Python 3 for Machine Learning

O. Campesato

Python 3 for Machine Learning

PYTHON 3 FOR MACHINE LEARNING

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
1-800-232-0223

O. Campesato. *Python 3 for Machine Learning*.
ISBN: 978-1-68392-495-1

Library of Congress Control Number: 2020930258

202122321 Printed on acid-free paper in the United States of America.

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

CONTENTS

PREFACE

What is the Primary Value Proposition for this Book?

This book endeavors to provide you with as much relevant information about Python and machine learning as possible that can be reasonably included in a book of this size.

The Target Audience

This book is intended to reach an international audience of readers with highly diverse backgrounds in various age groups. While many readers know how to read English, their native spoken language is not English (which could be their second, third, or even fourth language). Consequently, this book uses standard English rather than colloquial expressions that might be confusing to those readers. As you know, many people learn by different types of imitation, which includes reading, writing, or hearing new material. This book takes these points into consideration in order to provide a comfortable and meaningful learning experience for the intended readers.

Getting the Most from this Book

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

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

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

Why Are Software Installation Instructions not Included?

There are useful websites containing installation instructions for Python for various platforms. Instead of repeating those instructions in this book, that space is used for Python material. In general, this book attempts to avoid “filler” content as well as easily accessible set-up steps that are available online.

How Was the Code for This Book Tested?

The code samples in this book have been tested in Python version 3.6.8 on a Macbook Pro with OS X 10.8.5.

What Do I Need to Know for This Book?

The most useful prerequisite is some familiarity with another scripting language, such as Perl or PHP. Knowledge of other programming languages (such as Java) can also be helpful because of the exposure to programming concepts and constructs. The less technical knowledge that you have, the more diligence will be required in order to understand the various topics that are covered. Basic machine learning is helpful but not required.

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

Why Doesn't This Book Have 600 or More Pages?

The target audience consists of readers ranging from beginners to intermediate in terms of their knowledge of programming languages. During the preparation of this book, every effort has been made to accommodate those readers so that they will be adequately prepared to explore more advanced features of Python during their self-study.

Why so Many Code Samples in the Chapters?

One of the primary rules of exposition of virtually any kind is "show, don't tell." While this rule is not taken literally in this book, it’s the motivation for showing first and telling second. You can decide for yourself if show-first-then-tell is valid in this book by performing a simple experiment: when you see the code samples and the accompanying graphics effects in this book, determine if it's more effective to explain ("tell") the visual effects or to show them. If the adage “a picture is worth a thousand words” is true, then this book endeavors to provide both the pictures and the words.

Do the Companion Files Obviate the Need for this Book?

The companion files contain all of the code samples to save you time and effort from the error-prone process of manually typing code into a text file. Moreover, the book provides explanations that assist you in understanding the code samples.

The code samples are available for download by writing to the publisher at info@merclearning.com.

Does this Book Contain Production-Level Code Samples?

The code samples show you some features of Python3 that are useful for machine learning. In addition, clarity has higher priority than writing more compact code that is more difficult to understand (and possibly more prone to bugs). If you decide to use any of the code in this book in a production environment, submit that code to the same rigorous analysis as the other parts of your code base.

CHAPTER 1

INTRODUCTION TO PYTHON 3

In This Chapter

- Tools for Python
- Python Installation
- Setting the `PATH` Environment Variable (Windows Only)
- Launching Python on Your Machine
- Python Identifiers
- Lines, Indentation, and Multilines
- Quotation and Comments in Python
- Saving Your Code in a Module
- Some Standard Modules in Python
- The `help()` and `dir()` Functions
- Compile Time and Runtime Code Checking
- Simple Data Types in Python
- Working with Numbers
- Working with Fractions
- Unicode and UTF-8
- Working with Unicode
- Working with Strings
- Uninitialized Variables and the Value None in Python

- Slicing and Splicing Strings
- Search and Replace a String in Other Strings
- Remove Leading and Trailing Characters
- Printing Text without NewLine Characters
- Text Alignment
- Working with Dates
- Exception Handling in Python
- Handling User Input
- Command-Line Arguments
- Summary

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

The first part of this chapter covers how to install Python, some Python environment variables, and how to use the Python interpreter. You will see Python code samples, and you will also learn 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 documentation websites 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 are not compatible with Python 3.

1.1 Tools for Python

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

http://continuum.io/downloads

Anaconda is well-suited for modules such as `numPy` and `sciPy` (discussed in Chapter 7), and if you are a Windows user, Anaconda appears to be a better alternative.

1.1.1 `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.

1.1.2 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.

1.1.3 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 3.6.8 (v3.6.8:3c6b436a57, Dec 24 2018, 02:04:31)
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.

1.2 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:

```
python3 -V
```

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

```
Python 3.6.8
```

Note: install Python 3.6.8 (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 downloads link or navigate directly to this website:

http://www.python.org/download/

In addition, PythonWin is available for Windows, and its home page is here:

http://www.cgl.ucsf.edu/Outreach/pc204/pythonwin.html

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

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

1.3 Setting the `PATH` Environment Variable (Windows Only)

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

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

1.4 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.

1.4.1 The Python Interactive Interpreter

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

```
python3
```

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

```
Python 3.6.8 (v3.6.8:3c6b436a57, Dec 24 2018, 02:04:31)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)]
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
```

1.5 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` different identifiers in Python.

In addition, Python has the following naming convention:

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

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

1.6 Lines, Indentation, and Multilines

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

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

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

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

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

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

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

The output of the preceding code snippet is here:

```
10
15
```

Note: the use of semicolons and the continuation character are discouraged in Python.

1.7 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
```

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

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

The output of the preceding code block is here:

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

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

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

This will produce following result:

```
Hello, Python!
```

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

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

You can comment multiple lines as follows:

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

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

1.8 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, that 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.

1.9 Some Standard Modules in Python

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

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

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

The code samples in this book use the modules `math`, `os`, `random`, `re`, `socket`, `sys`, `time`, and `urllib`. You need to import these modules in order to use them in your code. For example, the following code block shows you how to import 4 standard Python modules:

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

The code samples in this book import one or more of the preceding modules, as well as other Python modules. In Chapter 8, you will learn how to write Python modules that import other user-defined Python modules.

1.10 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.

1.11 Compile Time and Runtime Code Checking

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

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

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

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

In Chapter 2, you will learn how to define and invoke user-defined functions, along with an explanation of the difference between local versus global variables in Python.

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.

1.12 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, all of which are discussed in Chapter 3. 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.

1.13 Working with Numbers

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

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

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

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

The following examples demonstrate arithmetic operations involving integers:

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

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

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

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

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

The output from the preceding code block is here:

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

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

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

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

1.13.1 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 wan 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'
```

1.13.2 The `chr()` Function

The Python `chr()` function takes a positive integer as a parameter and converts it to its corresponding alphabetic value (if one exists). The letters A through Z have decimal representation of `65` through `91` (which corresponds to hexadecimal `41` through `5b`), and the lowercase letters `a` through `z` have decimal representation `97` through `122` (hexadecimal `61` through `7b`).

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

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

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

```
result = ""
```

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

Note: Python 2 uses ASCII strings whereas Python 3 uses UTF-8.

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

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

However, the following equivalent code snippet is more intuitive:

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

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

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

1.13.3 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
```

1.13.4 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'
```

1.14 Working with Fractions

Python supports the `Fraction()` function (which is define in the `fractions` module) that accepts two integers that represent the numerator and the denominator (which must be nonzero) of a fraction. Several example 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.

1.15 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, Unicode Transformation Format (UTF)-8 is perhaps the most common, and it's also the default encoding for many systems. The digit 8 in UTF-8 indicates that the encoding uses 8-bit numbers, whereas UTF-16 uses 16-bit numbers (but this encoding is less common).

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

1.16 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.

1.17 Working with Strings

A string in Python 2 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 shows you how easy it is to extract the letters in a text string, and in Chapter 3 you will learn how to “unpack” other Python data structures.

You can extract substrings of a string as shown in the following examples:

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

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

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

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

1.17.1 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
```

1.17.2 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
```

1.18 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 (shown in Chapter 2).

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

1.19 Slicing 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.

1.19.1 Testing for Digits and Alphabetic Characters

Python enables you to examine each character in a string and then test whether that character is a bona fide digit or an alphabetic character. This section provides a precursor to regular expressions that are discussed in Chapter 4.

Listing 1.3 displays the contents of `CharTypes.py` that illustrates how to determine if a string contains digits or characters. In case you are unfamiliar with the conditional "if" statement in Listing 1.3, more detailed information is available in Chapter 2.

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 2 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
```

1.20 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
```

1.21 Remove Leading and Trailing Characters

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

Listing 1.6: Remove1.py

```
text = '   leading and trailing white space   '
print('text1:','x',text,'y')
text = text.lstrip()
print('text2:','x',text,'y')
text = text.rstrip()
print('text3:','x',text,'y')
```

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

The output from launching Listing 1.6 is here:

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

If you want to remove extra white spaces inside a text string, use the `replace()` function as discussed in the previous section. The following example illustrates how this can be accomplished, which also contains the `re` module as a "preview" for what you will learn in Chapter 4:

```
import re
text = 'a    b'
```

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

The result is here:

```
ab
a b
```

Chapter 2 shows you how to use the `join()` function in order to remove extra white spaces in a text string.

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

```
import sys
write = sys.stdout.write
```

```
write('123')
write('123456789')
The output is here:
1233
1234567899
```

1.23 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     '
>>>
```

1.24 Working with Dates

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

https://docs.python.org/3/library/datetime.html

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

Listing 1.7: Datetime2.py

```
import time
import datetime

print("Time in seconds since the epoch: %s" %time.time())
print("Current date and time: " , datetime.datetime.
now())
print("Or like this: " ,datetime.datetime.now().
strftime("%y-%m-%d-%H-%M"))
print("Current year: ", datetime.date.today().
strftime("%Y"))
print("Month of year: ", datetime.date.today().
strftime("%B"))
print("Week number of the year: ", datetime.date.
today().strftime("%W"))
print("Weekday of the week: ", datetime.date.today().
strftime("%w"))
print("Day of year: ", datetime.date.today().
strftime("%j"))
print("Day of the month : ", datetime.date.today().
strftime("%d"))
print("Day of week: ", datetime.date.today().
strftime("%A"))
```

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

Listing 1.8: datetime2.out

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

Python also enables you to perform arithmetic calculates 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
```

1.24.1 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
```

1.25 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.

1.26 Handling User Input

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

Listing 1.11: UserInput1.py

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

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

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

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

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

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

Listing 1.12: UserInput2.py

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

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

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

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

Listing 1.13: UserInput3.py

```
sum = 0
msg = 'Enter a number:'
```

```
val1 = input(msg)
   try:
  sum = sum + eval(val1)
except:
  print(val1,'is a string')
msg = 'Enter a number:'
val2 = input(msg)
try:
  sum = sum + eval(val2)
except:
  print(val2,'is a string')
print('The sum of',val1,'and',val2,'is',sum)
```

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

1.27 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 preceding script as follows:

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

This will produce following result:

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

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

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

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

This functionality is very useful because you can programmatically execute different methods in a Python class, which means that you can write unit tests for your code as well. Read Chapter 8 to learn how to create custom Python classes.

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.

1.28 Summary

This chapter showed you how to work with numbers and perform arithmetic operations on numbers, and then you learned how to work with strings and use string operations. The next chapter shows you how to work with conditional statements, loops, and user-defined functions in Python.

CHAPTER 2

CONDITIONAL LOGIC, LOOPS, AND FUNCTIONS

- Precedence of Operators in Python
- Python Reserved Words
- Working with Loops in Python
- Nested Loops
- The `split()` Function with for Loops
- Using the `split()` Function to Compare Words
- Using the `split()` Function to Print Justified Text
- Using the `split()` Function to Print Fixed Width Text
- Using the `split()` Function to Compare Text Strings
- Using the `split()` Function to Display Characters in a String
- The `join()` Function
- Python `while` Loops
- Conditional Logic in Python
- The `break/continue/pass` Statements
- Comparison and Boolean Operators
- Local and Global Variables
- Scope of Variables
- Pass by Reference versus Value
- Arguments and Parameters

- Using a `while` loop to Find the Divisors of a Number
- User-Defined Functions in Python
- Specifying Default Values in a Function
- Functions with a Variable Number of Arguments
- Lambda Expressions
- Recursion
- Summary

This chapter introduces you to various ways to perform conditional logic in Python, as well as control structures and user-defined functions in Python. Virtually every Python program that performs useful calculations requires some type of conditional logic or control structure (or both). Although the syntax for these Python features is slightly different from other languages, the functionality will be familiar to you.

The first part of this chapter contains code samples that illustrate how to handle `if-else` conditional logic in Python, as well as `if-elsif-else` statements. The second part of this chapter discusses loops and while statements in Python. This section contains an assortment of examples (comparing strings, computing numbers raised to different exponents, and so forth) that illustrate various ways that you can use loops and while statements in Python. The third part of this chapter contains examples that involve nested loops and recursion. The final part of this chapter introduces you to user-defined Python functions.

2.1 Precedence of Operators in Python

When you have an expression involving numbers, you might remember that multiplication ("*") and division ("/") have higher precedence than addition ("+") or subtraction ("-"). Exponentiation has even higher precedence than these four arithmetic operators.

However, instead of relying on precedence rules, it's simpler (as well as safer) to use parentheses. For example, `(x/y)+10` is clearer than `x/y+10`, even though they are equivalent expressions.

As another example, the following two arithmetic expressions are the equivalent, but the second is less error prone than the first:

```
x/y+3*z/8+x*y/z-3*x
x/y)+(3*z)/8+(x*y)/z-(3*x)
```

In any case, the following website contains precedence rules for operators in Python:

http://www.mathcs.emory.edu/~valerie/courses/fall10/155/resources/op_precedence.html

2.2 Python Reserved Words

Every programming language has a set of reserved words, which is a set of words that cannot be used as identifiers, and Python is no exception. The Python reserved words are: `and, exec, not, assert, finally, or, break, for, pass, class, from, print, continue, global, raise, def, if, return, del, import, try, elif, in, while, else, is, with, except, lambda,` and `yield.`

If you inadvertently use a reserved word as a variable, you will see an "invalid syntax" error message instead of a "reserved word" error message. For example, suppose you create a Python script `test1.py` with the following code:

```
break = 2
print('break =', break)
```

If you run the preceding Python code you will see the following output:

```
  File "test1.py", line 2
    break = 2
          ^
SyntaxError: invalid syntax
```

However, a quick inspection of the Python code reveals the fact that you are attempting to use the reserved word `break` as a variable.

2.3 Working with Loops in Python

Python supports `for` loops, `while` loops, and `range()` statements. The following subsections illustrate how you can use each of these constructs.

2.3.1 Python for Loops

Python supports the `for` loop whose syntax is slightly different from other languages (such as JavaScript and Java). The following code block shows you how to use a `for` loop in Python in order to iterate through the elements in a list:

```
>>> x = ['a', 'b', 'c']
>>> for w in x:
...   print(w)
...
a
b
c
```

The preceding code snippet prints three letters on three separate lines. You can force the output to be displayed on the same line (which will "wrap" if you specify a large enough number of characters) by appending a comma "," in the print statement, as shown here:

```
>>> x = ['a', 'b', 'c']
>>> for w in x:
...   print(w, end=' ')
...
a b c
```

You can use this type of code when you want to display the contents of a text file in a single line instead of multiple lines.

Python also provides the built-in `reversed()` function that reverses the direction of the loop, as shown here:

```
>>> a = [1, 2, 3, 4, 5]
>>> for x in reversed(a):
... print(x)
5
4
3
2
1
```

Note that reversed iteration only works if the size of the current object can be determined or if the object implements a `__reversed__()` special method.

2.3.2 A `for` Loop with `try/except` in Python

Listing 2.1 displays the contents of `StringToNums.py` that illustrates how to calculate the sum of a set of integers that have been converted from strings.

Listing 2.1: StringToNums.py

```
line = '1 2 3 4 10e abc'
sum  = 0
invalidStr = ""

print('String of numbers:',line)

for str in line.split(" "):
  try:
    sum = sum + eval(str)
  except:
    invalidStr = invalidStr + str + ' '

print('sum:', sum)
if(invalidStr != ""):
  print('Invalid strings:',invalidStr)
else:
  print('All substrings are valid numbers')
```

Listing 2.1 initializes the variables `line, sum, and invalidStr`, and then displays the contents of line. The next portion of Listing 2.1 splits the contents of `line` into words, and then uses a `try` block in order to add the numeric value of each word to the variable sum. If an exception occurs, the contents of the current `str` is appended to the variable `invalidStr`.

When the loop has finished execution, Listing 2.1 displays the sum of the numeric words, followed by the list of words that are not numbers. The output from Listing 2.1 is here:

```
String of numbers: 1 2 3 4 10e abc
sum: 10
Invalid strings: 10e abc
```

2.3.3 Numeric Exponents in Python

Listing 2.2 displays the contents of `Nth_exponent.py` that illustrates how to calculate intermediate powers of a set of integers.

Listing 2.2: Nth_exponent.py

```
maxPower = 4
maxCount = 4

def pwr(num):
  prod = 1
  for n in range(1,maxPower+1):
    prod = prod*num
    print(num,'to the power',n, 'equals',prod)
  print('-----------')

for num in range(1,maxCount+1):
    pwr(num)
```

Listing 2.2 contains a function called `pwr()` that accepts a numeric value. This function contains a loop that prints the value of that number raised to the power n, where n ranges between `1` and `maxPower+1`.

The second part of Listing 2.2 contains a `for` loop that invokes the function `pwr()` with the numbers between `1` and `maxPower+1`. The output from Listing 2.2 is here:

```
1 to the power 1 equals 1
1 to the power 2 equals 1
1 to the power 3 equals 1
1 to the power 4 equals 1
-----------
2 to the power 1 equals 2
2 to the power 2 equals 4
2 to the power 3 equals 8
2 to the power 4 equals 16
-----------
3 to the power 1 equals 3
3 to the power 2 equals 9
3 to the power 3 equals 27
3 to the power 4 equals 81
-----------
4 to the power 1 equals 4
4 to the power 2 equals 16
4 to the power 3 equals 64
4 to the power 4 equals 256
-----------
```

2.4 Nested Loops

Listing 2.3 displays the contents of `Triangular1.py` that illustrates how to print a row of consecutive integers (starting from `1`), where the length of each row is one greater than the previous row.

Listing 2.3: Triangular1.py

```
max = 8
for x in range(1,max+1):
  for y in range(1,x+1):
    print(y, '', end='')
  print()
```

Listing 2.3 initializes the variable `max` with the value 8, followed by an outer `for` loop whose loop variable `x` ranges from `1` to `max+1`. The inner loop has a loop variable `y` that ranges from `1` to `x+1`, and the inner loop prints the value of `y`. The output of Listing 2.4 is here:

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
1 2 3 4 5 6
1 2 3 4 5 6 7
1 2 3 4 5 6 7 8
```

2.5 The split() Function with for Loops

Python supports various useful string-related functions, including the `split()` function and the `join()` function. The `split()` function is useful when you want to tokenize ("split") a line of text into words and then use a `for` loop to iterate through those words and process them accordingly.

