batch(3)` operator as part of the definition of `dx`.

The next code snippet defines the variable `iterator` as an "initializable" iterator, followed by the variable `next_element` that is populated with the first integer in `dx` (which is the integer 0).

The final portion of Listing 6.11 contains a loop that executes 15 times, and prints five "blocks" of numbers, where a block consists of the following output:

```
[0 1 2]
[3 4 5]
[6 7 8]
```

The reason for this "chunked" effect is because of the conditional logic that "resets" the iterator to the first element of the dataset, as shown here:

```
if (i + 1) % (10 // 3) == 0 and i > 0:
  sess.run(iterator.initializer)
```

Now launch the code in Listing 6.11 to see the output in its entirety, as shown here:

```
[0 1 2]
[3 4 5]
[6 7 8]
[0 1 2]
[3 4 5]
[6 7 8]
[0 1 2]
[3 4 5]
[6 7 8]
[0 1 2]
[3 4 5]
[6 7 8]
[0 1 2]
[3 4 5]
[6 7 8]
```

THE TENSORFLOW BATCH() OPERATOR (2)

Listing 6.12 displays the contents of `tf-batch2.py` that illustrates how to use the `batch()` operator in TensorFlow with a one-shot iterator.

LISTING 6.12: tf-batch2.py

```
import tensorflow as tf
import numpy as np

x = np.arange(0, 33)
dx = tf.data.Dataset.from_tensor_slices(x).batch(3)
iterator = dx.make_one_shot_iterator()
next_element = iterator.get_next()

with tf.Session() as sess:
  for i in range(11):
    val = sess.run(next_element)
    print(val)
```

Listing 6.12 initializes the variable `x` as `NumPy` array consisting of the integers from 0 through 32 inclusive. Next, the variable `dx` is initialized as a TF `Dataset` that is created from the contents of the variable `x`. Notice how method chaining is performed by "tacking on" the `batch(3)` operator as part of the definition of dx.

The next code snippet defines the variable `iterator` as a "one-shot" iterator, followed by the variable `next_element` that is populated with the first integer in `dx` (which is the integer 0).

The final portion of Listing 6.12 contains a loop that executes 11 times, and during each iteration the loop prints eleven "triples" of consecutive integers, starting with the triple `[0 1 2]`. The output from launching the code in Listing 6.12 is here:

```
[0 1 2]
[0 1 2]
```

```
[3 4 5]
[6 7 8]
[ 9 10 11]
[12 13 14]
[15 16 17]
[18 19 20]
[21 22 23]
[24 25 26]
[27 28 29]
[30 31 32]
```

The companion disc contains `tf-batch3.py`, `tf-batch4.py`, and `tf-batch5.py` that illustrate variations of the preceding code sample. Experiment with the code by changing the hard-coded values and then see if you can correctly predict the output.

THE TENSORFLOW MAP() OPERATOR (1)

Listing 6.13 displays the contents of `tf-map.py` that illustrates how to use the `map()` operator in TensorFlow with a one-shot iterator.

LISTING 6.13: tf-map.py

```
import tensorflow as tf
import numpy as np

# a simple Numpy array
x = np.array([[1],[2],[3],[4]])

# make a dataset from a numpy array
dataset = tf.data.Dataset.from_tensor_slices(x)

# a lambda expression to double each value
dataset = dataset.map(lambda x: x*2)

# define an iterator
iter = dataset.make_one_shot_iterator()
el = iter.get_next()

with tf.Session() as sess:
  for _ in range(len(x)):
    print("value:",sess.run(el))
```

Listing 6.13 initializes the variable `x` as a `NumPy` array consisting of four elements, where each element is a 1x1 array consisting of the numbers 1, 2, 3, and 4. Next, the variable `dataset` is initialized as a TF `Dataset` that is created from the contents of the variable `x`. Notice how `dataset.map()` then defines a lambda expression that doubles each input value, which consists of the integers from 1 to 4 in this example.

The next code snippet defines the variable `iter` as a "one-shot" iterator, followed by the variable `el` that is populated with the first element in the variable dataset.

The final portion of Listing 6.13 iterates through the element of `dataset` and displays their values. The output from launching the code in Listing 6.13 is here:

```
('value:', array([2]))
('value:', array([4]))
('value:', array([6]))
('value:', array([8]))
```

THE TENSORFLOW MAP() OPERATOR (2)

Listing 6.14 displays the contents of `tf-map2.py` that illustrates how to invoke the `map()` operator three times in TensorFlow with a one-shot iterator.