The `join()` function does the opposite of `split()`: it "joins" two or more words into a single line. You can easily remove extra spaces in a sentence by using the `split()` function and then invoking the `join()` function, thereby creating a line of text with one white space between any two words.

2.6 Using the `split()` Function to Compare Words

Listing 2.4 displays the contents of `Compare2.py` that illustrates how to use the split function to compare each word in a text string with another word.

Listing 2.4: Compare2.py

```
x = 'This is a string that contains abc and Abc'
```

```
y = 'abc'
identical = 0
casematch = 0

for w in x.split():
  if(w == y):
    identical = identical + 1
  elif (w.lower() == y.lower()):
    casematch = casematch + 1

if(identical > 0):
 print('found identical matches:', identical)

if(casematch > 0):
 print('found case matches:', casematch)

if(casematch == 0 and identical == 0):
 print('no matches found')
```

Listing 2.4 uses the `split()` function in order to compare each word in the string `x` with the word `abc`. If there is an exact match, the variable `identical` is incremented. If a match does not occur, a case-insensitive match of the current word is performed with the string `abc`, and the variable `casematch` is incremented if the match is successful.

The output from Listing 2.5 is here:

```
found identical matches: 1
found case matches: 1
```

2.7 Using the `split()` Function to Print Justified Text

Listing 2.5 displays the contents of `FixedColumnCount.py` that illustrates how to print a set of words from a text string as justified text using a fixed number of columns.

Listing 2.5: FixedColumnCount1.py

```
import string

wordCount = 0
str1 = 'this is a string with a set of words in it'

print('Left-justified strings:')
print('-----------------------')
for w in str1.split():
   print('%-10s' % w)
   wordCount = wordCount + 1
   if(wordCount % 2 == 0):
      print("")
print("\n")

print('Right-justified strings:')
print('------------------------')

wordCount = 0
for w in str1.split():
   print('%10s' % w)
   wordCount = wordCount + 1
   if(wordCount % 2 == 0):
      print()
```

Listing 2.5 initializes the variables `wordCount` and `str1`, followed by two `for` loops. The first `for` loop prints the words in `str1` in left-justified format, and the second `for` loop prints the words in `str1` in right-justified format. In both loops, a linefeed is printed after a pair of consecutive words is printed, which occurs whenever the variable `wordCount` is even. The output from Listing 2.5 is here:

```
Left-justified strings:
-----------------------
this       is
a          string
with       a
set        of
words      in
it

Right-justified strings:
------------------------
      this        is
         a    string
      with         a
       set        of
     words        in
        it
```

2.8 Using the `split()` Function to Print Fixed Width Text

Listing 2.6 displays the contents of `FixedColumnWidth1.py` that illustrates how to print a text string in a column of fixed width.

Listing 2.6: FixedColumnWidth1.py

```
import string

left = 0
right = 0
columnWidth = 8

str1 = 'this is a string with a set of words in it and
it will be split into a fixed column width'
strLen = len(str1)

print('Left-justified column:')
print('----------------------')
rowCount = int(strLen/columnWidth)

for i in range(0,rowCount):
   left  = i*columnWidth
   right = (i+1)*columnWidth-1
   word  = str1[left:right]
   print("%-10s" % word)

# check for a 'partial row'
if(rowCount*columnWidth < strLen):
   left  = rowCount*columnWidth-1;
   right = strLen
   word  = str1[left:right]
   print("%-10s" % word)
```

Listing 2.6 initializes the integer variable `columnWidth` and the string variable `str1`. The variable strLen is the length of `str1`, and `rowCount` is `strLen` divided by `columnWidth`.

The next part of Listing 2.6 contains a loop that prints `rowCount` rows of characters, where each row contains `columnWidth` characters. The final portion of Listing 2.6 prints any “leftover” characters that comprise a partial row.

The newspaper-style output (but without any partial whitespace formatting) from Listing 2.6 is here:

```
Left-justified column:
----------------------
this is
a strin
 with a
set of
ords in
```

```
it and
t will
e split
into a
ixed co
umn wid
th
```

2.9 Using the `split()` Function to Compare Text Strings

Listing 2.7 displays the contents of `CompareStrings1.py` that illustrates how to determine whether or not the words in one text string are also words in a second text string.

Listing 2.7: CompareStrings1.py

```
text1 = 'a b c d'
text2 = 'a b c e d'

if(text2.find(text1) >= 0):
  print('text1 is a substring of text2')
else:
  print('text1 is not a substring of text2')

subStr = True
for w in text1.split():
  if(text2.find(w) == -1):
    subStr = False
    break

if(subStr == True):
  print('Every word in text1 is a word in text2')
else:
  print('Not every word in text1 is a word in text2')
```

Listing 2.7 initializes the string variables `text1` and `text2`, and uses conditional logic to determine whether or not `text1` is a substring of `text2` (and then prints a suitable message).

The next part of Listing 2.7 is a loop that iterates through the words in the string `text1` and checks if each of those words is also a word in the string `text2`. If a nonmatch occurs, the variable `subStr` is set to False, followed by the break statement that causes an early exit from the loop. The final portion of Listing 2.7 prints the appropriate message based on the value of `subStr`. The output from Listing 2.7 is here:

```
text1 is not a substring of text2
Every word in text1 is a word in text2
```

2.10 Using a Basic for Loop to Display Characters in a String

Listing 2.8 displays the contents of `StringChars1.py` that illustrates how to print the characters in a text string.

Listing 2.8: StringChars1.py

```
text = 'abcdef'
for ch in text:
   print('char:',ch,'ord value:',ord(ch))
print
```

Listing 2.8 is straightforward: a `for` loop iterates through the characters in the string `text` and then prints the character and its `ord` value. The output from Listing 2.8 is here:

```
('char:', 'a', 'ord value:', 97)
('char:', 'b', 'ord value:', 98)
('char:', 'c', 'ord value:', 99)
('char:', 'd', 'ord value:', 100)
('char:', 'e', 'ord value:', 101)
('char:', 'f', 'ord value:', 102)
```

2.11 The `join()` Function

Another way to remove extraneous spaces is to use the `join()` function, as shown here:

```
text1 = '   there are     extra   spaces    '
print('text1:',text1)

text2 = ' '.join(text1.split())
print('text2:',text2)

text2 = 'XYZ'.join(text1.split())
print('text2:',text2)
```

The `split()` function "splits" a text string into a set of words, and also removes the extraneous white spaces. Next, the `join()` function "joins" together the words in the string `text1`, using a single white space as the delimiter. The last code portion of the preceding code block uses the string `XYZ` as the delimiter instead of a single white space.

The output of the preceding code block is here:

```
text1:    there are     extra   spaces
text2: there are extra spaces
text2: thereXYZareXYZextraXYZspaces
```

2.12 Python while Loops

You can define a `while` loop to iterate through a set of numbers, as shown in the following examples:

```
>>> x = 0
>>> while x < 5:
...   print(x)
...   x = x + 1
...
0
1
2
3
4
5
```

Python uses indentation instead of curly braces that are used in other languages such as JavaScript and Java. Although the Python list data structure is not discussed until Chapter 3, you can probably understand the following simple code block that contains a variant of the preceding while loop that you can use when working with lists:

```
lst  = [1,2,3,4]

while lst:
  print('list:',lst)
  print('item:',lst.pop())
```

The preceding `while` loop terminates when the `lst` variable is empty, and there is no need to explicitly test for an empty list. The output from the preceding code is here:

```
list: [1, 2, 3, 4]
item: 4
list: [1, 2, 3]
item: 3
list: [1, 2]
item: 2
list: [1]
item: 1
```

This concludes the examples that use the `split()` function in order to process words and characters in a text string. The next part of this chapter shows you examples of using conditional logic in Python code.

2.13 Conditional Logic in Python

If you have written code in other programming languages, you have undoubtedly seen `if/then/else (or if-elseif-else)` conditional statements. Although the syntax varies between languages, the logic is essentially the same. The following example shows you how to use `if/elif` statements in Python:

```
>>> x = 25
>>> if x < 0:
...   print('negative')
... elif x < 25:
...   print('under 25')
... elif x == 25:
...   print('exactly 25')
... else:
...  print('over 25')
...
exactly 25
```

The preceding code block illustrates how to use multiple conditional statements, and the output is exactly what you expected.

2.14 The `break/continue/pass` Statements

The break statement in Python enables you to perform an "early exit" from a loop, whereas the continue statement essentially returns to the top of the loop and continues with the next value of the loop variable. The `pass` statement is essentially a "do nothing" statement.

Listing 2.9 displays the contents of `BreakContinuePass.py` that illustrates the use of these three statements.

Listing 2.9: BreakContinuePass.py

```
print('first loop')
for x in range(1,4):
  if(x == 2):
    break
  print(x)

print('second loop')
for x in range(1,4):
  if(x == 2):
    continue
  print(x)

print('third loop')
for x in range(1,4):
  if(x == 2):
    pass
  print(x)
```

The output of Listing 2.9 is here:

```
first loop
1
second loop
1
3
third loop
1
2
3
```

2.15 Comparison and Boolean Operators

Python supports a variety of Boolean operators, such as `in`, `not in`, `is`, `is not`, `and`, `or`, and `not`. The next several sections discuss these operators and provide some examples of how to use them.

2.15.1 The `in/not in/is/is not` Comparison Operators

The `in` and `not` in operators are used with sequences to check whether a value occurs or does not occur in a sequence. The operators `is` and `is not` determine whether or not two objects are the same object, which is important only for mutable objects such as lists. All comparison operators have the same priority, which is lower than that of all numerical operators. Comparisons can also be chained. For example, `a < b == c` tests whether a is less than `b` and moreover `b` equals `c`.

2.15.2 The and, or, and not Boolean Operators

The Boolean operators `and`, `or`, and `not` have lower priority than comparison operators. The Boolean `and` and `or` are binary operators whereas the Boolean `or` operator is a unary operator. Here are some examples:

`A and B` can only be true if both A and B are true

`A or B` is true if either A or B is true

`not(A)` is true if and only if A is false

You can also assign the result of a comparison or other Boolean expression to a variable, as shown here:

```
>>> string1, string2, string3 = '', 'b', 'cd'
>>> str4 = string1 or string2 or string3
>>> str4
'b'
```

The preceding code block initializes the variables `string1`, `string2`, and `string3`, where string1 is an empty string. Next, `str4` is initialized via the `or` operator, and since the first non-null value is `string2`, the value of `str4` is equal to `string2`.

2.16 Local and Global Variables

Python variables can be local or global. A Python variable is local to a function if the following are true:

- a parameter of the function
- on the left-side of a statement in the function
- bound to a control structure (such as for, with, and except)

A variable that is referenced in a function but is not local (according to the previous list) is a non-local variable. You can specify a variable as non-local with this snippet:

```
nonlocal z
```

A variable can be explicitly declared as global with this statement:

```
global z
```

The following code block illustrates the behavior of a global versus a local variable:

```
global z
z = 3

def changeVar(z):
  z = 4
  print('z in function:',z)

print('first global z:',z)

if __name__ == '__main__':
  changeVar(z)
  print('second global z:',z)
```

The output from the preceding code block is here:

```
first global z: 3
z in function: 4
second global z: 3
```

2.17 Scope of Variables

The accessibility or scope of a variable depends on where that variable has been defined. Python provides two scopes: global and local, with the added “twist” that global is actually module-level scope (i.e., the current file), and therefore you can have a variable with the same name in different files and they will be treated differently.

Local variables are straightforward: they are defined inside a function, and they can only be accessed inside the function where they are defined. Any variables that are not local variables have global scope, which means that those variables are “global” *only* with respect to the file where it has been defined, and they can be accessed anywhere in a file.

There are two scenarios to consider regarding variables. First, suppose two files (aka modules) `file1.py` and `file2`.py have a variable called `x`, and `file1.py` also imports `file2.py`. The question now is how to disambiguate between the `x` in the two different modules. As an example, suppose that `file2.py` contains the following two lines of code:

```
x = 3
print('unscoped x in file2:',x)
```

Suppose that `file1.py` contains the following code:

```
import file2 as file2

x = 5
```

```
print('unscoped x in file1:',x)
print('scoped x from file2:',file2.x)
```

Launch `file1.y` from the command line, and you will see the following output:

```
unscoped x in file2: 3
unscoped x in file1: 5
scoped x from file2: 3
```

The second scenario involves a program contains a local variable and a global variable with the same name. According to the earlier rule, the local variable is used in the function where it is defined, and the global variable is used outside of that function.

The following code block illustrates the use of a global and local variable with the same name:

```
#!/usr/bin/python
# a global variable:
total = 0;

def sum(x1, x2):
   # this total is local:
   total = x1+x2;

   print("Local total : ", total)
   return total

# invoke the sum function
sum(2,3);
print("Global total : ", total)
```

When the above code is executed, it produces following result:

```
Local total :   5
Global total :  0
```

What about unscoped variables, such as specifying the variable `x` without a module prefix? The answer consists of the following sequence of steps that Python will perform:

1. check the local scope for the name
2. ascend the enclosing scopes and check for the name
3. perform step #2 until the global scope (ie module level)
4. if x still hasn't been found, Python checks__builtins__

```
Python 3.6.8 (v3.6.8:3c6b436a57, Dec 24 2018, 02:04:31)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)]
on darwin
Type "help", "copyright", "credits" or "license" for
more information.
>>> x = 1
>>> g = globals()
>>> g
{'g': {...}, '__builtins__': <module '__builtin__'
(built-in)>, '__package__': None, 'x': 1, '__name__':
'__main__', '__doc__': None}
>>> g.pop('x')
1
>>> x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

Note: You can access the `dicts` that Python uses to track local and global scope by invoking `locals()` and `globals()` respectively.

2.18 Pass by Reference versus Value

All parameters (arguments) in the Python language are passed by reference. Thus, if you change what a parameter refers to within a function, the change is reflected in the calling function. For example:

```
def changeme(mylist):
   #This changes a passed list into this function
   mylist.append([1,2,3,4])
   print("Values inside the function: ", mylist)
   return

# Now you can call changeme function
mylist = [10,20,30]
changeme(mylist)
print("Values outside the function: ", mylist)
```

Here we are maintaining reference of the passed object and appending values in the same object, and the result is shown here:

```
Values inside the function:  [10, 20, 30, [1, 2, 3, 4]]
Values outside the function:  [10, 20, 30, [1, 2, 3, 4]]
```

The fact that values are passed by reference gives rise to the notion of mutability versus immutability that is discussed in Chapter 3.

2.19 Arguments and Parameters

Python differentiates between arguments to functions and parameter declarations in functions: a positional (mandatory) and keyword (optional/default value). This concept is important because Python has operators for packing and unpacking these kinds of arguments.

Python unpacks positional arguments from an iterable, as shown here:

```
>>> def foo(x, y):
...   return x - y
...
>>> data = 4,5
>>> foo(data) # only passed one arg
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: foo() takes exactly 2 arguments (1 given)
>>> foo(*data) # passed however many args are in tuple
-1
```

2.20 Using a `while` loop to Find the Divisors of a Number

Listing 2.10 contains a `while` loop, conditional logic, and the % (modulus) operator in order to find the factors of any integer greater than `1`.

Listing 2.10: Divisors.py

```
def divisors(num):
  div = 2

  while(num > 1):
    if(num % div == 0):
      print("divisor: ", div)
      num = num / div
    else:
      div = div + 1
  print("** finished **")

divisors(12)
```

Listing 2.10 defines a function `divisors()` that takes an integer value num and then initializes the variable `div` with the value `2`. The while loop divides `num` by `div` and if the remainder is `0`, it prints the value of `div` and then it divides `num` by `div`; if the value is not `0`, then `div` is incremented by 1. This `while` loop continues as long as the value of `num` is greater than `1`.

The output from Listing 2.10 passing in the value `12` to the function `divisors()` is here:

```
divisor:  2
divisor:  2
divisor:  3
** finished **
```

Listing 2.11 displays the contents of `Divisors2.py` that contains a `while` loop, conditional logic, and the `%` (modulus) operator in order to find the factors of any integer greater than `1`.

Listing 2.11: Divisors2.py

```
def divisors(num):
  primes = ""
  div = 2

  while(num > 1):
    if(num % div == 0):
      divList = divList + str(div) + ' '
      num = num / div
    else:
      div = div + 1
  return divList

result = divisors(12)
print('The divisors of',12,'are:',result)
```

Listing 2.11 is very similar to Listing 2.10: the main difference is that Listing 2.10 constructs the variable `divList` (which is a concatenated list of the divisors of a number) in the `while` loop, and then returns the value of `divList` when the `while` loop has completed. The output from Listing 2.11 is here:

```
The divisors of 12 are: 2 2 3
```

2.20.1 Using a `while` loop to Find Prime Numbers

Listing 2.12 displays the contents of `Divisors3.py` that contains a `while` loop, conditional logic, and the `%` (modulus) operator in order to count the number of prime factors of any integer greater than 1. If there is only one divisor for a number, then that number is a prime number.

Listing 2.12: Divisors3.py

```
def divisors(num):
  count = 1
  div = 2
  while(div < num):
    if(num % div == 0):
      count = count + 1
    div = div + 1
  return count

result = divisors(12)

if(result == 1):
  print('12 is prime')
else:
  print('12 is not prime')
```

2.21 User-Defined Functions in Python

Python provides built-in functions and also enables you to define your own functions. You can define functions to provide the required functionality. Here are simple rules to define a function in Python:

- Function blocks begin with the keyword `def` followed by the function name and parentheses.
- Any input arguments should be placed within these parentheses.
- The first statement of a function can be an optional statement—the documentation string of the function or docstring.
- The code block within every function starts with a colon (:) and is indented.
- The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return `None`.
- If a function does not specify return statement, the function automatically returns `None`, which is a special type of value in Python.

A very simple custom Python function is here:

```
>>> def func():
```

```
...   print 3
...
>>> func()
3
```

The preceding function is trivial, but it does illustrate the syntax for defining custom functions in Python. The following example is slightly more useful:

```
>>> def func(x):
...   for i in range(0,x):
...     print(i)
...
>>> func(5)
0
1
2
3
4
```

2.22 Specifying Default Values in a Function

Listing 2.13 displays the contents of `DefaultValues.py` that illustrates how to specify default values in a function.

Listing 2.13: DefaultValues.py

```
def numberFunc(a, b=10):
  print (a,b)

def stringFunc(a, b='xyz'):
  print (a,b)

def collectionFunc(a, b=None):
  if(b is None):
     print('No value assigned to b')

numberFunc(3)
stringFunc('one')
collectionFunc([1,2,3])
```

Listing 2.13 defines three functions, followed by an invocation of each of those functions. The functions `numberFunc()` and `stringFunc()` print a list contain the values of their two parameters, and `collectionFunc()` displays a message if the second parameter is `None`. The output from Listing 2.13 is here:

```
(3, 10)
('one', 'xyz')
No value assigned to b
```

2.22.1 Returning Multiple Values from a Function

This task is accomplished by the code in Listing 2.14, which displays the contents of `MultipleValues.py`.

Listing 2.14: MultipleValues.py

```
def MultipleValues():
    return 'a', 'b', 'c'

x, y, z = MultipleValues()

print('x:',x)
print('y:',y)
print('z:',z)
```

The output from Listing 2.14 is here:

```
x: a
y: b
z: c
```

2.23 Functions with a Variable Number of Arguments

Python enables you to define functions with a variable number of arguments. This functionality is useful in many situations, such as computing the sum, average, or product of a set of numbers. For example, the following code block computes the sum of two numbers:

```
def sum(a, b):
    return a + b

values = (1, 2)
s1 = sum(*values)
print('s1 = ', s1)
```

The output of the preceding code block is here:

```
s1 =  3
```

However, the sum function in the preceding code block can only be used for two numeric values.

Listing 2.15 displays the contents of `VariableSum1.py` that illustrates how to compute the sum of a variable number of numbers.

Listing 2.15:VariableSum1.py

```
def sum(*values):
  sum = 0
  for x in values:
    sum = sum + x
  return sum

values1 = (1, 2)
s1 = sum(*values1)
print('s1 = ',s1)

values2 = (1, 2, 3, 4)
s2 = sum(*values2)
print('s2 = ',s2)
```

Listing 2.15 defines the function sum whose parameter values can be an arbitrary list of numbers. The next portion of this function initializes sum to `0`, and then a `for` loop iterates through values and adds each of its elements to the variable `sum`. The last line in the function `sum()` returns the value of the variable `sum`. The output from Listing 2.15 is here:

```
s1 =  3
s2 =  10
```

2.24 Lambda Expressions

Listing 2.16 displays the contents of `Lambda1.py` that illustrates how to create a simple lambda function in Python.

Listing 2.16 Lambda1.py

```
add = lambda x, y: x + y

x1 = add(5,7)
x2 = add('Hello', 'Python')

print(x1)
print(x2)
```

Listing 2.16 defines the lambda expression add that accepts two input parameters and then returns their sum (for numbers) or their concatenation (for strings).

The output from Listing 2.16 is here:

```
12
HelloPython
```

2.25 Recursion

Recursion is a powerful technique that can provide an elegant solution to various problems. The following subsections contain examples of using recursion to calculate some well-known numbers.

2.25.1 Calculating Factorial Values

The factorial value of a positive integer n is the product of all the integers between 1 and n. The symbol for factorial is the exclamation point ("!") and some sample factorial values are here:

```
1! = 1
2! = 2
3! = 6
4! = 20
5! = 120
```

The formula for the factorial value of a number is succinctly defined as follows:

```
Factorial(n) = n*Factorial(n-1) for n > 0 and
Factorial(0) = 1
```

Listing 2.17 displays the contents of `Factorial.py` that illustrates how to use recursion in order to calculate the factorial value of a positive integer.

Listing 2.17 Factorial.py

```
def factorial(num):
  if (num > 1):
    return num * factorial(num-1)
  else:
    return 1

result = factorial(5)
print('The factorial of 5 =', result)
```

Listing 2.17 contains the function `factorial` that implements the recursive definition of the factorial value of a number. The output from Listing 2.17 is here:

```
The factorial of 5 = 120
```

In addition to a recursive solution, there is also an iterative solution for calculating the factorial value of a number. Listing 2.18 displays the contents of `Factorial2.py` that illustrates how to use the `range()` function in order to calculate the factorial value of a positive integer.

Listing 2.18: Factorial2.py

```
def factorial2(num):
  prod = 1
  for x in range(1,num+1):
    prod = prod * x
  return prod

result = factorial2(5)
print 'The factorial of 5 =', result
```

Listing 2.18 defines the function `factorial2()` with a parameter `num`, followed by the variable prod which has an initial value of 1. The next part of `factorial2()` is a for loop whose loop variable `x` ranges between `1` and `num+1`, and each iteration through that loop multiples the value of `prod` with the value of `x`, thereby computing the factorial value of `num`. The output from Listing 2.18 is here:

```
The factorial of 5 = 120
```

2.25.2 Calculating Fibonacci Numbers

The set of Fibonacci numbers represent some interesting patterns (such as the pattern of a sunflower) in nature, and its recursive definition is here:

```
Fib(0) = 0
Fib(1) = 1
Fib(n) = Fib(n-1) + Fib(n-2) for n >= 2
```

Listing 2.19 displays the contents of `fib.py` that illustrates how to calculate Fibonacci numbers.

Listing 2.19: fib.py

```
def fib(num):
  if (num == 0):
    return 1
  elif (num == 1):
    return 1
  else:
    return fib(num-1) + fib(num-2)

result = fib(10)
print('Fibonacci value of 5 =', result)
```

Listing 2.19 defines the `fib()` function with the parameter num. If `num` equals `0` or `1` then `fib()` returns `num`; otherwise, `fib()` returns the result of adding `fib(num-1)` and `fib(num-2)`. The output from Listing 2.19 is here:

```
Fibonacci value of 10 = 89
```

2.25.3 Calculating the GCD of Two Numbers

The greatest common divisor (GCD) of two positive integers is the largest integer that divides both integers with a remainder of `0`. Some values are shown here:

```
gcd(6,2)   = 2
gcd(10,4)  = 2
gcd(24,16) = 8
```

Listing 2.20 uses recursion and Euclid's algorithm in order to find the GCD of two positive integers.

Listing 2.20: gcd.py

```
def gcd(num1, num2):
  if(num1 % num2 == 0):
    return num2
  elif (num1 < num2):
    print("switching ", num1, " and ", num2)
    return gcd(num2, num1)
  else:
    print("reducing", num1, " and ", num2)
    return gcd(num1-num2, num2)

result = gcd(24, 10)
print("GCD of", 24, "and", 10, "=", result)
```

Listing 2.20 defines the function `gcd()` with the parameters `num1` and `num2`. If `num1` is divisible by `num2`, the function returns `num2`. If `num1` is less than `num2`, then gcd is invoked by switching the order of `num1` and `num2`. In all other cases, `gcd()` returns the result of computing `gcd()` with the values `num1-num2` and `num2`. The output from Listing 2.20 is here:

```
reducing 24  and  10
reducing 14  and  10
switching  4  and  10
reducing 10  and  4
reducing 6  and  4
switching  2  and  4
GCD of 24 and 10 = 2
```

2.25.4 Calculating the LCM of Two Numbers

The lowest common multiple (LCM) of two positive integers is the smallest integer that is a multiple of those two integers. Some values are shown here:

```
lcm(6,2)   = 2
lcm(10,4)  = 20
lcm(24,16) = 48
```

In general, if `x` and `y` are two positive integers, you can calculate their LCM as follows:

```
lcm(x,y) = x/gcd(x,y)*y/gcd(x,y)
```

Listing 2.21 uses the `gcd()` function that is defined in the previous section in order to calculate the LCM of two positive integers.

Listing 2.21: lcm.py

```
def gcd(num1, num2):
  if(num1 % num2 == 0):
    return num2
  elif (num1 < num2):
   #print("switching ", num1, " and ", num2)
    return gcd(num2, num1)
  else:
   #print("reducing", num1, " and ", num2)
    return gcd(num1-num2, num2)

def lcm(num1, num2):
  gcd1 = gcd(num1, num2)
  lcm1 = num1*num2/gcd1
  return lcm1

result = lcm(24, 10)
print("The LCM of", 24, "and", 10, "=", result)
```

Listing 2.21 defines the function `gcd()` that was discussed in the previous section, followed by the function lcm that takes the parameters `num1` and `num2`. The first line in `lcm()` computes gcd1, which is the `gcd()` of `num1` and `num2`. The second line in `lcm()` computes `lcm1`, which is `num1` divided by three values. The third line in `lcm()` returns the value of `lcm1`. The output of Listing 2.21 is here:

```
The LCM of 24 and 10 = 120
```

2.26 Summary

This chapter showed you how to use condition logic, such as if/elif statement. You also learned how to work with loops in Python, including for loops and while loops. You learned how to compute various values, such as the GCD (greatest common divisor) and LCM (lowest common multiple) of a pair of numbers, and also how to determine whether or not a positive number is prime.

CHAPTER 3

PYTHON COLLECTIONS

- Working with Lists
- Sorting Lists of Numbers and Strings
- Expressions in Lists
- Concatenating a List of Words
- The BubbleSort in Python
- The Python `range()` Function
- Arrays and the `append()` Function
- Working with Lists and the `split()` Function
- Counting Words in a List
- Iterating through Pairs of Lists
- Other List-Related Functions
- Using a List as a Stack and a Queue
- Working with Vectors
- Working with Matrices
- The NumPy Library for Matrices
- Queues
- Tuples (Immutable Lists)
- Sets
- Dictionaries
- Dictionary Functions and Methods

- Dictionary Formatting
- Ordered Dictionaries
- Other Sequence Types in Python
- Mutable and Immutable Types in Python
- The `type()` Function
- Summary

In Chapters 1 and 2, you learned how to work with numbers and strings, as well as control structures in Python. This chapter discusses Python collections, such as lists (or arrays), sets, tuples, and dictionaries. You will see many short code blocks that will help you rapidly learn how to work with these data structures in Python. After you have finished reading this chapter, you will be in a better position to create more complex Python modules using one or more of these data structures.

The first part of this chapter discusses Python lists and shows you code samples that illustrate various methods that are available for manipulating lists. The second part of this chapter discusses Python sets and how they differ from Python lists.

The third part of this chapter discusses Python tuples, and the final part of this chapter discusses Python dictionaries.

3.1 Working with Lists

Python supports a list data type, along with a rich set of list-related functions. Since lists are not typed, you can create a list of different data types, as well as multidimensional lists. The next several sections show you how to manipulate list structures in Python.

3.1.1 Lists and Basic Operations

A Python list consists of comma-separated values enclosed in a pair of square brackets. The following examples illustrate the syntax for defining a list in Python, and also how to perform various operations on a Python list:

```
>>> list = [1, 2, 3, 4, 5]
>>> list
[1, 2, 3, 4, 5]
>>> list[2]
```

```
3
>>> list2 = list + [1, 2, 3, 4, 5]
>>> list2
[1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
>>> list2.append(6)
>>> list2
[1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 6]
>>> len(list)
5
>>> x = ['a', 'b', 'c']
>>> y = [1, 2, 3]
>>> z = [x, y]
>>> z[0]
['a', 'b', 'c']
>>> len(x)
3
```

You can assign multiple variables to a list, provided that the number and type of the variables match the structure. Here is an example:

```
>>> point = [7,8]
>>> x,y = point
>>> x
7
>>> y
8
```

The following example shows you how to assign values to variables from a more complex data structure:

```
>>> line = ['a', 10, 20, (2020,10,31)]
>>> x1,x2,x3,date1 = line
>>> x1
'a'
>>> x2
10
>>> x3
20
>>> date1
(2020, 10, 31)
```

If you want to access the year/month/date components of the `date1` element in the preceding code block, you can do so with the following code block:

```
>>> line = [‘a’, 10, 20, (2020,10,31)]
>>> x1,x2,x3,(year,month,day) = line
>>> x1
'a'
>>> x2
10
>>> x3
20
>>> year
2020
>>> month
10
>>> day
31
```

If the number and/or structure of the variables do not match the data, an error message is displayed, as shown here:

```
>>> point = (1,2)
>>> x,y,z = point
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: need more than 2 values to unpack
```

If the number of variables that you specify is less than the number of data items, you will see an error message, as shown here:

```
>>> line = ['a', 10, 20, (2014,01,31)]
>>> x1,x2 = line
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: too many values to unpack
```

3.1.2 Reversing and Sorting a List

The Python `reverse()` method reverses the contents of a list, as shown here:

```
>>> a = [4, 1, 2, 3]
>>> a.reverse()
[3, 2, 1, 4]
```

The Python `sort()` method sorts a list:

```
>>> a = [4, 1, 2, 3]
>>> a.sort()
```

```
[1, 2, 3, 4]
```

You can sort a list and then reverse its contents, as shown here:

```
>>> a = [4, 1, 2, 3]
>>> a.reverse(a.sort())
[4, 3, 2, 1]
```

Another way to reverse a list:

```
>>> L = [0,10,20,40]
>>> L[::-1]
[40, 20, 10, 0]
```

Keep in mind is that `reversed(array)` is an iterable and not a list. However, you can convert the reversed array to a list with this code snippet:

```
list(reversed(array)) or L[::-1]
```

Listing 3.1 contains a `while` loop whose logic is the opposite of the listing in the previous section: if `num` is divisible by multiple numbers (each of which is strictly less than `num`), then `num` is not prime.

Listing 3.1: Uppercase1.py

```
list1 = ['a', 'list', 'of', 'words']
list2 = [s.upper() for s in list1]
list3 = [s for s in list1 if len(s) <=2 ]
list4 = [s for s in list1 if 'w' in s ]

print('list1:',list1)
print('list2:',list2)
print('list3:',list3)
print('list4:',list4)
```

The output from launching the code in Listing 3.1 is here:

```
list1: ['a', 'list', 'of', 'words']
list2: ['A', 'LIST', 'OF', 'WORDS']
list3: ['a', 'of']
list4: ['words']
```

3.1.3 Lists and Arithmetic Operations

The minimum value of a list of numbers is the first number of in the sorted list of numbers. If you reverse the sorted list, the first number is the maximum value. There are several ways to reverse a list, starting with the technique shown in the following code:

```
x = [3,1,2,4]
maxList = x.sort()
minList = x.sort(x.reverse())

min1 = min(x)
max1 = max(x)
print min1
print max1
```

The output of the preceding code block is here:

```
1
4
```

A second (and better) way to sort a list is shown here:

```
minList = x.sort(reverse=True)
```

A third way to sort a list involves the built-in functional version of the `sort()` method, as shown here:

```
sorted(x, reverse=True)
```

The preceding code snippet is useful when you do not want to modify the original order of the list or you want to compose multiple list operations on a single line.

3.1.4 Lists and Filter-Related Operations

Python enables you to filter a list (also called list comprehension) as shown here:

```
mylist = [1, -2, 3, -5, 6, -7, 8]
pos = [n for n in mylist if n > 0]
neg = [n for n in mylist if n < 0]

print pos
print neg
```

You can also specify `if/else` logic in a filter, as shown here:

```
mylist = [1, -2, 3, -5, 6, -7, 8]
negativeList = [n if n < 0 else 0 for n in mylist]
positiveList = [n if n > 0 else 0 for n in mylist]

print positiveList
print negativeList
```

The output of the preceding code block is here:

```
[1, 3, 6, 8]
[-2, -5, -7]
[1, 0, 3, 0, 6, 0, 8]
[0, -2, 0, -5, 0, -7, 0]
```

3.2 Sorting Lists of Numbers and Strings

Listing 3.2 displays the contents of the Python script `Sorted1.py` that determines whether or not two lists are sorted.

Listing 3.2: Sorted1.py

```
list1 = [1,2,3,4,5]
list2 = [2,1,3,4,5]

sort1 = sorted(list1)
sort2 = sorted(list2)

if(list1 == sort1):
  print(list1,'is sorted')
else:
  print(list1,'is not sorted')

if(list2 == sort2):
  print(list2,'is sorted')
else:
  print(list2,'is not sorted')
```

Listing 3.2 initializes the lists `list1` and `list2`, and the sorted lists `sort1` and `sort2` based on the lists `list1` and `list2`, respectively. If `list1` equals `sort1` then `list1` is already sorted; similarly, if `list2` equals `sort2` then `list2` is already sorted.

The output from Listing 3.2 is here:

```
[1, 2, 3, 4, 5] is sorted
[2, 1, 3, 4, 5] is not sorted
```

Note that if you sort a list of character strings the output is case sensitive, and that uppercase letters appear before lowercase letters. This is due to the fact that the collating sequence for ASCII places uppercase letter (decimal `65` through decimal `91`) before lowercase letters (decimal `97` through decimal `127`). The following example provides an illustration:

```
>>> list1 = ['a', 'A', 'b', 'B', 'Z']
>>> print sorted(list1)
```

```
['A', 'B', 'Z', 'a', 'b']
```

You can also specify the reverse option so that the list is sorted in reverse order:

```
>>> list1 = ['a', 'A', 'b', 'B', 'Z']
>>> print sorted(list1, reverse=True)
['b', 'a', 'Z', 'B', 'A']
```

You can even sort a list based on the length of the items in the list:

```
>>> list1 = ['a', 'AA', 'bbb', 'BBBBB', 'ZZZZZZZ']
>>> print sorted(list1, key=len)
['a', 'AA', 'bbb', 'BBBBB', 'ZZZZZZZ']
>>> print sorted(list1, key=len, reverse=True)
['ZZZZZZZ', 'BBBBB', 'bbb', 'AA', 'a']
```

You can specify `str.lower` if you want treat uppercase letters as though they are lowercase letters during the sorting operation, as shown here:

```
>>> print sorted(list1, key=str.lower)
['a', 'AA', 'bbb', 'BBBBB', 'ZZZZZZZ']
```

3.3 Expressions in Lists

The following construct is similar to a `for` loop but without the colon ":" character that appears at the end of a loop construct. Consider the following example:

```
nums = [1, 2, 3, 4]
cubes = [ n*n*n for n in nums ]

print 'nums: ',nums
print 'cubes:',cubes
The output from the preceding code block is here:
nums:  [1, 2, 3, 4]
cubes: [1, 8, 27, 64]
```

3.4 Concatenating a List of Words

Python provides the `join()` method for concatenating text strings, as shown here:

```
>>> parts = ['Is', 'SF', 'In', 'California?']
>>> ' '.join(parts)
'Is SF In California?'
>>> ','.join(parts)
'Is,SF,In,California?'
>>> ''.join(parts)

'IsSFInCalifornia?'
```

There are several ways to concatenate a set of strings and then print the result. The following is the most inefficient way to do so:

```
print "This" + " is" + " a" + " sentence"
```

Either of the following is preferred:

```
print "%s %s %s %s" % ("This", "is", "a", "sentence")
print " ".join(["This","is","a","sentence"])
```

3.5 The BubbleSort in Python

The previous sections contain examples that illustrate how to sort a list of numbers using the `sort()` function. However, sometimes you need to implement different types of sorts in Python. Listing 3.3 displays the contents of `BubbleSort.py` that illustrates how to implement the bubble sort in Python.

Listing 3.3: BubbleSort.py

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

print("Initial list:",list1)

for i in range(0,len(list1)-1):
  for j in range(i+1,len(list1)):
    if(list1[i] > list1[j]):
      temp = list1[i]
      list1[i] = list1[j]
      list1[j] = temp

print("Sorted list: ",list1)
```

The output from Listing 3.3 is here:

```
Initial list: [1, 5, 3, 4]
Sorted list:  [1, 3, 4, 5]
```

3.6 The Python `range()` Function

In this section you will learn about the Python `range()` function that you can use to iterate through a list, as shown here:

```
>>> for i in range(0,5):
...   print i
...
0
1
2
3
4
```

You can use a `for` loop to iterate through a list of strings, as shown here:

```
>>> x
['a', 'b', 'c']
>>> for w in x:
...   print w
...
a
b
c
```

You can use a `for` loop to iterate through a list of strings and provide additional details, as shown here:

```
>>> x
['a', 'b', 'c']
>>> for w in x:
...   print len(w), w
...
1 a
1 b
1 c
```

The preceding output displays the length of each word in the list `x`, followed by the word itself.

3.6.1 Counting Digits, Uppercase, and Lowercase Letters

Listing 3.4 displays the contents of the Python script `CountCharTypes.py` that counts the occurrences of digits and letters in a string.

Listing 3.4: Counter1.py

```
str1 = "abc4234AFde"
```

```
digitCount = 0
alphaCount = 0
upperCount = 0
lowerCount = 0

for i in range(0,len(str1)):
  char = str1[i]
  if(char.isdigit()):
   #print("this is a digit:",char)
    digitCount += 1
    alphaCount += 1
  elif(char.isalpha()):
   #print("this is alphabetic:",char)
    alphaCount  += 1
    if(char.upper() == char):
      upperCount  += 1
    else:
      lowerCount  += 1

print('Original String:   ',str1)
print('Number of digits:  ',digitCount)
print('Total alphanumeric:',alphaCount)
print('Upper Case Count:  ',upperCount)
print('Lower Case Count:  ',lowerCount)
```

Listing 3.4 initializes counter-related variables, followed by a loop (with loop variable `i`) that iterates from `0` to the length of the string `str1`. The string variable `char` is initialized with the letter at index `i` of the string `str1`. The next portion of the loop uses conditional logic to determine whether `char` is a digit or an alphabetic character; in the latter case, the code checks whether or not the character is uppercase or lowercase. In all cases, the values of the appropriate counter-related variables are incremented.

The output of Listing 3.4 is here:

```
Original String:    abc4234AFde
Number of digits:   4
Total alphanumeric: 11
Upper Case Count:   2
Lower Case Count:   5
```

3.7 Arrays and the `append()` Function

Although Python does have an array type (`import array`), which is essentially a heterogeneous list, the array type has no advantages over the list type other than a slight saving in memory use. You can also define heterogeneous arrays:

```
a = [10, ‘hello’, [5, ‘77’]]
```

You can append a new element to an element inside a list:

```
>>> a = [10, 'hello', [5, '77']]
>>> a[2].append('abc')
>>> a
[10, 'hello', [5, '77', 'abc']]
```

You can assign simple variables to the elements of a list, as shown here:

```
myList = [ 'a', 'b', 91.1, (2014, 01, 31) ]
x1, x2, x3, x4 = myList
print 'x1:',x1
print 'x2:',x2
print 'x3:',x3
print 'x4:',x4
```

The output of the preceding code block is here:

```
x1: a
x2: b
x3: 91.1
x4: (2014, 1, 31)
```

The Python `split()` function is more convenient (especially when the number of elements is unknown or variable) than the preceding sample, and you will see examples of the `split()` function in the next section.

3.8 Working with Lists and the `split()` Function

You can use the Python `split()` function to split the words in a text string and populate a list with those words. An example is here:

```
>>> x = "this is a string"
>>> list = x.split()
>>> list
['this', 'is', 'a', 'string']
```

A simple way to print the list of words in a text string is shown here:

```
>>> x = "this is a string"
>>> for w in x.split():
...   print w
```

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

You can search for a word in a string as follows:

```
>>> x = "this is a string"
>>> for w in x.split():
...   if(w == 'this'):
...     print "x contains this"
...
x contains this
...
```

3.9 Counting Words in a List

Python provides the `Counter` class that enables you to count the words in a list. Listing 3.5 displays the contents of `CountWord2.py` that displays the top three words with greatest frequency.

Listing 3.5: CountWord2.py

```
from collections import Counter

mywords = ['a', 'b', 'a', 'b', 'c', 'a', 'd', 'e',
'f', 'b']

word_counts = Counter(mywords)
topThree = word_counts.most_common(3)
print(topThree)
```

Listing 3.5 initializes the variable `mywords` with a set of characters and then initializes the variable `word_counts` by passing `mywords` as an argument to `Counter`. The variable `topThree` is an array containing the three most common characters (and their frequency) that appear in `mywords`. The output from Listing 3.5 is here:

```
[('a', 3), ('b', 3), ('c', 1)]
```

3.10 Iterating through Pairs of Lists

Python supports operations on pairs of lists, which means that you can perform vector-like operations. The following snippet multiplies every list element by 3:

```
>>> list1 = [1, 2, 3]
>>> [3*x for x in list1]
[3, 6, 9]
```

Create a new list with pairs of elements consisting of the original element and the original element multiplied by 3:

```
>>> list1 = [1, 2, 3]
>>> [[x, 3*x] for x in list1]
[[1, 3], [2, 6], [3, 9]]
```

Compute the product of every pair of numbers from two lists:

```
>>> list1 = [1, 2, 3]
>>> list2 = [5, 6, 7]
>>> [a*b for a in list1 for b in list2]
[5, 6, 7, 10, 12, 14, 15, 18, 21]
```

Calculate the sum of every pair of numbers from two lists:

```
>>> list1 = [1, 2, 3]
>>> list2 = [5, 6, 7]
>>> [a+b for a in list1 for b in list2]
[6, 7, 8, 7, 8, 9, 8, 9, 10]
```

Calculate the pair-wise product of two lists:

```
>>> [list1[i]*list2[i] for i in range(len(list1))]
[8, 12, -54]
```

3.11 Other List-Related Functions

Python provides additional functions that you can use with lists, such as `append()`, `insert()`, `delete()`, `pop()`, and `extend()`. Python also supports the functions `index()`, `count()`, `sort()`, and `reverse()`. Examples of these functions are illustrated in the following code block.

Define a Python list (notice that duplicates are allowed):

```
>>> a = [1, 2, 3, 2, 4, 2, 5]
```

Display the number of occurrences of 1 and 2:

```
>>> print a.count(1), a.count(2)
1 3
```

Insert -8 in position 3:

```
>>> a.insert(3,-8)
>>> a
[1, 2, 3, -8, 2, 4, 2, 5]
```

Remove occurrences of 3:

```
>>> a.remove(3)
>>> a
[1, 2, -8, 2, 4, 2, 5]
```

Remove occurrences of 1:

```
>>> a.remove(1)
>>> a
[2, -8, 2, 4, 2, 5]
```

Append 19 to the list:

```
>>> a.append(19)
>>> a
[2, -8, 2, 4, 2, 5, 19]
```

Print the index of 19 in the list:

```
>>> a.index(19)
6
```

Reverse the list:

```
>>> a.reverse()
>>> a
[19, 5, 2, 4, 2, -8, 2]
```

Sort the list:

```
>>> a.sort()
>>> a
[-8, 2, 2, 2, 4, 5, 19]
```

Extend list a with list b:

```
>>> b = [100,200,300]
>>> a.extend(b)
>>> a
[-8, 2, 2, 2, 4, 5, 19, 100, 200, 300]
```

Remove the first occurrence of 2:

```
>>> a.pop(2)
2
>>> a
```

```
[-8, 2, 2, 4, 5, 19, 100, 200, 300]
```

Remove the last item of the list:

```
>>> a.pop()
300
>>> a
[-8, 2, 2, 4, 5, 19, 100, 200]
```

Now that you understand how to use list-related operations, the next section shows you how to use a Python list as a stack.

3.12 Using a List as a Stack and a Queue

A stack is a LIFO ("Last In First Out") data structure with `push()` and `pop()` functions for adding and removing elements, respectively. The most recently added element in a stack is in the top position, and therefore the first element that can be removed from the stack.

The following code block illustrates how to create a stack and also remove and append items from a stack in Python. Create a Python list (which we'll use as a stack):

```
>>> s = [1,2,3,4]
```

Append 5 to the stack:

```
>>> s.append(5)
>>> s
[1, 2, 3, 4, 5]
```

Remove the last element from the stack:

```
>>> s.pop()
5
>>> s
[1, 2, 3, 4]
```

A queue is a FIFO ("First In First Out") data structure with `insert()` and `pop()` functions for inserting and removing elements, respectively. The most recently added element in a queue is in the top position, and therefore the last element that can be removed from the queue.

The following code block illustrates how to create a queue and also insert and append items to a queue in Python.

Create a Python list (which we'll use as a queue):

```
>>> q = [1,2,3,4]
```

Insert 5 at the beginning of the queue:

```
>>> q.insert(0,5)
>>> q
[5, 1, 2, 3, 4]
```

Remove the last element from the queue:

```
>>> q.pop(0)
1
>>> q
[5, 2, 3, 4]
```

The preceding code uses `q.insert(0, 5)` to insert in the beginning and `q.pop()` to remove from the end. However, keep in mind that the `insert()` operation is slow in Python: insert at `0` requires copying all the elements in underlying array down one space. Therefore, use `collections.deque` with `coll.appendleft()` and `coll.pop()`, where `coll` is an instance of the `Collection` class.

The next section shows you how to work with vectors in Python.

3.13 Working with Vectors

A vector is a one-dimensional array of values, and you can perform vector-based operations, such as addition, subtraction, and inner product. Listing 3.6 displays the contents of `MyVectors.py` that illustrates how to perform vector-based operations.

Listing 3.6: MyVectors.py

```
v1 = [1,2,3]
v2 = [1,2,3]
v3 = [5,5,5]

s1 = [0,0,0]
d1 = [0,0,0]
p1 = 0

print("Initial Vectors"
print('v1:',v1)
print('v2:',v2)
print('v3:',v3)

for i in range(len(v1)):
    d1[i] = v3[i] - v2[i]
    s1[i] = v3[i] + v2[i]
    p1    = v3[i] * v2[i] + p1

print("After operations")
print('d1:',d1)
print('s1:',s1)
print('p1:',p1)
```

Listing 3.6 starts with the definition of three lists in Python, each of which represents a vector. The lists `d1` and `s1` represent the difference of `v2` and the sum `v2`, respectively. The number `p1` represents the "inner product" (also called the "dot product") of `v3` and `v2`. The output from Listing 3.6 is here:

```
Initial Vectors
v1: [1, 2, 3]
v2: [1, 2, 3]
v3: [5, 5, 5]
After operations
d1: [4, 3, 2]
s1: [6, 7, 8]
p1: 30
```

3.14 Working with Matrices

A two-dimensional matrix is a two-dimensional array of values, and you can easily create such a matrix. For example, the following code block illustrates how to access different elements in a 2D matrix:

```
mm = [["a","b","c"],["d","e","f"],["g","h","i"]];
print 'mm:      ',mm
print 'mm[0]:   ',mm[0]
print 'mm[0][1]:',mm[0][1]
```

The output from the preceding code block is here:

```
mm:       [['a', 'b', 'c'], ['d', 'e', 'f'], ['g', 'h',
'i']]
mm[0]:    ['a', 'b', 'c']
mm[0][1]: b
```

Listing 3.7 displays the contents of My2DMatrix.py that illustrates how to create and populate 2 two-dimensional matrix.

Listing 3.7: My2DMatrix.py

```
rows = 3
cols = 3

my2DMatrix = [[0 for i in range(rows)] for j in
range(rows)]
print('Before:',my2DMatrix)

for row in range(rows):
  for col in range(cols):
    my2DMatrix[row][col] = row*row+col*col
print('After: ',my2DMatrix)
```

Listing 3.7 initializes the variables rows and cols and then uses them to create the `rows x cols` matrix `my2DMatrix` whose values are initially 0. The next part of Listing 3.7 contains a nested loop that initializes the element of `my2DMatrix` whose position is `(row,col)` with the value `row*row+col*col`. The last line of code in Listing 3.7 prints the contents of `my2DArray`. The output from Listing 3.7 is here:

```
Before: [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
After:  [[0, 1, 4], [1, 2, 5], [4, 5, 8]]
```

3.15 The NumPy Library for Matrices

The `NumPy` library (which you can install via `pip`) has a matrix object for manipulating matrices in Python. The following examples illustrate some of the features of `NumPy`.

Initialize a matrix `m` and then display its contents:

```
>>> import numpy as np
>>> m = np.matrix([[1,-2,3],[0,4,5],[7,8,-9]])
>>> m
matrix([[ 1, -2, 3],
        [ 0,  4,  5],
        [ 7,  8, -9]])
```

The next snippet returns the transpose of matrix `m`:

```
>>> m.T
```

```
matrix([[ 1, 0, 7],
        [-2,  4,  8],
        [ 3,  5, -9]])
```

The next snippet returns the inverse of matrix `m` (if it exists):

```
>>> m.I
matrix([[ 0.33043478, -0.02608696, 0.09565217],
        [-0.15217391,  0.13043478,  0.02173913],
        [ 0.12173913,  0.09565217, -0.0173913 ]])
```

The next snippet defines a vector `y` and then computes the product `m*v`:

```
>>> v = np.matrix([[2],[3],[4]])
>>> v
matrix([[2],[3],[4]])
>>> m * v
matrix([[ 8],[32],[ 2]])
```

The next snippet imports the `numpy.linalg` subpackage and then computes the determinant of the matrix `m`:

```
>>> import numpy.linalg
>>> numpy.linalg.det(m)
-229.99999999999983
```

The next snippet finds the eigenvalues of the matrix `m`:

```
>>> numpy.linalg.eigvals(m)
array([-13.11474312, 2.75956154, 6.35518158])
```

The next snippet finds solutions to the equation `m*x = v`:

```
>>> x = numpy.linalg.solve(m, v)
>>> x
matrix([[ 0.96521739],
        [ 0.17391304],
        [ 0.46086957]])
```

In addition to the preceding samples, the `NumPy` package provides additional functionality, which you can find by performing an Internet search for articles and tutorials.

3.16 Queues

A queue is a FIFO ("First In First Out") data structure. Thus, the oldest item in a queue is removed when a new item is added to a queue that is already full.

Earlier in the chapter you learned how to use a Python List to emulate a queue. However, there is also a queue object in Python. The following code snippets illustrate how to use a queue in Python.

```
>>> from collections import deque
>>> q = deque('',maxlen=10)
>>> for i in range(10,20):
...   q.append(i)
...
>>> print q
deque([10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
maxlen=10)
```

The next section shows you how to use tuples in Python.

3.17 Tuples (Immutable Lists)

Python supports a data type called a *tuple* that consists of comma-separated values without brackets (square brackets are for lists, round brackets are for arrays, and curly braces are for dictionaries). Various examples of Python tuples are here:

https://docs.python.org/3.6/tutorial/datastructures.html#tuples-and-sequences

The following code block illustrates how to create a tuple and create new tuples from an existing type in Python.

Define a Python tuple `t` as follows:

```
>>> t = 1,'a', 2,'hello',3
>>> t
(1, 'a', 2, 'hello', 3)
```

Display the first element of t:

```
>>> t[0]
1
```

Create a tuple `v` containing 10, 11, and `t`:

```
>>> v = 10,11,t
>>> v
(10, 11, (1, 'a', 2, 'hello', 3))
```

Try modifying an element of `t` (which is immutable):

```
>>> t[0] = 1000
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item
assignment
```

Python "deduplication" is useful because you can remove duplicates from a set and obtain a list, as shown here:

```
>>> lst = list(set(lst))
```

Note: The "in" operator on a list to search is O(n) whereas the "in" operator on set is O(1).

The next section discusses Python sets.

3.18 Sets

A Python set in Python is an unordered collection that does not contain duplicate elements. Use curly braces or the `set()` function to create sets. Set objects support set-theoretic operations such as union, intersection, and difference.

Note: `set()` is required in order to create an empty set because `{}` creates an empty dictionary.

The following code block illustrates how to work with a Python set.

Create a list of elements:

```
>>> l = ['a', 'b', 'a', 'c']
```

Create a set from the preceding list:

```
>>> s = set(l)
>>> s
set(['a', 'c', 'b'])
```

Test if an element is in the set:

```
>>> 'a' in s
True
>>> 'd' in s
False
>>>
```

Create a set from a string:

```
>>> n = set('abacad')
>>> n
```

```
set(['a', 'c', 'b', 'd'])
>>>
```

Subtract n from s:

```
>>> s - n
set([])
```

Subtract s from n:

```
>>> n - s
set(['d'])
>>>
```

The union of s and n:

```
>>> s | n
set(['a', 'c', 'b', 'd'])
```

The intersection of s and n:

```
>>> s & n
set(['a', 'c', 'b'])
```

The exclusive-or of s and n:

```
>>> s ^ n
set(['d'])
```

The next section shows you how to work with Python dictionaries.

3.19 Dictionaries

Python has a key/value structure called a "dict" that is a hash table. A Python dictionary (and hash tables in general) can retrieve the value of a key in constant time, regardless of the number of entries in the dictionary (and the same is true for sets). You can think of a set as essentially just the keys (not the values) of a `dict` implementation.

The contents of a `dict` can be written as a series of key:value pairs, as shown here:

```
dict1 = {key1:value1, key2:value2, ... }
```

The "empty dict" is just an empty pair of curly braces `{}`.

3.19.1 Creating a Dictionary

A Python dictionary (or hash table) contains of colon-separated key/value bindings inside a pair of curly braces, as shown here:

```
dict1 = {}
dict1 = {'x' : 1, 'y' : 2}
```

The preceding code snippet defines `dict1` as an empty dictionary, and then adds two key/value bindings.

3.19.2 Displaying the Contents of a Dictionary

You can display the contents of `dict1` with the following code:

```
>>> dict1 = {'x':1,'y':2}
>>> dict1
{'y': 2, 'x': 1}
>>> dict1['x']
1
>>> dict1['y']
2
>>> dict1['z']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'z'
```

Note: Key/value bindings for a `dict` and a set are not necessarily stored in the same order that you defined them.

Python dictionaries also provide the `get` method in order to retrieve key values:

```
>>> dict1.get('x')
1
>>> dict1.get('y')
2
>>> dict1.get('z')
```

As you can see, the Python `get` method returns `None` (which is displayed as an empty string) instead of an error when referencing a key that is not defined in a dictionary.

You can also use `dict` comprehensions to create dictionaries from expressions, as shown here:

```
>>> {x: x**3 for x in (1, 2, 3)}
{1: 1, 2: 8, 3: 37}
```

3.19.3 Checking for Keys in a Dictionary

You can easily check for the presence of a key in a Python dictionary as follows:

```
>>> 'x' in dict1
True
>>> 'z' in dict1
False
```

Use square brackets for finding or setting a value in a dictionary. For example, `dict['abc']` finds the value associated with the key `'abc'`. You can use strings, numbers, and tuples work as key values, and you can use any type as the value.

If you access a value that is not in the `dict`, Python throws a `KeyError`. Consequently, use the "in" operator to check if the key is in the dict. Alternatively, use `dict.get(key)` which returns the value or `None` if the key is not present. You can even use the expression `get(key, not-found-string)` to specify the value to return if a key is not found.

3.19.4 Deleting Keys from a Dictionary

Launch the Python interpreter and enter the following commands:

```
>>> MyDict = {'x' : 5,  'y' : 7}
>>> MyDict['z'] = 13
>>> MyDict
{'y': 7, 'x': 5, 'z': 13}
>>> del MyDict['x']
>>> MyDict
{'y': 7, 'z': 13}
>>> MyDict.keys()
['y', 'z']
>>> MyDict.values()
[13, 7]
>>> 'z' in MyDict
True
```

3.19.5 Iterating through a Dictionary

The following code snippet shows you how to iterate through a dictionary:

```
MyDict = {'x' : 5,  'y' : 7, 'z' : 13}

for key, value in MyDict.iteritems():
    print key, value
```

The output from the preceding code block is here:

```
y 7
x 5
```

```
z 13
```

3.19.6 Interpolating Data from a Dictionary

The % operator substitutes values from a Python dictionary into a string by name. Listing 3.8 contains an example of doing so.