LISTING 6.14: tf-map2.py

```
import tensorflow as tf
import numpy as np

# a simple Numpy array
x = np.array([[1],[2],[3],[4]])

# make a dataset from a Numpy array
dataset = tf.data.Dataset.from_tensor_slices(x)

# METHOD #1: THE LONG WAY
# a lambda expression to double each value
#dataset = dataset.map(lambda x: x*2)
# a lambda expression to add one to each value
#dataset = dataset.map(lambda x: x+1)
# a lambda expression to cube each value
#dataset = dataset.map(lambda x: x**3)

# METHOD #2: A SHORTER WAY
dataset = dataset.map(lambda x: x*2).map(lambda x: x+1).
                                        map(lambda x: x**3)

# define an iterator
iter = dataset.make_one_shot_iterator()
el = iter.get_next()

with tf.Session() as sess:
  for _ in range(len(x)):
    print("value:",sess.run(el))
```

Listing 6.14 initializes the variable `x` as a `NumPy` array consisting of four elements, where each element is a 1x1 array consisting of the numbers 1, 2, 3, and 4. Next, the variable `dataset` is initialized as a TF `Dataset` that is created from the contents of the variable `x`.

The next portion of Listing 6.14 is a "commented out" code block that consists of three lambda expressions, followed by a code snippet (shown

in bold) that is a more compact way of defining the same three lambda expressions:

```
dataset = dataset.map(lambda x: x*2).map(lambda x: x+1).
                                        map(lambda x: x**3)
```

The preceding code snippet transforms each input value by first doubling the value, then adding one to the first result, and then cubing the second result.

Although method chaining is a concise way to chain operators, invoking dozens of lazy operators in a single (very long) line of code can quickly become difficult to understand, whereas writing code using the "longer way" would be easier to debug.

A suggestion: start with each lazy operator in a separate line of code, and after you are satisfied that the individual results are correct, *then* use method chaining to combine the operators in a single line of code (up to a maximum of four or five lazy operators).

The next code snippet defines the variable `iter` as a "one-shot" iterator, followed by the variable el that is populated with the first element in the variable `dataset`.

The final portion of Listing 6.16 contains a `for` loop that iterates through the transformed values and displays their values. The output from launching the code in Listing 6.14 is here:

```
('value:', array([27]))
('value:', array([125]))
('value:', array([343]))
('value:', array([729]))
```

THE TENSORFLOW FLATMAP() OPERATOR (1)

In addition to the TF `map()` operator, TF also supports the TF `flat_map()` operator. However, the TF `Dataset.map()` and TF `Dataset.flat_map()` operators expect functions with different signatures. Specifically, `Dataset.map()` takes a function that maps a single element of the input dataset to a single new element, whereas `Dataset.flat_map()` takes a function that maps a single element of the input dataset to a `Dataset` of elements.

Listing 6.15 displays the contents of `tf-flatmap1.py` that illustrates how to use the `flatmap()` operator in TensorFlow with a one-shot iterator.

LISTING 6.15: tf-flatmap1.py

```
import tensorflow as tf
import numpy as np

x = np.array([[1,2,3], [4,5,6], [7,8,9]])

ds = tf.data.Dataset.from_tensor_slices(x)
ds.flat_map(lambda x: tf.data.Dataset.from_tensor_slices(x))
```

```
iterator = ds.make_one_shot_iterator()
next_element = iterator.get_next()

with tf.Session() as sess:
  for i in range(3):
    value = sess.run(next_element)
    print("value:",value)
    next_element = iterator.get_next()
```

Listing 6.15 initializes the variable `x` as a `NumPy` array consisting of three elements, where each element is a 1x3 array of numbers. Next, the variable `ds` is initialized as a TF `Dataset` that is created from the contents of the variable `x`.

The next code snippet defines the variable iterator as a "one-shot" iterator, followed by the variable `next_element` that is populated with the first element in the variable `ds`.

The final portion of Listing 6.15 iterates through the element of `dataset` and displays their values. The output from launching the code in Listing 6.15 is here:

```
('value:', array([1, 2, 3]))
('value:', array([4, 5, 6]))
('value:', array([7, 8, 9]))
```

THE TENSORFLOW FLATMAP() OPERATOR (2)

The code in the previous section works fine, but there is one drawback: there is a hard-coded value 3 in the code block that displays the elements of the dataset. This section removes the hard-coded value.

Listing 6.16 displays the contents of `tf-flatmap2.py` that illustrates how to use the `flatmap()` operator in TensorFlow with a one-shot iterator, and then iterate through the elements of the dataset.

LISTING 6.16: tf-flatmap2.py

```
import tensorflow as tf
import numpy as np

x = np.array([[1,2,3], [4,5,6], [7,8,9]])

ds = tf.data.Dataset.from_tensor_slices(x)
ds.flat_map(lambda x: tf.data.Dataset.from_tensor_slices(x))

iterator = ds.make_one_shot_iterator()
next_element = iterator.get_next()

with tf.Session() as sess:
  try:
    while True:
      value = sess.run(next_element)
      print("value:",value)
  except tf.errors.OutOfRangeError:
    pass
```

Listing 6.16 initializes the variable x as a NumPy array consisting of three elements, where each element is a 1x3 array of numbers. Next, the variable ds is initialized as a TF Dataset that is created from the contents of the variable x.

The next code snippet defines the variable iterator as a "one-shot" iterator, followed by the variable next_element that is populated with the first element in the variable ds.

The final portion of Listing 6.18 iterates through the element of dataset and displays their values. Notice that this code block contains a try/except block, inside of which there is a while True code block that iterates through the elements of the dataset until a tf.errors.OutOfRangeError error is reached. When this error occurs, the code exits the loop gracefully. The output from launching the code in Listing 6.16 is the same as the output from Listing 6.15:

```
('value:', array([1, 2, 3]))
('value:', array([4, 5, 6]))
('value:', array([7, 8, 9]))
```

THE TENSORFLOW FLAT_MAP() AND FILTER() OPERATORS

Listing 6.17 displays the contents of comments.txt and Listing 6.18 displays the contents of tf-flatmap-filter.py that illustrates how to use the filter() operator in TensorFlow with a one-shot iterator.

LISTING 6.17: comments.txt

```
#this is file line #1
#this is file line #2
this is file line #3
this is file line #4
#this is file line #5
```

LISTING 6.18: tf-flatmap-filter.py

```
import tensorflow as tf
import numpy as np

filenames = ["comments.txt"]

dataset = tf.data.Dataset.from_tensor_slices(filenames)

# 1) Use Dataset.flat_map() to transform each file as
# a separate nested dataset, and then concatenate their
# contents sequentially into a single "flat" dataset.
# 2) Skip the first line (header row).
# 3) Filter out lines beginning with "#" (comments).

dataset = dataset.flat_map(
    lambda filename: (
        tf.data.TextLineDataset(filename)
```

```
        .skip(1)
        .filter(lambda line: tf.not_equal(tf.strings.
                                substr(line, 0, 1), "#"))))

iterator = dataset.make_one_shot_iterator()
next_element = iterator.get_next()

with tf.Session() as sess:
  print(sess.run(next_element))
  print(sess.run(next_element))
```

Listing 6.18 defines the variable `filenames` as an array of text filenames, which in this case consists of just one file called `comments.txt` (shown in Listing 6.17). Next, the variable `dataset` is initialized as a TF `Dataset` that contains the contents of `comments.txt`.

The next section of Listing 6.18 is a code block that explains the purpose of the next code block, which involves a moderately complex set of operations that are executed via method chaining in order to transform the contents of the variable dataset.

Specifically, the `dataset.flat_map()` operator "flattens" the contents of its input, which means that the input files (remember, it is just `comments.txt` in this example) are treated as an input stream. Second, the lambda expression "maps" each filename to an instance of the TF `tf.data.TextLineDataset` class. Third, the `skip(1)` operator skips the first line of each input file (`skip(n)` would skip the first n lines). Fourth, the `filter()` operator returns each input line (which is a line of text from each file) if and only if the input line does *not* start with the "#" character.

The next portion of Listing 6.18 defines the variable `iterator` as a "one-shot" iterator, followed by the variable `next_element` that is populated with the first element in the variable `dataset`.

The final portion of Listing 6.18 prints two lines of output, which might seem anticlimactic after defining such a fancy set of transformations!

Launch the code in Listing 6.18 and you will see the following output:

```
this is file line #3
this is file line #4
```

THE TENSORFLOW REPEAT() OPERATOR

Listing 6.19 displays the contents of `tf-repeat.py` that illustrates how to use the `repeat()` operator in TensorFlow.

LISTING 6.19: tf-repeat.py

```
import tensorflow as tf
import numpy as np

ds1 = tf.data.Dataset.from_tensor_slices(tf.range(4))
ds1 = ds1.repeat(2)
```

```
iterator = ds1.make_one_shot_iterator()
next_element = iterator.get_next()

with tf.Session() as sess:
  try:
    while True:
      value = sess.run(next_element)
      print("value:",value)
  except tf.errors.OutOfRangeError:
    pass
```

Listing 6.19 initializes the variable `ds1` as a TF `Dataset` that is created from the integers between 0 and 3 inclusive. The next code snippet "tacks on" the `repeat()` operator to `ds1`, which has the effect of appending the contents of `ds1` to itself.