Listing 3.8: InterpolateDict1.py

```
hash = {}
hash['beverage'] = 'coffee'
hash['count'] = 3

# %d for int, %s for string
s = 'Today I drank %(count)d cups of %(beverage)s' %
hash
print('s:', s)
```

The output from the preceding code block is here:

```
Today I drank 3 cups of coffee
```

3.20 Dictionary Functions and Methods

Python provides various functions and methods for a Python dictionary, such as `cmp()`, `len()`, and `str()` that compare two dictionaries, return the length of a dictionary, and display a string representation of a dictionary, respectively.

You can also manipulate the contents of a Python dictionary using the functions `clear()` to remove all elements, `copy()` to return a shall copy, get() to retrieve the value of a key, items() to display the (key,value) pairs of a dictionary, keys() to displays the keys of a dictionary, and values() to return the list of values of a dictionary.

3.21 Dictionary Formatting

The % operator works conveniently to substitute values from a `dict` into a string by name:

```
#create a dictionary
>>> h = {}
#add a key/value pair
>>> h['item'] = 'beer'
>>> h['count'] = 4
#interpolate using %d for int, %s for string
>>> s = 'I want %(count)d bottles of %(item)s' % h
>>> s
'I want 4 bottles of beer'
```

The next section shows you how to create an ordered Python dictionary.

3.22 Ordered Dictionaries

Regular Python dictionaries iterate over key/value pairs in arbitrary order. Python 2.7 introduced a new `OrderedDict` class in the collections module. The `OrderedDict` application programming interface (API) provides the same interface as regular dictionaries but iterates over keys and values in a guaranteed order depending on when a key was first inserted:

```
>>> from collections import OrderedDict
>>> d = OrderedDict([('first', 1),
...                  ('second', 2),
...                  ('third', 3)])
>>> d.items()
[('first', 1), ('second', 2), ('third', 3)]
```

If a new entry overwrites an existing entry, the original insertion position is left unchanged:

```
>>> d['second'] = 4
>>> d.items()
[('first', 1), ('second', 4), ('third', 3)]
```

Deleting an entry and reinserting it will move it to the end:

```
>>> del d['second']
>>> d['second'] = 5
>>> d.items()
[('first', 1), ('third', 3), ('second', 5)]
```

3.22.1 Sorting Dictionaries

Python enables you to support the entries in a dictionary. For example, you can modify the code in the preceding section to display the alphabetically sorted words and their associated word count.

3.22.2 Python Multidictionaries

You can define entries in a Python dictionary so that they reference lists or other types of Python structures. Listing 3.9 displays the contents of `MultiDictionary1.py` that illustrates how to define more complex dictionaries.

Listing 3.9: MultiDictionary1.py

```
from collections import defaultdict

d = {'a' : [1, 2, 3], 'b' : [4, 5]}
print 'firsts:',d

d = defaultdict(list)
d['a'].append(1)
d['a'].append(2)
d['b'].append(4)
print 'second:',d

d = defaultdict(set)
d['a'].add(1)
d['a'].add(2)
d['b'].add(4)
print 'third:',d
```

Listing 3.9 starts by defining the dictionary `d` and printing its contents. The next portion of Listing 3.9 specifies a list-oriented dictionary, and then modifies the values for the keys `a` and `b`. The final portion of Listing 3.9 specifies a set-oriented dictionary, and then modifies the values for the keys `a` and `b`, as well.

The output from Listing 3.9 is here:

```
first: {'a': [1, 2, 3], 'b': [4, 5]}
second: defaultdict(<type 'list'>, {'a': [1, 2], 'b':
[4]})
third: defaultdict(<type 'set'>, {'a': set([1, 2]), 'b':
set([4])})
```

The next section discusses other Python sequence types that have not been discussed in previous sections of this chapter.

3.23 Other Sequence Types in Python

Python supports 7 sequence types: `str`, `unicode`, `list`, `tuple`, `bytearray`, `buffer`, and `xrange`.

You can iterate through a sequence and retrieve the position index and corresponding value at the same time using the `enumerate()` function.

```
>>> for i, v in enumerate(['x', 'y', 'z']):
...     print i, v
...
0 x
1 y
2 z
```

`Bytearray` objects are created with the built-in function `bytearray()`. Although buffer objects are not directly supported by Python syntax, you can create them via the built-in `buffer()` function.

Objects of type `xrange` are created with the `xrange()` function. An `xrange` object is similar to a buffer in the sense that there is no specific syntax to create them. Moreover, `xrange` objects do not support operations such as slicing, concatenation or repetition.

At this point you have seen all the Python type that you will encounter in the remaining chapters of this book, so it makes sense to discuss mutable and immutable types in Python, which is the topic of the next section.

3.24 Mutable and Immutable Types in Python

Python represents its data as objects. Some of these objects (such as lists and dictionaries) are mutable, which means you can change their content without changing their identity. Objects such as integers, floats, strings and tuples are objects that cannot be changed. The key point to understand is the difference between changing the value versus assigning a new value to an object; you cannot change a string but you can assign it a different value. This detail can be verified by checking the `id` value of an object, as shown in Listing 3.10.

Listing 3.10: Mutability.py

```
s = "abc"
print('id #1:', id(s))
print('first char:', s[0])

try:
  s[0] = "o"
```

```
except:
  print('Cannot perform reassignment')

s = "xyz"
print('id #2:',id(s))
s += "uvw"
print('id #3:',id(s))
```

The output of Listing 3.x is here:

```
id #1: 4297972672
first char: a
Cannot perform reassignment
id #2: 4299809336
id #3: 4299777872
```

Thus, a Python type is immutable if its value cannot be changed (even though it's possible to assign a new value to such a type), otherwise a Python type is mutable. The Python immutable objects are of type `bytes`, `complex`, `float`, `int`, `str`, or `tuple`. On the other hand, dictionaries, lists, and sets are mutable. The key in a hash table must be an immutable type.

Since strings are immutable in Python, you cannot insert a string in the "middle" of a given text string unless you construct a second string using concatenation. For example, suppose you have the string:

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

and you want to create the following string:

```
"this is a longer string"
```

The following Python code block illustrates how to perform this task:

```
text1 = "this is a string"
text2 = text1[0:10] + "longer" + text1[9:]
print 'text1:',text1
print 'text2:',text2
```

The output of the preceding code block is here:

```
text1: this is a string
text2: this is a longer string
```

3.25 The `type()` Function

The `type()` primitive returns the type of any object, including Python primitives, functions, and user-defined objects. The following code sample displays the type of an integer and a string:

```
var1 = 123
var2 = 456.78
print("type var1: ",type(var1))
print("type var2: ",type(var2))
```

The output of the preceding code block is here:

```
type var1:  <type 'int'>
type var2:  <type 'float'>
```

3.26 Summary

This chapter showed you how to work with various Python data types. In particular, you learned about tuples, sets, and dictionaries. Next you learned how to work with lists and how to use list-related operations to extract sublists. You also learned how to use Python data types in order to define tree-like structures of data.

CHAPTER 4

INTRODUCTION TO NUMPY AND PANDAS

- What is NumPy?
- What Are NumPy Arrays?
- Working with Loops
- Appending Elements to Arrays (1)
- Appending Elements to Arrays (2)
- Multiply Lists and Arrays
- Doubling the Elements in a List
- Lists and Exponents
- Arrays and Exponents
- Math Operations and Arrays
- Working with "-1" Subranges with Vectors
- Working with "_1" Subranges with Arrays
- Other Useful NumPy Methods
- Arrays and Vector Operations
- NumPy and Dot Products (1)
- NumPy and Dot Products (2)
- NumPy and the "Norm" of Vectors
- NumPy and Other Operations
- NumPy and the `reshape()` Method

- Calculating the Mean and Standard Deviation
- Calculating Mean and Standard Deviation: another Example
- What is `Pandas`?
- A Labeled `Pandas` Dataframe
- `Pandas` Numeric
- `Pandas` Boolean `DataFrames`
- Transposing a `Pandas` Dataframe
- `Pandas` Dataframes and Random Numbers
- Combining `Pandas` `DataFrames` (1)
- Combining `Pandas` `DataFrames` (2)
- Data Manipulation with `Pandas` Dataframes (1)
- Data Manipulation with `Pandas` Dataframes (2)
- Data Manipulation with `Pandas` Dataframes (3)
- `Pandas` `DataFrames` and CSV Files
- `Pandas` `DataFrames` and Excel Spreadsheets (1)
- Select, Add, and Delete Columns in `DataFrames`
- `Pandas` `DataFrames` and Scatterplots
- `Pandas` `DataFrames` and Simple Statistics
- Useful One_line Commands in `Pandas`
- Summary

The first half of this chapter starts with a quick introduction to the Python `NumPy` package, followed by a quick introduction to `Pandas` and some of its useful features. The `Pandas` package for Python 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 section 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 section 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 briefly describes `Pandas` and some of its useful features. This section contains code samples that illustrate some nice features of `DataFrame`s 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 `DataFrame`s that you can create, such as numeric and Boolean `DataFrame`s. In addition, you will see examples of creating `DataFrame`s with `NumPy` functions and random numbers.

The fifth section of this chapter shows you how to manipulate the contents of `DataFrame`s with various operations. In particular, you will also see code samples that illustrate how to create `Pandas DataFrame`s from CSV (Comma Separated Values) 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.

4.1 What is `NumPy`?

`NumPy` is a Python module that provides many convenience methods and also better performance. `NumPy` provides a core library for scientific computing in Python, with performant multidimensional arrays and good vectorized math functions, along with support for linear algebra and random numbers.

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

4.1.1 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 all 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's delve into some examples that illustrate how to work with `NumPy` arrays, which is the topic of the next section.

4.2 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 4.1 displays the contents of `nparray1.py` that illustrates some operations on a `NumPy` array.

Listing 4.1: 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 4.1 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 4.1 is here:

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

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

4.3 Working with Loops

Listing 4.2 displays the contents of `loop1.py` that illustrates how to iterate through the elements of a `NumPy` array and a Python list.

Listing 4.2: 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 4.2 initializes the variable `list`, which is a Python list, and also the variable `arr1`, which is a `NumPy` array. The next portion of Listing 4.2 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 4.2 is here:

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

4.4 Appending Elements to Arrays (1)

Listing 4.3 displays the contents of `append1.py` that illustrates how to append elements to a `NumPy` array and a Python list.

Listing 4.3: 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 4.3 initializes the variable `list`, which is a Python list, and also the variable `arr1`, which is a `NumPy` array. The output from launching Listing 4.3 is here:

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

4.5 Appending Elements to Arrays (2)

Listing 4.4 displays the contents of `append2.py` that illustrates how to append elements to a `NumPy` array and a Python list.

Listing 4.4: 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 4.4 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 4.4 is here:

```
1
2
3
4
```

```
2
4
6
8
```

4.6 Multiply Lists and Arrays

Listing 4.5 displays the contents of `multiply1.py` that illustrates how to multiply elements in a Python list and a `NumPy` array.

Listing 4.5: 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 4.5 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 4.5:

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

4.7 Doubling the Elements in a List

Listing 4.6 displays the contents of `double_list1.py` that illustrates how to double the elements in a Python list.

Listing 4.6: 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 4.6 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 4.6 is here:

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

4.8 Lists and Exponents

Listing 4.7 displays the contents of `exponent_list1.py` that illustrates how to compute exponents of the elements in a Python list.

Listing 4.7: exponent_list1.py

```
import numpy as np

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

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

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

Listing 4.7 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 4.7 is here:

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

4.9 Arrays and Exponents

Listing 4.8 displays the contents of `exponent_array1.py` that illustrates how to compute exponents of the elements in a `NumPy` array.

Listing 4.8: 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 4.8 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 4.8 is here:

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

4.10 Math Operations and Arrays

Listing 4.9 displays the contents of `mathops_array1.py` that illustrates how to compute exponents of the elements in a `NumPy` array.

Listing 4.9: 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 4.9 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 4.9 is here:

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

4.11 Working with "-1" Subranges with Vectors

Listing 4.10 displays the contents of `npsubarray2.py` that illustrates how to compute exponents of the elements in a `NumPy` array.

Listing 4.10: 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 4.10 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 4.10 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]))
```

4.12 Working with "-1" Subranges with Arrays

Listing 4.11 displays the contents of `np2darray2.py` that illustrates how to compute exponents of the elements in a `NumPy` array.

Listing 4.11: 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 4.11 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 4.11 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]))
```

4.13 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.

4.14 Arrays and Vector Operations

Listing 4.12 displays the contents of `array_vector.py` that illustrates how to perform vector operations on the elements in a `NumPy` array.

Listing 4.12: 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 4.12 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 4.12 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]]))
```

4.15 NumPy and Dot Products (1)

Listing 4.13 displays the contents of `dotproduct1.py` that illustrates how to perform the dot product on the elements in a `NumPy` array.

Listing 4.13: 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 4.13 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's calculated in three different ways. The output from launching Listing 4.13 is here:

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

4.16 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 4.14 displays the contents of `dotproduct2.py` that illustrates how to calculate the dot product of two `NumPy` arrays.

Listing 4.14: 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 4.14 contains two `NumPy` arrays called `a` and `b` followed by six `print` statements that display the contents of `a` and `b`, and also their inner product that's calculated in three different ways. The output from launching Listing 4.14 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)
```

4.17 `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 4.15 displays the contents of `array_norm.py` that illustrates how to calculate the magnitude ("norm") of a `NumPy` array of numbers.

Listing 4.15: 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 4.15 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 4.15 consists of six `print` statements, each of which displays the computed values. The output from launching Listing 4.15 is here:

```
('a:       ', array([2, 3]))
('asquare:', array([4, 9]))
('asqsum: ', 13)
('anorm1: ', 3.605551275463989)
('anorm2: ', 3.605551275463989)
('anorm3: ', 3.605551275463989)
```

4.18 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 4.16 displays the contents of `otherops.py` that illustrates how to perform other operations on a `NumPy` array.

Listing 4.16: 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 4.16 contains two `NumPy` arrays called `a` and `b` followed five `print` statements that display the contents of `a` and `b`, their Hadamard product, and also their inner product that's calculated in two different ways. The output from launching Listing 4.16 is here:

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

4.19 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 4.17 displays the contents of `numpy_reshape.py` that illustrates how to use the `reshape()` method on a `NumPy` array.

Listing 4.17: 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 4.17 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 4.17 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([,
       [3],
       [4],
       [5],
       [6],
       [7]]))
(1, 6)
```

4.20 Calculating the Mean and Standard Deviation

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

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

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

The `np.linspace()` method generates a set of equally spaced numbers between a lower bound and an upper bound. The `np.mean()` and `np.std()` methods calculate the mean and standard deviation, respectively, of a set of numbers. Listing 4.18 displays the contents of `sample_mean_std.py` that illustrates how to calculate statistical values from a `NumPy` array.

Listing 4.18: 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 4.18 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 4.18 is here:

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

4.21 Calculating Mean and Standard Deviation: Another Example

The code sample in this section extends the code sample in the previous section with additional statistical values, and the code in Listing 4.19 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 4.19 displays the contents of `stat_summary.py` that illustrates how to display various statistical values from a `NumPy` array of random numbers.

Listing 4.19: 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)
print('Maximum:  %.3f' % data_max)
```

The data sample (shown in bold) in Listing 4.19 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 4.19 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
```

This concludes the portion of the chapter pertaining to `NumPy`. The second half of this chapter discusses some of the features of `Pandas`.

4.22 What is Pandas?

`Pandas` is a Python package that is compatible with other Python packages, such as `NumPy`, `Matplotlib`, and so forth. Install

`Pandas` by opening a command shell and invoking this command for Python 3.x:

```
pip3 install pandas
```

In many ways the `Pandas` package has the semantics of a spreadsheet, and it also works with `xsl, xml, html, 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` (not discussed in this chapter), this data structure provides another mechanism for managing data (search online for more details).

4.22.1 Pandas Dataframes

In simplified terms, a `Pandas DataFrame` is a two-dimensional data structure, and it's convenient to think of the data structure in terms of rows and columns. `DataFrames` can be labeled (rows as well as columns), and the columns can contain different data types.

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:

- Dataframe methods
- Dataframe statistics
- Grouping, pivoting, and reshaping
- Dealing with missing data
- Joining dataframes

4.22.2 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

4.23 A Labeled `Pandas` Dataframe

Listing 4.20 *d*isplays the contents of `Pandas_labeled_df.py` that illustrates how to define a `Pandas DataFrame` whose rows and columns are labeled.

Listing 4.20: 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 4.20 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 following output requires a single `print` statement (which simply displays the contents of `mydf`). The second portion of the output is generated by invoking the `describe()` method that is available for any `NumPy DataFrame`. The `describe()` method is very useful: you will see various statistical quantities, such as the mean, standard deviation minimum, and maximum performed column_wise (not row_wise), along with values for the 25th, 50th, and 75th percentiles. The output of Listing 4.20 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
```

4.24 Pandas Numeric DataFrames

Listing 4.21 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 4.21: 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 4.21 involves initializing the `DataFrames` `df1` and `df2`, and then defining the `DataFrame` `df3` as the sum of `df1` and `df2`. The output from Listing 4.21 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
```

4.25 Pandas Boolean DataFrames

Pandas supports Boolean operations on `DataFrames`, such as the logical or, the logical and, and the logical negation of a pair of `DataFrames`. Listing 4.22 displays the contents of `pandas_boolean_df.py` that illustrates how to define a `Pandas DataFrame` whose rows and columns are Boolean values.

Listing 4.22: pandas_boolean_df.py

```
import pandas as pd
```

```
df1 = pd.DataFrame({'a' : [1, 0, 1], 'b' : [0, 1, 1] },
dtype=bool)
df2 = pd.DataFrame({'a' : [0, 1, 1], 'b' : [1, 1, 0] },
dtype=bool)

print("df1 & df2:")
print(df1 & df2)

print("df1 | df2:")
print(df1 | df2)

print("df1 ^ df2:")
print(df1 ^ df2)
```

Listing 4.22 initializes the `DataFrames` `df1` and `df2`, and then computes `df1 & df2`, df1 `| df2`, `df1 ^ df2`, which represent the logical `AND`, the logical `OR`, and the logical negation, respectively, of `df1` and `df2`. The output from launching the code in Listing 4.22 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
```

4.25.1 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
```

4.26 Pandas Dataframes and Random Numbers

Listing 4.23 displays the contents of `pandas_random_df.py` that illustrates how to create a `Pandas DataFrame` with random numbers.