The next code snippet defines the variable `iterator` as a "one-shot" iterator, followed by the variable `next_element` that is populated with the first integer in `ds1` (which is the integer 0).

The final portion of Listing 6.21 contains a `try/except` block, inside of which there is a `while True` code block that iterates through the elements of the dataset until an `tf.errors.OutOfRangeError` error is reached. When this error occurs, the code exits the loop gracefully. The output from launching the code in Listing 6.19 is here:

```
('value:', 0)
('value:', 1)
('value:', 2)
('value:', 3)
('value:', 0)
('value:', 1)
('value:', 2)
('value:', 3)
```

THE TENSORFLOW TAKE() OPERATOR

Listing 6.20 displays the contents of `tf-take.py` that illustrates how to use the `take()` operator in TensorFlow.

LISTING 6.20: tf-take.py

```
import tensorflow as tf
import numpy as np

ds1 = tf.data.Dataset.from_tensor_slices(tf.range(8))
ds1 = ds1.take(5)

iterator = ds1.make_one_shot_iterator()
next_element = iterator.get_next()

with tf.Session() as sess:
```

```
try:
  while True:
    value = sess.run(next_element)
    print("value:",value)
except tf.errors.OutOfRangeError:
  pass
```

Listing 6.20 initializes the variable `ds1` as a TF `Dataset` that is created from the integers between 0 and 7 inclusive. The next code snippet "tacks on" the `take()` operator to ds1, which has the effect of limiting the output to the first five integers.

The next code snippet defines the variable `iterator` as a "one-shot" iterator, followed by the variable `next_element` that is populated with the first integer in `ds1` (which is the integer 0).

The final portion of Listing 6.20 contains a `try/except` block, inside of which there is a `while True` code block that iterates through the elements of the dataset until an `tf.errors.OutOfRangeError` error is reached. When this error occurs, the code exits the loop gracefully.

The output from launching the code in Listing 6.20 is here:

```
('value:', 0)
('value:', 1)
('value:', 2)
('value:', 3)
('value:', 4)
```

COMBINING THE TF MAP() AND TAKE() OPERATORS

Listing 6.21 displays the contents of `tf-map-take.py` that illustrates how to invoke the `map()` operator followed by the `take()` operator in TensorFlow with a one-shot iterator.

LISTING 6.21: tf-map-take.py

```
import tensorflow as tf
import numpy as np

# a simple Numpy array
x = np.array([[1],[2],[3],[4]])

# make a dataset from a Numpy array
dataset = tf.data.Dataset.from_tensor_slices(x)

tf.enable_eager_execution()

# a simple Numpy array
x = np.array([[1],[2],[3],[4]])

# make a dataset from a numpy array
dataset = tf.data.Dataset.from_tensor_slices(x)
```

```
dataset = dataset.map(lambda x: x*2).map(lambda x: x+1).
                                          map(lambda x: x**3)

# define an iterator
iter = dataset.make_one_shot_iterator()
el = iter.get_next()

for value in dataset.take(2):
  print("value:",value)
```

Listing 6.21 initializes the variable `x` as a `NumPy` array consisting of four elements, where each element is a 1x1 array consisting of the numbers 1, 2, 3, and 4. Next, the variable `dataset` is initialized as a TF `Dataset` that is created from the contents of the variable `x`.

The next portion of Listing 6.21 involves three lambda expressions that's shown in bold and reproduced here:

```
dataset = dataset.map(lambda x: x*2).map(lambda x: x+1).
                                          map(lambda x: x**3)
```

The preceding code snippet transforms each input value by first doubling the value, then adding one to the first result, and then cubing the second result.

The next code snippet defines the variable `iter` as a "one-shot" iterator, followed by the variable `el` that is populated with the first element in the variable `dataset`.

The final portion of Listing 6.21 "takes" only the first two elements from the variable `dataset` and displays their contents, as shown here:

```
('value:', <tf.Tensor: id=36, shape=(1,), dtype=int64,
                                          numpy=array([27])>)
('value:', <tf.Tensor: id=38, shape=(1,), dtype=int64,
                                         numpy=array([125])>)
```

Note that eager execution *must* be enabled; otherwise you will see the following error message:

```
RuntimeError: dataset.__iter__() is only supported when
                                    eager execution is enabled.
```

COMBINING THE TF ZIP() AND BATCH() OPERATORS

Listing 6.22 displays the contents of `tf-zip-batch.py` that illustrates how to combine the `zip()` and `batch()` operators in TensorFlow.