Listing 4.23: pandas_random_df.py

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randint(1, 5, size=(5, 2)),
columns=['a','b'])
df = df.append(df.agg(['sum', 'mean']))

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

Listing 4.23 defines the `Pandas DataFrame df` that consists of 5 rows and 2 columns of random integers between 1 and 5. Notice that the columns of `df` are labeled "a" and "b." In addition, the next code snippet appends two rows consisting of the sum and the mean of the numbers in both columns. The output of Listing 4.23 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
```

4.27 Combining `Pandas DataFrames` (1)

Listing 4.24 displays the contents of `Pandas_combine_df.py` that illustrates how to combine `Pandas DataFrames`.

Listing 4.24: pandas_combine_df.py

```
import pandas as pd
import numpy as np

df = pd.DataFrame({'foo1' : np.random.randn(5),
                   'foo2' : np.random.randn(5)})

print("contents of df:")
print(df)

print("contents of foo1:")
print(df.foo1)

print("contents of foo2:")
print(df.foo2)
```

Listing 4.24 defines the `Pandas DataFrame df` that consists of 5 rows and 2 columns (labeled "`foo1`" and "`foo2`") of random real numbers between 0 and 5. The next portion of Listing 4.5 displays the contents of `df` and `foo1`. The output of Listing 4.24 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
```

4.28 Combining `Pandas` `DataFrames` (2)

`Pandas` supports the "concat" method in `DataFrames` in order to concatenate `DataFrames`. Listing 4.25 displays the contents of `concat_frames.py` that illustrates how to combine two `Pandas` `DataFrames`.

Listing 4.25: concat_frames.py

```
import pandas as pd

can_weather = pd.DataFrame({
    "city": ["Vancouver","Toronto","Montreal"],
    "temperature": [72,65,50],
    "humidity": [40, 20, 25]
})

us_weather = pd.DataFrame({
    "city": ["SF","Chicago","LA"],
    "temperature": [60,40,85],
    "humidity": [30, 15, 55]
})

df = pd.concat([can_weather, us_weather])
print(df)
```

The first line in Listing 4.25 is an import statement, followed by the definition of the `Pandas` dataframes `can_weather` and `us_weather` that contain weather-related information for cities in Canada and the United States, respectively. The `Pandas` dataframe `df` is the concatenation of `can_weather` and `us_weather`. The output from Listing 4.25 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
```

4.29 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 4.26 displays the contents of `pandas_quarterly_df1.py` that illustrates how to define a `Pandas DataFrame` consisting of income-related values.

Listing 4.26: pandas_quarterly_df1.py

```
import pandas as pd

summary = {
    'Quarter': ['Q1', 'Q2', 'Q3', 'Q4'],
    'Cost':    [23500, 34000, 57000, 32000],
    'Revenue': [40000, 40000, 40000, 40000]
}

df = pd.DataFrame(summary)

print("Entire Dataset:\n",df)
print("Quarter:\n",df.Quarter)
print("Cost:\n",df.Cost)
print("Revenue:\n",df.Revenue)
```

Listing 4.26 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 4.26 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
```

4.30 Data Manipulation with `Pandas DataFrames` (2)

In this section, let's suppose that we have a two-person company that keeps track of income and expenses on a quarterly basis, and we want to calculate the profit/loss for each quarter, and also the overall profit/loss.

Listing 4.27 displays the contents of `pandas_quarterly_df1.py` that illustrates how to define a `Pandas DataFrame` consisting of income-related values.

Listing 4.27: pandas_quarterly_df2.py

```
import pandas as pd
```

```
summary = {
    'Quarter': ['Q1', 'Q2', 'Q3', 'Q4'],
    'Cost':    [_23500, _34000, _57000, _32000],
    'Revenue': [40000, 40000, 40000, 40000]
}

df = pd.DataFrame(summary)
print("First Dataset:\n",df)

df['Total'] = df.sum(axis=1)
print("Second Dataset:\n",df)
```

Listing 4.27 defines the variable `summary` that contains quarterly information about cost and revenue for our two-person company. The variable `df` is a `Pandas DataFrame` based on the data in the `summary` variable. The three `print` statements display the quarters, the cost per quarter, and the revenue per quarter. The output from Listing 4.27 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
```

4.31 Data Manipulation with `Pandas Dataframes` (3)

Let's start with the same assumption as the previous section: we have a two-person company that keeps track of income and expenses on a quarterly basis, and we want to calculate the profit/loss for each quarter, and also the overall profit/loss. In addition, we want to compute column totals and row totals.

Listing 4.28 displays the contents of `pandas_quarterly_df1.py` that illustrates how to define a `Pandas DataFrame` consisting of income-related values.

Listing 4.28: pandas_quarterly_df3.py

```
import pandas as pd
```

```
summary = {
    'Quarter': ['Q1', 'Q2', 'Q3', 'Q4'],
    'Cost':    [_23500, _34000, _57000, _32000],
    'Revenue': [40000, 40000, 40000, 40000]
}

df = pd.DataFrame(summary)
print("First Dataset:\n",df)

df['Total'] = df.sum(axis=1)
df.loc['Sum'] = df.sum()
print("Second Dataset:\n",df)

# or df.loc['avg'] / 3
#df.loc['avg'] = df[:3].mean()
#print("Third Dataset:\n",df)
```

Listing 4.28 defines the variable `summary` that contains quarterly information about cost and revenue for our two-person company. The variable `df` is a `Pandas DataFrame` based on the data in the `summary` variable. The three `print` statements display the quarters, the cost per quarter, and the revenue per quarter. The output from Listing 4.28 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
```

4.32 Pandas DataFrames and CSV Files

The code samples in several earlier sections contain hard-coded data inside the Python scripts. However, it's also very common to read data from a CSV file. You can use the Python `csv.reader()` function, the `NumPy` `loadtxt()` function, or the Pandas function `read_csv()` function (shown in this section) to read the contents of CSV files.

Listing 4.29 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 4.29: weather_data.py

```
import pandas as pd

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

print(df)
print(df.shape)  # rows, columns
print(df.head()) # df.head(3)
print(df.tail())
print(df[1:3])
print(df.columns)
print(type(df['day']))
print(df[['day','temperature']])
print(df['temperature'].max())
```

Listing 4.29 invokes the `Pandas read_csv()` function to read the contents of the CSV file `weather_data.csv`, followed by a set of Python `print()` statements that display various portions of the CSV file. The output from Listing 4.29 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's applied to a column value.

Listing 4.30 displays the contents of the CSV file `people.csv` and Listing 4.31 displays the contents of `people_pandas.py` that illustrates how to define a `Pandas DataFrame` that reads the CSV file and manipulates the data.

Listing 4.30: 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 4.31: people_pandas.py

```
import pandas as pd

df = pd.read_csv('people.csv')
df.info()
print('fname:')
print(df['fname'])
print('____________')
print('age over 33:')
print(df['age'] > 33)
print('____________')
print('age over 33:')
myfilter = df['age'] >  33
print(df[myfilter])
```

Listing 4.31 populate the `Pandas` dataframe `df` with the contents of the CSV file people.csv. The next portion of Listing 4.12 displays the structure of `df`, followed by the first names of all the people. The next portion of Listing 4.12 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 4.31 displays a tabular list of two rows containing all the details of the people who are over 33. The output from Listing 4.31 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
```

4.33 `Pandas DataFrames` and Excel Spreadsheets (1)

Listing 4.32 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 4.32: people_xslx.py

```
import pandas as pd

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

Listing 4.32 is straightforward: the Pandas dataframe `df` is initialized with the contents of the spreadsheet `people.xlsx` (whose contents are the same as `people.csv`) via the `Pandas` function `read_excel()`. The output from Listing 4.32 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
```

4.34 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 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`.

4.35 Pandas `DataFrames` and Scatterplots

Listing 4.33 displays the contents of `pandas_scatter_df.py` that illustrates how to generate a scatterplot from a `Pandas DataFrame`.

Listing 4.33: pandas_scatter_df.py

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from pandas import read_csv
from pandas.plotting import scatter_matrix

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

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

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

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

scatter_matrix(mydf)
plt.show()
```

Listing 4.33 starts with various import statements, followed by the definition of the `NumPy` array `myarray`. Next, the variables `myarray` and `colnames` are initialized with values for the rows and columns, respectively. The next portion of Listing 4.33 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
```

4.36 Pandas DataFrames and Simple Statistics

Listing 4.34 displays the contents of `housing_stats.py` that illustrates how to gather basic statistics from data in a `Pandas DataFrame`.

Listing 4.34: housing_stats.py

```
import pandas as pd

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

minimum_bdrms = df["bedrooms"].min()
median_bdrms  = df["bedrooms"].median()
maximum_bdrms = df["bedrooms"].max()

print("minimum # of bedrooms:",minimum_bdrms)
print("median  # of bedrooms:",median_bdrms)
print("maximum # of bedrooms:",maximum_bdrms)
print("")

print("median values:",df.median().values)
print("")

prices = df["price"]
print("first 5 prices:")
print(prices.head())
print("")

median_price = df["price"].median()
print("median price:",median_price)
print("")

corr_matrix = df.corr()
print("correlation matrix:")
print(corr_matrix["price"].sort_values(ascending=False))
```

Listing 4.34 initializes the `Pandas DataFrame df` with the contents of the CSV file `Housing.csv`. The next three variables are initialized with the minimum, median, and maximum number of bedrooms, respectively, and then these values are displayed.

The next portion of Listing 4.34 initializes the variable `prices` with the contents of the prices column of the `Pandas DataFrame df`. Next, the first five rows are printed via the `prices.head()` statement, followed by the median value of the prices.

The final portion of Listing 4.34 initializes the variable `corr_matrix` with the contents of the correlation matrix for the `Pandas DataFrame df`, and then displays its contents. The output from Listing 4.34 is here:

```
Apples
10
```

4.37 Useful `One_line` Commands in Pandas

This section contains an eclectic mix of one-line commands in `Pandas` (some of which you have already seen in this chapter) that are useful to know:

Save a dataframe to a CSV file (comma separated and without indices):

```
df.to_csv("data.csv", sep=",", index=False)
```

List the column names of a `DataFrame`:

```
df.columns
```

Drop missing data from a `DataFrame`:

```
df.dropna(axis=0, how='any')
```

Replace missing data in a `DataFrame`:

```
df.replace(to_replace=None, value=None)
```

Check for NANs in a `DataFrame`:

```
pd.isnull(object)
```

Drop a feature in a `DataFrame`:

```
df.drop('feature_variable_name', axis=1)
```

Convert object type to float in a `DataFrame`:

```
pd.to_numeric(df["feature_name"], errors='coerce')
```

Convert data in a `DataFrame` to `NumpPy` 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 dataframe by 3:

```
df["height"].apply(lambda height: 3 * height)
```

OR:

```
def multiply(x):
    return x * 3
```

```
df["height"].apply(multiply)
```

Rename the fourth column of the dataframe as "height":

```
df.rename(columns = {df.columns[3]:'height'},
inplace=True)
```

Get the unique entries of the column "first" in a `DataFrame`:

```
df[""first"].unique()
```

Create a dataframe with columns "first" and "last" from an existing `DataFrame`:

```
new_df = df[["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.

4.38 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 Dataframes` from data stored in CSV files. Then 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`. Finally, you saw how to use `Jupyter`, which is a Python-based application for displaying and executing Python code in a browser.

CHAPTER 5

INTRODUCTION TO MACHINE LEARNING

- Linear Regression with Keras
- Summary

This chapter introduces numerous concepts in machine learning, such as feature selection, feature engineering, data cleaning, training sets, and test sets.

The first part of this chapter briefly discusses machine learning and the sequence of steps that are typically required *in order to prepare a dataset. These steps include feature selection or feature extraction* that can be performed using various algorithms.

The second section describes the types of data that you can encounter, issues that can arise with the data in datasets, and how to rectify them. You will also learn about the difference between *hold out* and *k-fold* when you perform the training step.

The third part of this chapter briefly discusses the basic concepts involved in linear regression. Although linear regression was developed more than 200 years ago, this technique is still one of the core techniques for solving (albeit simple) problems in statistics and machine learning. In fact, the technique known as mean squared error (MSE) for finding a best-fitting line for data points in a 2D plane (or a hyperplane for higher dimensions) is implemented in Python and `TensorFlow` in order to minimize so-called loss functions that are discussed later.

The fourth section in this chapter contains additional code samples involving linear regression tasks using standard techniques in `NumPy`. Hence, if you are comfortable with this topic, you can probably skim quickly through the first two sections of this chapter. The third section shows you how to solve linear regression using `Keras`.

One point to keep in mind is that some algorithms are mentioned without delving into details about them. For instance, the section pertaining to supervised learning contains a list of algorithms that appear later in the chapter in the section that pertains to classification algorithms. The algorithms that are displayed in bold in a list are the algorithms that are of greater interest for this book. In some cases the algorithms are discussed in greater detail in the next chapter; otherwise, you can perform an online search for additional information about the algorithms that are not discussed in detail in this book.

5.1 What is Machine Learning?

In high-level terms, machine learning is a subset of AI that can solve tasks that are infeasible or too cumbersome with more traditional programming languages. A spam filter for email is an early example of machine learning. Machine learning generally supersedes the accuracy of older algorithms.

Despite the variety of machine learning algorithms, the data is arguably more important than the selected algorithm. Many issues can arise with data, such as insufficient data, poor quality of data, incorrect data, missing data, irrelevant data, duplicate data values, and so forth. Later in this chapter you will see techniques that address many of these data-related issues.

If you are unfamiliar with machine learning terminology, a dataset is a collection of data values, which can be in the form of a CSV file or a spreadsheet. Each column is called a feature, and each row is a datapoint that contains a set of specific values for each feature. If a dataset contains information about customers, then each row pertains to a specific customer.

5.1.1 Types of Machine Learning

There are three main types of machine learning (combinations of these are also possible) that you will encounter:

- supervised learning
- unsupervised learning
- semisupervised learning

Supervised learning means that the datapoints in a dataset have a label that identifies its contents. For example, the `MNIST` dataset contains 28x28 PNG files, each of which contains a single hand-drawn digit (i.e. 0 through 9 inclusive). Every image with the digit 0 has the label 0; every image with the digit 1 has the label 1; all other images are labeled according to the digit that is displayed in those images.

As another example, the columns in the Titanic dataset are features about passengers, such as their gender, the cabin class, the price of their ticket, whether or not the passenger survived, and so forth. Each row contains information about a single passenger, including the value 1 if the passenger survived. The MNIST dataset and the Titanic dataset involve *classification* tasks: the goal is to train a model based on a training dataset and then predict the class of each row in a test dataset.

In general, the datasets for classification tasks have a small number of possible values: one of nine digits in the range of 0 through 9, one of four animals (dog, cat, horse, giraffe), one of two values (survived versus perished, purchased versus not purchased). As a rule of thumb, if the number of outcomes can be displayed in a relatively small number of values (which is subjective number) in a drop-down list, then it's probably a classification task.

In the case of a dataset that contains real estate data, each row contains information about a specific house, such as the number of bedrooms, the square feet of the house, the number of bathrooms, the price of the house, and so forth. In this dataset the price of the house is the label for each row. Notice that the range of possible prices is too large to fit reasonably well in a drop-down list. A real estate dataset involves a *regression* task: the goal is to train a model based on a training dataset and then predict the price of each house in a test dataset.

Unsupervised learning involves unlabeled data, which is typically the case for clustering algorithms (discussed later). Some important unsupervised learning algorithms that involve *clustering* are as follows:

- k-Means
- hierarchical cluster analysis (HCA)
- expectation maximization

Some important unsupervised learning algorithms that involve *dimensionality reduction* (discussed in more detail later) are as follows:

- principal component analysis (PCA)
- kernel PCA
- locally linear embedding (LLE)
- t-distributed stochastic neighbor embedding (t-SNE)

There is one more very important unsupervised task called anomaly detection. This task is relevant for fraud detection and detecting outliers (discussed later in more detail).

Semisupervised learning is a combination of supervised and unsupervised learning: some datapoints are labeled and some are unlabeled. One technique involves using the labeled data in order to classify (i.e., label) the unlabeled data, after which you can apply a classification algorithm.

5.2 Types of Machine Learning Algorithms

There are three main types of machine learning algorithms:

- regression (ex., linear regression)
- classification (ex., k-nearest-neighbor)
- clustering (ex., kMeans)

Regression is a supervised learning technique to predict numerical quantities. An example of a regression task is predicting the value of a particular stock. Note that this task is different from predicting whether the value of a particular stock will increase or decrease tomorrow (or some other future time period). Another example of a regression task involves predicting the cost of a house in a real estate dataset. Both of these tasks are examples of a regression task.

Regression algorithms in machine learning include linear regression and generalized linear regression (also called multivariate analysis in traditional statistics).

Classification is also a supervised learning technique, for predicting numeric or categorical quantities. An example of a classification task is detecting the occurrence of spam, fraud, or determining the digit in a PNG file (such as the `MNIST` dataset). In this case, the data is already labeled, so you can compare the prediction with the label that was assigned to the given PNG.

Classification algorithms in machine learning include the following list of algorithms (they are discussed in greater detail in the next chapter):

- decision trees (a single tree)
- random forests (multiple trees)
- kNN (k nearest neighbor)
- logistic regression (despite its name)
- naïve Bayes
- support vector machines (SVM)

Some machine learning algorithms (such as SVMs, random forests, and kNN) support regression as well as classification. In the case of SVMs, the scikit-learn implementation of this algorithm provides two APIs: SVC for classification and SVR for regression.

Each of the preceding algorithms involves a model that is trained on a dataset, after which the model is used to make a prediction. By contrast, a random forest consists of *multiple* independent trees (the number is specified by you), and each tree makes a prediction regarding the value of a feature. If the feature is numeric, take the mean or the mode (or perform some other calculation) in order to determine the final prediction. If the feature is categorical, use the mode (i.e., the most frequently occurring class) as the result; in the case of a tie you can select one of them in a random fashion.

Incidentally, the following link contains more information regarding the kNN algorithm for classification as well as regression:

http://saedsayad.com/k_nearest_neighbors_reg.htm

Clustering is an unsupervised learning technique for grouping similar data together. Clustering algorithms put data points in different clusters without knowing the nature of the data points. After the data has been separated into different clusters, you can use the SVM (Support Vector Machine) algorithm to perform classification.

Clustering algorithms in machine learning include the following (some of which are variations of each other):

- k-Means
- meanshift
- hierarchical cluster analysis (HCA)
- expectation maximization

Keep in mind the following points. First, the value of `k` in `k-Means` is a hyperparameter, and it's usually an odd number to avoid ties between two classes. Next, the `meanshift` algorithm is a variation of the `k-Means` algorithm that does *not* require you to specify a value for k. In fact, the `meanshift` algorithm determines the optimal number of clusters. However, this algorithm does not scale well for large datasets.

5.2.1 Machine Learning Tasks

Unless you have a dataset that has already been sanitized, you need to examine the data in a dataset to make sure that it's in a suitable condition.

The data preparation phase involves (1) examining the rows (data cleaning) to ensure that they contain valid data (which might require domain-specific knowledge), and (2) examining the columns (feature selection or feature extraction) to determine if you can retain only the most important columns.

A high-level list of the sequence of machine learning tasks (some of which might not be required) is shown here:

- obtain a dataset
- data cleaning
- feature selection
- dimensionality reduction
- algorithm selection
- train-versus-test data
- training a model
- testing a model
- fine-tuning a model
- obtain metrics for the model

First, you obviously need to obtain a dataset for your task. In the ideal scenario, this dataset already exists; otherwise, you need to cull the data from one or more data sources (e.g., a CSV file, a relational database, a no-SQL database, a Web service, and so forth).