LISTING 6.22: tf-zip-batch.py

```
import tensorflow as tf
import numpy as np

ds1 = tf.data.Dataset.range(100)
ds2 = tf.data.Dataset.range(0, -100, -1)
```

```
ds3 = tf.data.Dataset.zip((ds1, ds2))
ds4 = ds3.batch(4)

iterator = ds4.make_one_shot_iterator()
next_element = iterator.get_next()

with tf.Session() as sess:
  print(sess.run(next_element))
  print(sess.run(next_element))
  print(sess.run(next_element))
```

Listing 6.22 initializes the variables `ds1`, `ds2`, `ds3`, and `ds4` as TF `Datasets` that are created successively, starting from `ds1` that contains the integers between 0 and 99 inclusive.

The next portion of Listing 6.22 defines the variable `iterator` as a "one-shot" iterator, followed by the variable `next_element` that is populated with the first element in the variable `dataset`. The final portion of Listing 6.22 prints three lines of "batched" output, as shown here:

```
(array([0, 1, 2, 3]), array([ 0, -1, -2, -3]))
(array([4, 5, 6, 7]), array([-4, -5, -6, -7]))
(array([ 8,  9, 10, 11]), array([ -8,  -9, -10, -11]))
```

COMBINING THE TF ZIP() AND TAKE() OPERATORS

Listing 6.23 displays the contents of `tf-zip-take.py` that illustrates how to combine the `zip()` and `take()` operators in TensorFlow.

LISTING 6.23: tf-zip-take.py

```
import tensorflow as tf
import numpy as np

x = np.arange(0, 10)
y = np.arange(1, 11)

# create dataset objects from the arrays
dx = tf.data.Dataset.from_tensor_slices(x)
dy = tf.data.Dataset.from_tensor_slices(y)

# zip the two datasets together
dcomb = tf.data.Dataset.zip((dx, dy)).batch(3)
iterator = dcomb.make_initializable_iterator()

# extract an element
next_element = iterator.get_next()

with tf.Session() as sess:
  sess.run(iterator.initializer)
  for i in range(15):
    val = sess.run(next_element)
    print(val)
    if (i + 1) % (10 // 3) == 0 and i > 0:
      sess.run(iterator.initializer)
```

Listing 6.23 initializes the variables `x` and y as a range of integers from 0 to 11 and from 0 to 12, respectively. Next, the variables `dx` and `dy` are initialized as TF `Datasets` that are created from the contents of the variables x and y, respectively.

The next code snippet defines the variable `dcomb` as a TF `Dataset` that combines the elements from `dx` and `dy` in a pairwise fashion via the `zip()` operator, as shown here:

```
dcomb = tf.data.Dataset.zip((dx, dy)).batch(3)
```

Notice how method chaining is performed by "tacking on" the `batch(3)` operator as part of the definition of `dcomb`.

The next code snippet defines the variable `iterator` as an "initializable" iterator, followed by the variable `next_element` that is populated with the first integer in `dx` (which is the integer 0).

The final portion of Listing 6.23 contains a loop that executes 15 times, and during each iteration the loop prints the current contents of the variable `iterator`. Each line of output consists of two "blocks" of numbers, where a block consists of three consecutive integers. The output from launching the code in Listing 6.23 is here:

```
(array([0, 1, 2]), array([1, 2, 3]))
(array([3, 4, 5]), array([4, 5, 6]))
(array([6, 7, 8]), array([7, 8, 9]))
(array([0, 1, 2]), array([1, 2, 3]))
(array([3, 4, 5]), array([4, 5, 6]))
(array([6, 7, 8]), array([7, 8, 9]))
(array([0, 1, 2]), array([1, 2, 3]))
(array([3, 4, 5]), array([4, 5, 6]))
(array([6, 7, 8]), array([7, 8, 9]))
(array([0, 1, 2]), array([1, 2, 3]))
(array([3, 4, 5]), array([4, 5, 6]))
(array([6, 7, 8]), array([7, 8, 9]))
(array([0, 1, 2]), array([1, 2, 3]))
(array([3, 4, 5]), array([4, 5, 6]))
(array([6, 7, 8]), array([7, 8, 9]))
```

TF DATASETS AND RANDOM NUMBERS (1)

Listing 6.24 displays the contents of `tf-random-one-shot.py` that illustrates how to create a TF `Dataset` with random numbers.

LISTING 6.24: tf-random-one-shot.py

```
import tensorflow as tf
import numpy as np

x = np.random.sample((100,2))

# make a dataset from a numpy array
dataset = tf.data.Dataset.from_tensor_slices(x)
```

```
iter = dataset.make_one_shot_iterator()
el = iter.get_next()

with tf.Session() as sess:
  print(sess.run(el))
```

Listing 6.24 initializes the variable `x` as a `NumPy` array consisting of 100 rows and 2 columns of randomly generated numbers. Next, the variable `dataset` is initialized as a TF `Dataset` that is created from the contents of the variable `x`.

The next code snippet defines the variable `iter` as a "one-shot" iterator, followed by the variable `el` that is populated with the first element in the variable dataset.

The final portion of Listing 6.24 prints the first line of transformed data, as shown here:

```
[0.69707335 0.21129127]
```

TF DATASETS AND RANDOM NUMBERS (2)

Listing 6.25 displays the contents of `tf-random-one-shot2.py` that illustrates how to create a TF `Dataset` with random numbers.

LISTING 6.25: tf-random-one-shot2.py

```
import tensorflow as tf
import numpy as np

# two Numpy arrays with random numbers
features, labels = (np.random.sample((100,2)), np.random.
                                                sample((100,1)))
dataset = tf.data.Dataset.from_tensor_slices
                                          ((features,labels))

iter = dataset.make_one_shot_iterator()
el = iter.get_next()

with tf.Session() as sess:
  print(sess.run(el))
```

Listing 6.25 initializes the variables `features` and `labels` that are generated from a `NumPy` array consisting of 100 rows and 2 columns of randomly generated numbers, and from a `NumPy` array consisting of 100 rows and 1 column of randomly generated numbers.

Next, the variable `dataset` is initialized as a TF `Dataset` that is created from the contents of the variables `features` and `labels`. The next code snippet defines the variable `iter` as a "one-shot" iterator, followed by the variable `el` that is populated with the first element in the variable `dataset`. The final portion of Listing 6.25 prints the first line of transformed data, as shown here:

```
(array([0.47582573, 0.36387037]), array([0.26763744]))
```

TF DATASETS FROM CSV FILES

Listing 6.26 displays the contents of `simple.csv`, and Listing 6.27 displays the contents of `tf-csv-dataset.py` that illustrates how to create a TF `Dataset` from data in a CSV file.

LISTING 6.26: simple.csv

```
text,sentiment
'a good restaurant', 1
'a poor movie ', 0
'a great dessert', 1
'a plain hamburger', 0
```

LISTING 6.27: tf-csv-dataset.py

```
import tensorflow as tf
import numpy as np

csv_file = './simple.csv'
dataset = tf.data.experimental.make_csv_dataset(csv_file,
                                                batch_size=2)
iter = dataset.make_one_shot_iterator()
next = iter.get_next()

# next is a dict with key=columns names and value=column
                                                        data
print("next:",next)
inputs, labels = next['text'], next['sentiment']

with tf.Session() as sess:
  sess.run([inputs, labels])
  print("inputs:",sess.run([inputs, labels]))
```

Listing 6.27 initializes the variable `csv_file` with the name of a CSV file (which is `simple.csv` in this example). Next, the variable `dataset` is initialized as a CSV-based TF `Dataset` that is created from the contents of the variable `csv_file`. The next code snippet defines the variable `iter` as a "one-shot" iterator, followed by the variable `next` that is populated with the first element in the variable `dataset`.

The next portion of Listing 6.27 initializes the variables `inputs` and `labels` with the first column and second column, respectively, of the first line of data from the CSV file. The last portion of Listing 6.27 displays the values of `inputs` and `labels`, as shown here:

```
('next:', OrderedDict([('text', <tf.Tensor 'IteratorGetNext:1'
        shape=(2,) dtype=string>), ('sentiment', <tf.Tensor
            'IteratorGetNext:0' shape=(2,) dtype=int32>)]))
('inputs:', [array(["'a poor movie '", "'a good restaurant'"],
                dtype=object), array([0, 1], dtype=int32)])
```

WORKING WITH TF.ESTIMATORS AND TF.LAYERS (OPTIONAL)

The first sub-section introduced below is useful if you have some experience with well-known Machine Learning algorithms using a Python library such as `scikit-learn`. You will see a list of the TensorFlow classes that are similar to their Python-based counterparts in Machine Learning, with classes for regression tasks and classes for classification tasks.

The second subsection contains a list of TensorFlow classes that are relevant for defining `CNN`s (Convolutional Neural Networks) in TensorFlow.

If you are new to Machine Learning then this section will have much more limited value to you right now, but you can still learn what is available for future reference.

What are TF Estimators?

The `tf.estimator` namespace contains an assortment of classes that implement various algorithms that are available in Machine Learning, such as boosted trees, `DNN` classifiers, `DNN` regressors, linear classifiers, and linear regressors.