Second, you need to perform *data cleaning*, which you can do via the following techniques:

- missing value ratio
- low variance filter
- high correlation filter

In general, data cleaning involves checking the data values in a dataset in order to resolve one or more of the following:

- Fix incorrect values.
- Resolve duplicate values.
- Resolve missing values.
- Decide what to do with outliers.

Use the missing value ratio technique if the dataset has too many missing values. In extreme cases, you might be able to drop features with a large number of missing values. Use the low variance filter technique to identify and drop features with constant values from the dataset. Use the high correlation filter technique to find highly correlated features, which increase multicollinearity in the dataset: such features can be removed from a dataset (but check with your domain expert before doing so).

Depending on your background and the nature of the dataset, you might need to work with a domain expert, which is a person who has a deep understanding of the contents of the dataset.

For example, you can use a statistical value (mean, mode, and so forth) to replace incorrect values with suitable values. Duplicate values can be handled in a similar fashion. You can replace missing numeric values with zero, the minimum, the mean, the mode, or the maximum value in a numeric column. You can replace missing categorical values with the mode of the categorical column.

If a row in a dataset contains a value that is an outlier, you have three choices:

- Delete the row.
- Keep the row.
- Replace the outlier with some other value (mean?).

When a dataset contains an outlier, you need to make a decision based on domain knowledge that is specific to the given dataset.

Suppose that a dataset contains stock-related information. As you know, there was a stock market crash in 1929, which you can view as an outlier. Such an occurrence is rare, but it can contain meaningful information. Incidentally, the source of wealth for some families in the 20th century was based on buying massive amounts of stock are very low prices during the Great Depression.

5.3 Feature Engineering, Selection, and Extraction

In addition to creating a dataset and cleaning its values, you also need to examine the features in that dataset to determine whether or not you can reduce the dimensionality (i.e., the number of columns) of the dataset. The process for doing so involves three main techniques:

- feature engineering
- feature selection
- feature extraction (aka feature projection)

Feature engineering is the process of determining a new set of features that are based on a combination of existing features in order to create a meaningful dataset for a given task. Domain expertise is often required for this process, even in cases of relatively simple datasets. Feature engineering can be tedious and expensive, and in some cases you might consider using automated feature learning. After you have created a dataset, it's a good idea to perform feature selection or feature extraction (or both) to ensure that you have a high quality dataset.

Feature selection is also called variable selection, attribute selection or variable subset selection. Feature selection involves of selecting a subset of relevant features in a dataset. In essence, feature selection involves selecting the most significant features in a dataset, which provides these advantages:

- reduced training time
- simpler models are easier to interpret
- avoidance of the curse of dimensionality
- better generalization due to a reduction in overfitting (*reduction of variance*)

Feature selection techniques are often used in domains where there are many features and comparatively few samples (or data points). Keep in mind that a low-value feature can be redundant or irrelevant, which are two different concepts. For instance, a relevant feature might be redundant when it's combined with another strongly correlated feature.

Feature selection can use three strategies: the filter strategy (e.g., information gain), the wrapper strategy (e.g., search guided by accuracy), and the embedded strategy (prediction errors are used to determine whether features are included or excluded while developing a model). One other interesting point is that feature selection can also be useful for regression as well as for classification tasks.

Feature extraction creates new features from functions that produce combinations of the original features. By contrast, feature selection involves determining a subset of the existing features.

Feature selection and feature extraction both result in *dimensionality reduction* for a given dataset, which is the topic of the next section.

5.4 Dimensionality Reduction

Dimensionality Reduction refers to algorithms that reduce the number of features in a dataset: hence the term *dimensionality reduction*. As you will see, there are many techniques available, and they involve either feature selection or feature extraction.

Algorithms that use feature selection to perform dimensionality reduction are listed here:

- backward feature elimination
- forward feature selection
- factor analysis
- independent component analysis

Algorithms that use feature extraction to perform dimensionality reduction are listed here:

- principal component analysis (PCA)
- nonnegative matrix factorization (NMF)
- kernel PCA
- graph-based kernel PCA
- linear discriminant analysis (LDA)
- generalized discriminant analysis (GDA)
- autoencoder

The following algorithms combine feature extraction and dimensionality reduction:

- principal component analysis (PCA)
- linear discriminant analysis (LDA)

- canonical correlation analysis (CCA)
- nonnegative matrix factorization (NMF)

These algorithms can be used during a preprocessing step before using clustering or some other algorithm (such as kNN) on a dataset.

One other group of algorithms involves methods based on projections, which includes t-distributed stochastic neighbor embedding (t-SNE) as well as UMAP (Uniform Manifold Approximation and Projection).

This chapter discusses PCA, and you can perform an online search to find more information about the other algorithms.

5.4.1 PCA

Principal components are new components that are linear combinations of the initial variables in a dataset. In addition, these components are uncorrelated and the most meaningful or important information is contained in these new components.

There are two advantages to PCA: (1) reduced computation time due to far fewer features, and (2) the ability to graph the components when there are at most three components. If you have four or five components, you won't be able to display them visually, but you could select subsets of three components for visualization, and perhaps gain some additional insight into the dataset.

PCA uses the variance as a measure of information: the higher the variance, the more important the component. In fact, just to jump ahead slightly: PCA determines the eigenvalues and eigenvectors of a covariance matrix (discussed later), and constructs a new matrix whose columns are eigenvectors, ordered from left-to-right based on the maximum eigenvalue in the left-most column, decreasing until the right-most eigenvector also has the smallest eigenvalue.

5.4.2 Covariance Matrix

As a reminder, the statistical quantity called the variance of a random variable `X` is defined as follows:

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

A covariance matrix `C` is an nxn matrix whose values on the main diagonal are the variance of the variables `X1`, `X2`, . . ., `Xn`. The other values of `C` are the covariance values of each pair of variables `Xi` and `Xj`.

The formula for the covariance of the variables `X` and `Y` is a generalization of the variance of a variable, and the formula is shown here:

```
covariance(X, Y) = [SUM (x – xbar)*(y-ybar)]/n
```

Notice that you can reverse the order of the product of terms (multiplication is commutative), and therefore the covariance matrix `C` is a symmetric matrix:

```
covariance(X, Y) = covariance(Y,X)
```

PCA calculates the eigenvalues and the eigenvectors of the covariance matrix A.

5.5 Working with Datasets

In addition to data cleaning, there are several other steps that you need to perform, such as selecting training data versus test data, and deciding whether to use hold out or cross-validation during the training process. More details are provided in the subsequent sections.

5.5.1 Training Data versus Test Data

After you have performed the tasks described earlier in this chapter (i.e., data cleaning and perhaps dimensionality reduction), you are ready to split the dataset into two parts. The first part is the *training set*, which is used to train a model, and the second part is the *test set*, which is used for *inferencing* (another term for making predictions). Make sure that you conform to the following guidelines for your test sets:

1. The set is large enough to yield statistically meaningful results.
2. It's representative of the dataset as a whole.
3. Never train on test data.
4. Never test on training data.

5.5.2 What is Cross-validation?

The purpose of cross-validation is to test a model with nonoverlapping test sets, and is performed in the following manner:

1. Split the data into k subsets of equal size.
2. Select one subset for testing and the others for training.
3. Repeat step 2 for the other k-1 subsets.

This process is called *k-fold cross-validation*, and the overall error estimate is the average of the error estimates. A standard method for evaluation involves ten-fold cross-validation. Extensive experiments have shown that 10 subsets is the best choice to obtain an accurate estimate. In fact, you can repeat ten-fold cross-validation ten times and compute the average of the results, which helps to reduce the variance.

The next section discusses regularization, which is an important yet optional topic if you are primarily interested in TF 2 code. If you plan to become proficient in machine learning, you will need to learn about regularization.

5.6 What is Regularization?

Regularization helps to solve overfitting problem, which occurs when a model performs well on training data but poorly on validation or test data.

Regularization solves this problem by adding a penalty term to the cost function, thereby controlling the model complexity with this penalty term. Regularization is generally useful for:

- large number of variables
- low ration of (# observations)/(# of variables
- high multicollinearity

There are two main types of regularization: L1 regularization (which is related to MAE, or the absolute value of differences) and L2 regularization (which is related to MSE, or the square of differences). In general, L2 performs better than L1 and L2 is efficient in terms of computation.

5.6.1 ML and Feature Scaling

Feature scaling standardizes the range of features of data. This step is performed during the data preprocessing step, in part because gradient descent benefits from feature scaling.

The assumption is that the data conforms to a standard normal distribution, and standardization involves subtracting the mean and divide by the standard deviation for every data point, which results in a N(0,1) normal distribution.

5.6.2 Data Normalization versus Standardization

Data normalization is a linear scaling technique. Let's assume that a dataset has the values `{X1, X2, . . . , Xn}` along with the following terms:

```
Minx = minimum of Xi values
```

```
Maxx = maximum of Xi values
```

Now calculate a set of new `Xi` values as follows:

```
Xi = (Xi - Minx)/[Maxx - Minx]
```

The new `Xi` values are now scaled so that they are between 0 and 1.

5.7 The Bias-Variance Tradeoff

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

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

Variance in machine learning is the expected value of the squared deviation from the mean. High variance can/might cause an algorithm to model the random noise in the training data, rather than the intended outputs (aka overfitting).

Adding parameters to a model increases its complexity, increases the variance, and decreases the bias. Dealing with bias and variance is dealing with underfitting and overfitting.

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

5.8 Metrics for Measuring Models

One of the most frequently used metrics is R-squared, which measures how close the data is to the fitted regression line (regression coefficient). The R-squared value is always a percentage between 0 and 100%. The value

0% indicates that the model explains none of the variability of the response data around its mean. The value 100% indicates that the model explains all the variability of the response data around its mean. In general, a higher R-squared value indicates a better model.

5.8.1 Limitations of R-Squared

Although high R-squared values are preferred, they are not necessarily always good values. Similarly, low R-squared values are not always bad. For example, an R-squared value for predicting human behavior is often less than 50%. Moreover, R-squared cannot determine whether the coefficient estimates and predictions are biased. In addition, an R-squared value does not indicate whether a regression model is adequate. Thus, it's possible to have a low R-squared value for a good model, or a high R-squared value for a poorly fitting model. Evaluate R-squared values in conjunction with residual plots, other model statistics, and subject area knowledge.

5.8.2 Confusion Matrix

In its simplest form, a confusion matrix (also called an error matrix) is a type of contingency table with two rows and two columns that contains the # of false positives, false negatives, true positives, and true negatives. The four entries in a 2x2 confusion matrix can be labeled as follows:

```
TP: True Positive
FP: False Positive
TN: True Negative
FN: False Negative
```

The diagonal values of the confusion matrix are correct, whereas the off-diagonal values are incorrect predictions. In general a lower FP value is better than a FN value. For example, an FP indicates that a healthy person was incorrectly diagnosed with a disease, whereas an FN indicates that an unhealthy person was incorrectly diagnosed as healthy.

5.8.3 Accuracy versus Precision versus Recall

A 2x2 confusion matrix has four entries that that represent the various combinations of correct and incorrect classifications. Given the definitions in the preceding section, the definitions of precision, accuracy, and recall are given by the following formulas:

```
precision = TP/(TN + FP)
accuracy  = (TP + TN)/[P + N]
```

```
recall    = TP/[TP + FN]
```

Accuracy can be an unreliable metric because it yields misleading results in unbalanced datasets. When the number of observations in different classes are significantly different, it gives equal importance to both false positive and false negative classifications. For example, declaring cancer as benign is worse than incorrectly informing patients that they are suffering from cancer. Unfortunately, accuracy won't differentiate between these two cases.

Keep in mind that the confusion matrix can be an nxn matrix and not just a 2x2 matrix. For example, if a class has 5 possible values, then the confusion matrix is a 5x5 matrix, and the numbers on the main diagonal are the *true positive* results.

5.8.4 The ROC Curve

The receiver operating characteristic (ROC) curve is a curve that plots the the true positive rate (TPR; i.e., the recall) against the false positive rate (FPR). Note that the the true negative rate (TNR) is also called the specificity.

The following link contains a Python code sample using SKLearn and the Iris dataset, and also code for plotting the ROC:

https://scikit-learn.org/stable/auto_examples/model_selection/plot_roc.html

The following link contains an assortment of Python code samples for plotting the ROC:

https://stackoverflow.com/questions/25009284/how-to-plot-roc-curve-in-python

5.9 Other Useful Statistical Terms

Machine learning relies on a number of statistical quantities in order to assess the validity of a model, some of which are listed here:

- `RSS`
- `TSS`
- `R^2`
- `F1 score`
- `p-value`

The definitions of RSS, TSS, and R^2 are shown in the following, where y^ is the y-coordinate of a point on a best-fitting line and y_ is the mean of the y-values of the points in the dataset:

```
➡ RSS = sum of squares of residuals (y - y^)**2
➡ TSS = toal sum of squares        (y - y_)**2
R^2 = 1 - RSS/TSS
```

5.9.1 What Is an F1 score?

The F1 score is a measure of the accuracy of a test, and it's defined as the harmonic mean of precision and recall. Here are the relevant formulas, where p is the precision and r is the recall:

```
p = (# of correct positive results)/(# of all positive
results)
r = (# of correct positive results)/(# of all relevant
samples)

F1-score  = 1/[((1/r) + (1/p))/2]
          = 2*[p*r]/[p+r]
```

The best value of an F1 score is 0 and the worse value is 0. Keep in mind that an F1 score tends to be used for categorical classification problems, whereas the R^2 value is typically for regression tasks (such as linear regression).

5.9.2 What Is a p-value?

The p-value is used to reject the null hypothesis if the p-value is small enough (< 0.005) which indicates a higher significance. Recall that the null hypothesis states that there is no correlation between a dependent variable (such as y) and an independent variable (such as x). The threshold value for p is typically 1% or 5%.

There is no straightforward formula for calculating p-values, which are values that are always between 0 and 1. In fact, p-values are statistical quantities to evaluate the so-called *null hypothesis*, and they are calculated by means of p-value tables or via spreadsheet/statistical software.

5.10 What is Linear Regression?

The goal of linear regression is to find the best-fitting line that represents a dataset. Keep in mind two key points. First, the best-fitting line does not necessarily pass through all (or even most of) the points in the dataset. The purpose of a best-fitting line is to minimize the vertical distance of that line from the points in dataset. Second, linear regression does not determine the best-fitting polynomial: the latter involves finding a higher-degree polynomial that passes through many of the points in a dataset.

Moreover, a dataset in the plane can contain two or more points that lie on the same *vertical* line, which is to say that those points have the same x value. However, a function *cannot* pass through such a pair of points: if two points `(x1,y1)` and `(x2,y2)` have the same x value then they must have the same y value (i.e., `y1=y2`). On the other hand, a function can have two or more points that lie on the same *horizontal* line.

Now consider a scatter plot with many points in the plane that are sort of *clustered* in an elongated cloud-like shape: a best-fitting line will probably intersect only limited number of points (in fact, a best-fitting line might not intersect *any* of the points).

One other scenario to keep in mind: suppose a dataset contains a set of points that lie on the same line. For instance, let's say the x values are in the set `{1,2,3,...,10}` and the y values are in the set `{2,4,6,...,20}`. Then the equation of the best-fitting line is `y=2*x+0`. In this scenario, all the points are *collinear*, which is to say that they lie on the same line.

5.10.1 Linear Regression versus Curve-Fitting

Suppose a dataset consists of n data points of the form (x, y), and no two of those data points have the same `x` value. Then according to a well-known result in mathematics, there is a polynomial of degree less than or equal to n-1 that passes through those n points (if you are really interested, you can find a mathematical proof of this statement in online articles). For example, a line is a polynomial of degree one and it can intersect any pair of nonvertical points in the plane. For any triple of points (that are not all on the same line) in the plane, there is a quadratic equation that passes through those points.

In addition, sometimes a lower degree polynomial is available. For instance, consider the set of 100 points in which the `x` value equals the `y` value: in this case, the line `y = x` (which is a polynomial of degree one) passes through all 100 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, which is measured by the *variance* of the points (the variance is a statistical quantity). The more collinear the points, the smaller the variance; conversely, the more spread out the points are, the larger the variance.

5.10.2 When Are Solutions Exact Values?

Although statistics-based solutions provide closed-form solutions for linear regression, neural networks provide *approximate* solutions. This is due to the fact that machine learning algorithms for linear regression involve a sequence of approximations that converges to optimal values, which means that machine learning algorithms produce estimates of the exact values. For example, the slope `m` and y-intercept `b` of a best-fitting line for a set of points a 2D plane have a closed-form solution in statistics, but they can only be approximated via machine learning algorithms (exceptions do exist, but they are rare situations).

Keep in mind that even though a closed-form solution for traditional linear regression provides an exact value for both `m` and `b`, sometimes you can only use an approximation of the exact value. For instance, suppose that the slope m of a best-fitting line equals the square root of 3 and the y-intercept b is the square root of 2. If you plan to use these values in source code, you can only work with an approximation of these two numbers. In the same scenario, a neural network computes approximations for `m` and `b`, regardless of whether or not the exact values for `m` and `b` are irrational, rational, or integer values. However, machine learning algorithms are better suited for complex, nonlinear, multi-dimensional datasets, which is beyond the capacity of linear regression.

As a simple example, suppose that the closed form solution for a linear regression problem produces integer or rational values for both `m` and `b`. Specifically, let's suppose that a closed form solution yields the values 2.0 and 1.0 for the slope and y-intercept, respectively, of a best-fitting line. The equation of the line looks like this:

```
y = 2.0 * x + 1.0
```

However, the corresponding solution from training a neural network might produce the values 2.0001 and 0.9997 for the slope m and the y-intercept `b`, respectively, as the values of `m` and `b` for a best-fitting line. Always keep this point in mind, especially when you are training a neural network.

5.10.3 What is Multivariate Analysis?

Multivariate analysis generalizes the equation of a line in the Euclidean plane to higher dimensions, and it's called a *hyperplane* instead of a line. The generalized equation has the following form:

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

In the case of 2D linear regression, you only need to find the value of the slope (m) and the y-intercept (b), whereas in multivariate analysis you need to find the values for w1, w2, . . ., wn. Note that multivariate analysis is a term from statistics, and in machine learning it's often referred to as *generalized linear regression*.

Keep in mind that most of the code samples in this book that pertain to linear regression involve 2D points in the Euclidean plane.

5.11 Other Types of Regression

Linear regression finds the best-fitting line that represents a dataset, but what happens if a line in the plane is not a good fit for the dataset? This is a relevant question when you work with datasets.

Some alternatives to linear regression include quadratic equations, cubic equations, or higher-degree polynomials. However, these alternatives involve trade-offs, as we'll discuss later.

Another possibility is a sort of hybrid approach that involves piece-wise linear functions, which comprises a set of line segments. If contiguous line segments are connected then it's a piece-wise linear continuous function; otherwise it's a piece-wise linear discontinuous function.

Thus, given a set of points in the plane, regression involves addressing the following questions:

1. What type of curve fits the data well? How do we know?

2. Does another type of curve fit the data better?

3. What does "best fit" mean?

One way to check if a line fits the data involves a visual check, but this approach does not work for data points that are higher than two dimensions. Moreover, this is a subjective decision, and some sample datasets are displayed later in this chapter. By visual inspection of a dataset, you might decide that a quadratic or cubic (or even higher degree) polynomial has the potential of being a better fit for the data. However, visual inspection is probably limited to points in a 2D plane or in three dimensions.