The estimator-related classes `DNNRegressor`, `LinearRegressor`, and `DNNLinearCombinedRegressor` are for regression tasks, whereas the classes `DNNClassifier`, `LinearClassifier`, and `DNNLinearCombinedClassifier` are for classification tasks. A more extensive list of estimator classes (with very brief descriptions) is listed below:

- BoostedTreesClassifier: A Classifier for Tensorflow Boosted Trees models
- BoostedTreesRegressor: A Regressor for Tensorflow Boosted Trees models
- CheckpointSaverHook: Saves checkpoints every N steps or seconds
- DNNClassifier: A classifier for TensorFlow DNN models
- DNNEstimator: An estimator for TensorFlow DNN models with user-specified head
- DNNLinearCombinedClassifier: An estimator for TensorFlow Linear and DNN joined classification models
- DNNLinearCombinedRegressor: An estimator for TensorFlow Linear and DNN joined models for regression
- DNNRegressor: A regressor for TensorFlow DNN models
- Estimator: Estimator class to train and evaluate TensorFlow models
- LinearClassifier: Linear classifier model
- LinearEstimator: An estimator for TensorFlow linear models with user-specified head
- LinearRegressor: An estimator for TensorFlow Linear regression problems

All estimator classes are in the `tf.estimator` namespace, and all the estimator classes inherit from the `tf.estimator.Estimator` class. Read the online documentation for the details of the preceding classes as well as online tutorials for relevant code samples.

TF Estimators and tf.data.Dataset (optional)

This section is marked optional because you need to launch the code with TF 2 instead of TF 1.x.

In addition to creating a dataset from NumPy arrays of data or from `Pandas Dataframes`, you can create a dataset from existing datasets. For example, Listing 6.28 displays the contents of `tf2-mnist-estimator.py` that illustrates how to create a `tf.data.Dataset` from the `MNIST` dataset.

LISTING 6.28: tf2-mnist-estimator.py

```
import tensorflow as tf

train, test = tf.keras.datasets.mnist.load_data()
mnist_x, mnist_y = train

mnist_ds = tf.data.Dataset.from_tensor_slices(mnist_x)
print(mnist_ds)

iterator = tf.compat.v1.data.make_one_shot_iterator(mnist_
                                                    ds)

count = 0
max_count = 4

try:
  while True:
    print("value:",iterator.get_next())
    count += 1
    if(count > max_count):
      break
except tf.errors.OutOfRangeError:
  pass
```

Listing 6.28 starts by initializing the variables `train` and `test` that contain the training data and the test data, respectively. Next, the mnist_ds variable is initialized as an instance of the `tf.data.Dataset` class. The variable iterator is initialized as a "one shot" iterator, which works in TF 2 because of the `tf.compat.v1.data` namespace that is used to define the variable `iterator`.

The next portion of Listing 6.28 initializes two scalar variables, followed by a `try/except` block that contains a `while` loop. The code inside the `while` loop prints the current value of the iterator variable as it "iterates" through the elements in `mnist_ds`.

The output from launching the code in Listing 6.30 is here:

```
<TensorSliceDataset shapes: (28, 28), types: tf.uint8>
value: tf.Tensor(
[[  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   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   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   3  18
                                                  18  18 126 136
  175  26 166 255 247 127   0   0   0   0]
 [  0   0   0   0   0   0   0   0  30  36  94 154 170 253
                                                 253 253 253 253
  225 172 253 242 195  64   0   0   0   0]
 [  0   0   0   0   0   0   0  49 238 253 253 253 253 253
                                                 253 253 253 251
   93  82  82  56  39   0   0   0   0   0]
 [  0   0   0   0   0   0   0  18 219 253 253 253 253 253
                                                 198 182 247 241
    0   0   0   0   0   0   0   0   0   0]
 // output omitted for brevity
[  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
                                                 218 252  56   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
                                                  96 252 189  42
    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
                                                  14 184 252 170
   11   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  14 147 252
   42   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]], shape=(28,
                                                  28), dtype=uint8)
```

What are tf.layers?

The `tf.layers` namespace contains an assortment of classes for the layers in Neural Networks, including DNNs (Dense Neural Networks) and CNNs (Convolutional Neural Networks). Some of the more common classes in the `tf.layers` namespace are listed below:

- BatchNormalization: Batch Normalization layer
- Conv2D: 2D convolution layer (e.g., spatial convolution over images)
- Dense: Densely-connected layer class
- Dropout: Applies Dropout to the input
- Flatten: Flattens an input tensor while preserving the batch axis (axis 0)

- Layer: Base layer class
- MaxPooling2D: Max pooling layer for 2D inputs (e.g., images)

For example, a minimalistic CNN starts with a "triple" that consists of a `Conv2D` layer, followed by `ReLU` (Rectified Linear Unit) activation function, and then a `MaxPooling2D` layer. If you see this triple appear a second time, followed by two consecutive `Dense` layers and then a softmax activation function, it is known as "LeNet."

A bit of trivia: in the late 1990s, when people deposited checks at an automated bank machine, LeNet scanned the contents of those checks to determine the digits of the check amount (of course, customers had to confirm that the number determined by LeNet was correct). LeNet had an accuracy rate around 90%, which is a very impressive result for such a simple Convolutional Neural Network!

Other Useful Parts of TensorFlow

In addition to the classes in the previous sections, TensorFlow provides a number of other useful namespaces, including the following list:

- tf.data
- tf.linalg
- tf.lite
- tf.losses
- tf.math
- tf.nn
- tf.random
- tf.saved_model
- tf.test
- tf.train
- tf.version

The `tf.data` namespace contains the `tf.data.Dataset` namespace, which contains classes that are discussed in the first half of this chapter; the `tf.linalg` namespace contains classes an assortment of classes that perform operations in linear algebra; the `tf.lite` namespace contains classes for mobile application development.

The `tf.math` namespace contains classes for trigonometric calculations; the `tf.nn` namespace contains classes for batch normalization, RNNs, dropout rate, and max pooling. Read the extensive online documentation for more details regarding these (and other classes) in TensorFlow.

WHAT IS A TFRECORD?

A `TFRecord` is a file that describes the data required during the training phase and the testing phase of a model. There are two protocol buffer

message types for available for a TFRecord: the Example message type and the SequenceExample message type. These protocol buffer message types enable you to arrange data as a map from string keys to values that are lists of integers, 32-bit floats, or bytes.

The data in a TFRecord is "wrapped" inside a Feature class. In addition, each feature is stored in a key value pair, where the key corresponds to the title that is allotted to each feature. These titles are used later for extracting the data from TFRecord. The created dictionary is passed as input to a Feature class. Finally, the features object is passed as input to Example class that is appended into the TFRecord. The preceding process is repeated for every type of data that is stored in TFRecord.

The TFRecord file format is a record-oriented binary format that you can use for training data. In addition, the tf.data.TFRecordDataset class enables you to stream over the contents of one or more TFRecord files as part of an input pipeline.

You can store any type of data, including images, in the Example format. However, you specify the mechanism for arranging the data into serialized bytes, as well as reconstructing the original format.

A Simple TFRecord

Listing 6.29 displays the contents of tf-record1.py that illustrates how to define a TFRecord.

LISTING 6.29: tf-record1.py

```
import tensorflow as tf

simple1 = tf.train.Example(features=tf.train.
                                        Features(feature={
  'my_ints':  tf.train.Feature(int64_list=tf.train.
                                  Int64List(value=[2, 5])),
  'my_float': tf.train.Feature(float_list=tf.train.
                                   FloatList(value=[3.6])),
  'my_bytes': tf.train.Feature(bytes_list=tf.train.
                                 BytesList(value=['data']))
}))

print("my_ints:",  simple1.featurs.feature['my_ints'].
                                          int64_list.value)
print("my_floats:",simple1.features.feature['my_float'].float_
                                               list.value)
print("my_bytes:", simple1.features.feature['my_bytes'].
                                          bytes_list.value)

#print("simple1:",simple1)
```

Listing 6.29 contains the definition of the variable simple1 that is an instance of the tf.train.Example class. The simple1 variable defines a record consisting of the fields my_ints, my_floats, and my_bytes that are of

type `Int64List`, `FloatList`, and `ByteList`, respectively. The final portion of Listing 6.29 contains `print()` statements that display the values of various elements in the `simple1` variable, as shown here:

```
('my_ints:', [2L, 5L])
('my_floats:', [3.5999999046325684])
('my_bytes:', ['data'])
```

SUMMARY

This chapter introduced you to TensorFlow `Datasets` and iterators that are well suited for processing the contents of "normal" size datasets as well datasets that are too large to fit in memory. You saw how to define a lambda expression and use that expression in a TensorFlow `Dataset`.

Next, you learned about various "lazy operators," including `filter()`, `map()`, `filter()`, `flatmap()`, `take()`, and `zip()`, and how to use them to define a subset of the data in a TensorFlow `Dataset`. You also learned how to use one-shot iterators and reuseable iterators in TensorFlow in order to iterate through the elements of a TensorFlow `Datasets`.

Then you got a brief introduction to the `tf.estimators` namespace, which contains an assortment of classes that implement various algorithms, such as boosted trees, `DNN` classifiers, `DNN` regressors, linear classifiers, and linear regressors. After that you learned that the `tf.layers` namespace contains an assortment of classes for `DNNs` (Dense Neural Networks) and `CNNs` (Convolutional Neural Networks).

Finally, you learned about a TensorFlow `TFRecord`, which is a file that describes the data required during the training phase and the testing phase of a model.

INDEX