Let's defer the nonlinear scenario and let's make the assumption that a line would be a good fit for the data. There is a well-known technique for finding the "best-fitting" line for such a dataset that involves minimizing the mean squared error (MSE) that we'll discuss later in this chapter.

The next section provides a quick review of linear equations in the plane, along with some images that illustrate examples of linear equations.

5.12 Working with Lines in the Plane (optional)

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

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

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

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

The value of `m` is the slope of the line and the value of `b` is the y-intercept (i.e., the place where the line intersects the y-axis).

If need be, you can use a more general equation that can also represent vertical lines, as shown here:

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

However, we won't be working with vertical lines, so we'll stick with the first formula.

Figure 5.1 displays three horizontal lines whose equations (from top to bottom) are `y = 3`, `y = 0`, and `y = -3`, respectively.

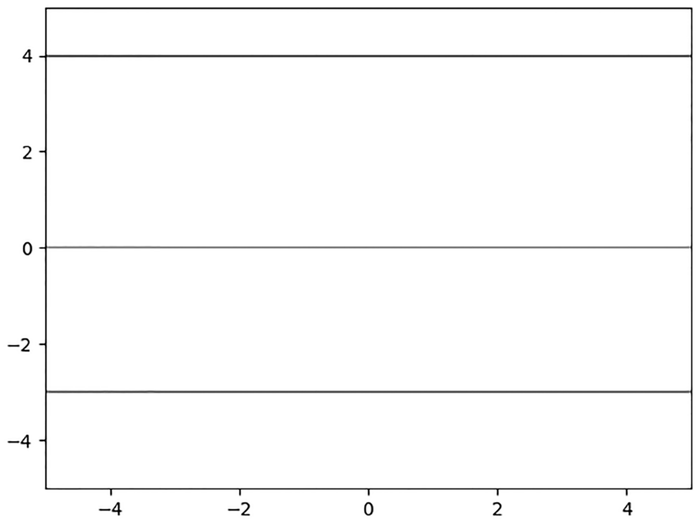

FIGURE 5.1: A graph of three horizontal line segments.

Figure 5.2 displays two slanted lines whose equations are `y = x` and `y = -x`, respectively.

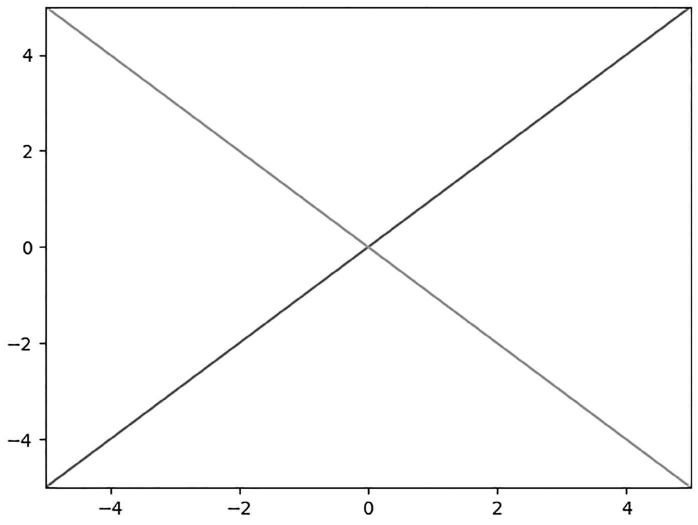

FIGURE 5.2: A graph of two diagonal line segments.

Figure 5.3 displays two slanted parallel lines whose equations are `y = 2*x` and `y = 2*x + 3`, respectively.

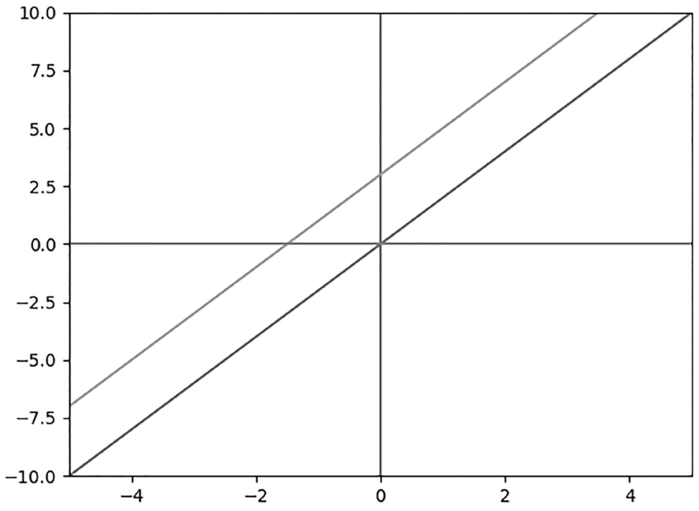

FIGURE 5.3: A graph of two slanted parallel line segments.

Figure 5.4 displays a piece-wise linear graph consisting of connected line segments.

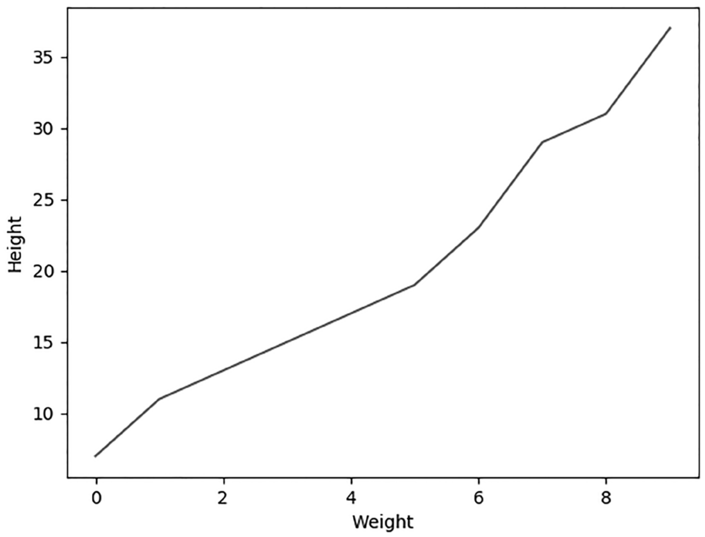

FIGURE 5.4: A piecewise linear graph of line segments.

Now let's turn our attention to generating quasi-random data using a NumPy API, and then we'll plot the data using Matplotlib.

5.13 Scatter Plots with `NumPy` and Matplotlib (1)

Listing 5.1 displays the contents of `np_plot1.py` that illustrates how to use the `NumPy randn()` API to generate a dataset and then the `scatter()` API in Matplotlib to plot the points in the dataset.

One detail to note is that all the adjacent horizontal values are equally spaced, whereas the vertical values are based on a linear equation plus a "perturbation" value. This *perturbation technique* (which is not a standard term) is used in other code samples in this chapter in order to add a slightly randomized effect when the points are plotted. The advantage of this technique is that the best-fitting values for `m` and `b` are known in advance, and therefore we do not need to guess their values.

Listing 5.1: np_plot1.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)

print("x:",x)
print("y:",y)

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

Listing 5.1 contains two `import` statements and then initializes the array variable `x` with 15 random numbers between 0 and 1.

Next, the array variable `y` is defined in two parts: the first part is a linear equation `2.5*x + 5` and the second part is a "perturbation" value that is based on a random number. Thus, the array variable `y` simulates a set of values that closely approximate a line segment.

This technique is used in code samples that simulate a line segment, and then the training portion approximates the values of `m` and `b` for the best-fitting line. Obviously we already *know* the equation of the best-fitting line: the purpose of this technique is to compare the trained values for the slope `m` and y-intercept `b` with the known values (which in this case are 2.5 and 5).

A partial output from Listing 5.1 is here:

```
x: [[-1.42736308]
 [ 0.09482338]
 [-0.45071331]
 [ 0.19536304]
 [-0.22295205]
 // values omitted for brevity
y: [[1.12530514]
 [5.05168677]
 [3.93320782]
 [5.49760999]
 [4.46994978]
 // values omitted for brevity
```

Figure 5.5 displays a scatter plot of points based on the values of x and y.

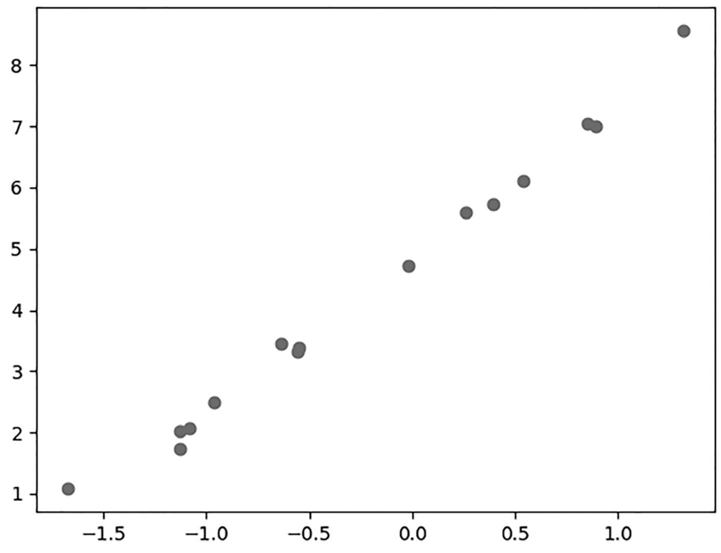

FIGURE 5.5: A scatter plot of points for a line segment.

5.13.1 Why the "Perturbation Technique" is Useful

You already saw how to use the "perturbation technique" and by way of comparison, consider a dataset with the following points that are defined in the Python array variables X and Y:

```
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]
```

If you need to find the best-fitting line for the preceding dataset, how would you guess the values for the slope m and the y-intercept b? In most cases, you probably cannot guess their values. On the other hand, the "perturbation technique" enables you to randomize the points on a line whose value for the slope m (and optionally the value for the y-intercept b) is specified in advance.

Keep in mind that the "perturbation technique" only works when you introduce small random values that do not result in different values for m and b.

5.14 Scatter Plots with NumPy and Matplotlib (2)

The code in Listing 5.1 assigned random values to the variable x, whereas a hard-coded value is assigned to the slope m. The y values are a hard-coded multiple of the x values, plus a random value that is calculated via the "perturbation technique." Hence we do not know the value of the y-intercept b.

In this section the values for trainX are based on the np.linspace() API, and the values for trainY involve the "perturbation technique" that is described in the previous section.

The code in this example simply prints the values for trainX and trainY, which correspond to data points in the Euclidean plane. Listing 5.2 displays the contents of np_plot2.py that illustrates how to simulate a linear dataset in NumpPy.

Listing 5.2: np_plot2.py

```
import numpy as np

  trainX = np.linspace(-1, 1, 11)
trainY = 4*trainX + np.random.randn(*trainX.shape)*0.5

print("trainX: ",trainX)
print("trainY: ",trainY)
```

Listing 5.6 initializes the NumPy array variable trainX via the NumPy linspace() API, followed by the array variable trainY that is defined in two parts. The first part is the linear term 4*trainX and the second part involves the "perturbation technique" that is a randomly generated number. The output from Listing 5.6 is here:

```
trainX:  [-1.  -0.8 -0.6 -0.4 -0.2  0.   0.2  0.4  0.6
0.8  1. ]
trainY:  [-3.60147459 -2.66593108 -2.26491189
-1.65121314 -0.56454605  0.22746004 0.86830728
1.60673482  2.51151543  3.59573877  3.05506056]
```

The next section contains an example that is similar to Listing 5.2, using the same "perturbation technique" to generate a set of points that approximate a quadratic equation instead of a line segment.

5.15 A Quadratic Scatterplot with `NumPy` and matplotlib

Listing 5.3 displays the contents of `np_plot_quadratic.py` that illustrates how to plot a quadratic function in the plane.

Listing 5.3: np_plot_quadratic.py

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

#see what happens with this set of values:
#x = np.linspace(-5,5,num=100)

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

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

Listing 5.3 initializes the array variable `x` with the values that are generated via the `np.linspace()` API, which in this case is a set of 100 equally spaced decimal numbers between -5 and 5. Notice the snippet `[:,None]` in the initialization of `x`, which results in an array of elements, each of which is an array consisting of a single number.

The array variable `y` is defined in two parts: the first part is a quadratic equation `-0.5 + 2.2*x +0.3*x**2` and the second part is a "perturbation" value that is based on a random number (similar to the code in Listing 5.1). Thus, the array variable `y` simulates a set of values that approximates a quadratic equation. The output from Listing 5.3 is here:

```
x:
[[-5.        ]
 [-4.8989899 ]
 [-4.7979798 ]
 [-4.6969697 ]
 [-4.5959596 ]
 [-4.49494949]
 // values omitted for brevity
 [ 4.8989899 ]
 [ 5.        ]]
```

Figure 5.6 displays a scatter plot of points based on the values of `x` and `y`, which have an approximate shape of a quadratic equation.

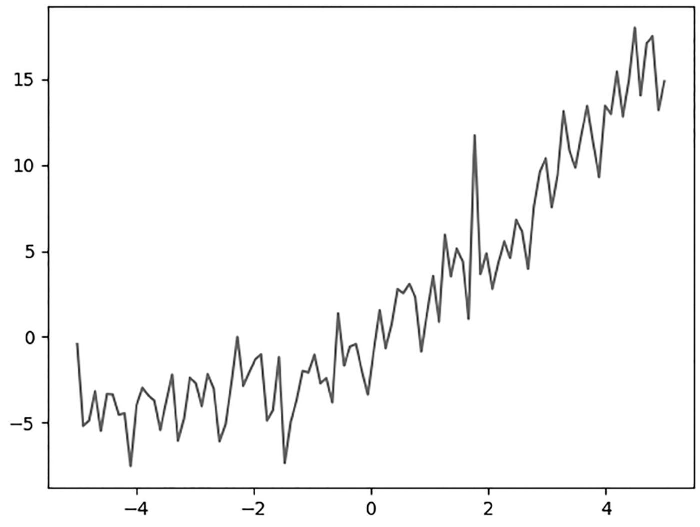

FIGURE 5.6: A scatter plot of points for a quadratic equation.

5.16 The MSE Formula

In plain English, the MSE is the sum of the squares of the difference between an actual y value and the predicted y value, divided by the number of points. Notice that the predicted y value is the y value that each point would have if that point were actually on the best-fitting line.

Although the MSE is popular for linear regression, there are other error types available, some of which are discussed briefly in the next section.

5.16.1 A List of 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), which is then divided by the number of terms.

The RMSProp optimizer utilizes the magnitude of recent gradients to normalize the gradients. Specifically, RMSProp maintain a moving average over the root mean squared (RMS) gradients, and then divides that term by the current gradient.

Although it's easier to compute the derivative of MSE, it's also true that MSE is more susceptible to outliers, whereas MAE is less susceptible to outliers. The reason is simple: a squared term can be significantly larger than the absolute value of a term. For example, if a difference term is 10, then a squared term of 100 is added to MSE, whereas only 10 is added to MAE. Similarly, if a difference term is -20, then a squared term 400 is added to MSE, whereas only 20 (which is the absolute value of -20) is added to MAE.

5.16.2 Nonlinear Least Squares

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

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

The next section contains several code samples, the first of which involves calculating the MSE manually, followed by an example that uses `NumPy` formulas to perform the calculations. Finally, we'll look at a TensorFlow example for calculating the MSE.

5.17 Calculating the MSE Manually

This section contains two line graphs, both of which contain a line that approximates a set of points in a scatter plot.

Figure 5.7 displays a line segment that approximates a scatter plot of points (some of which intersect the line segment). The MSE for the line in Figure 5.7 is computed as follows:

```
MSE = [1*1 + (-1)*(-1) + (-1)*(-1) + 1*1]/7 = 4/7
```

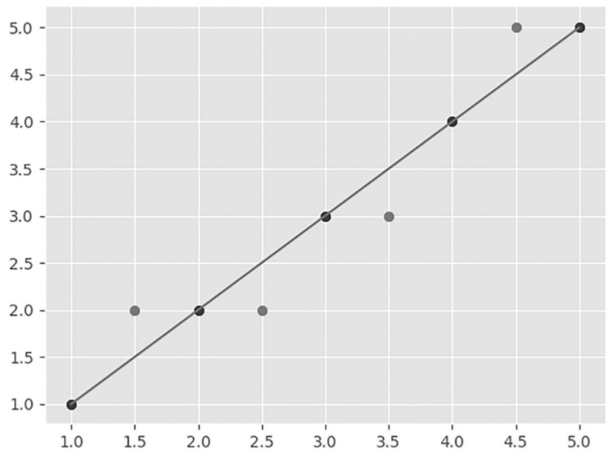

FIGURE 5.7: A line graph that approximates points of a scatter plot.

Figure 5.8 displays a set of points and a line that is a potential candidate for best-fitting line for the data. The MSE for the line in Figure 5.8 is computed as follows:

```
MSE = [(-2)*(-2) + 2*2]/7 = 8/7
```

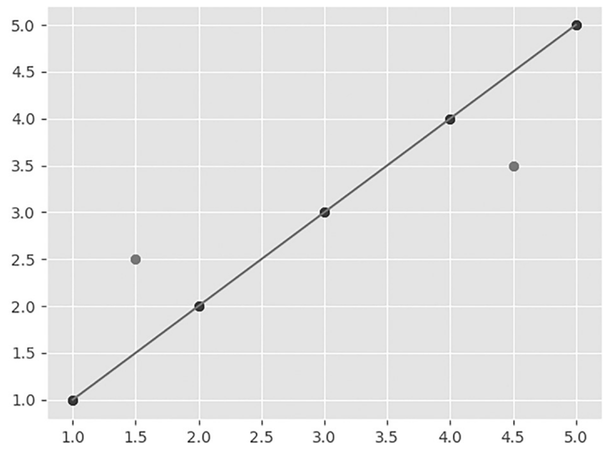

FIGURE 5.8: A line graph that approximates points of a scatter plot.

Thus, the line in Figure 5.7 has a smaller MSE than the line in Figure 5.8, which might have surprised you (or did you guess correctly?)

In these two figures we calculated the MSE easily and quickly, but in general it's significantly more difficult. For instance, if we plot 10 points in the Euclidean plane that do not closely fit a line, with individual terms that involve noninteger values, we would probably need a calculator.

A better solution involves NumPy functions, such as the `np.linspace()` API, as discussed in the next section.

5.18 Approximating Linear Data with `np.linspace()`

Listing 5.4 displays the contents of `np_linspace1.py` that illustrates how to generate some data with the `np.linspace()` API in conjunction with the "perturbation technique."

Listing 5.4: np_linspace1.py

```
import numpy as np

trainX = np.linspace(-1, 1, 6)
trainY = 3*trainX+ np.random.randn(*trainX.shape)*0.5

print("trainX: ", trainX)
print("trainY: ", trainY)
```

The purpose of this code sample is merely to generate and display a set of randomly generated numbers. Later in this chapter we will use this code as a starting point for an actual linear regression task.

Listing 5.4 starts with the definition of the array variable `trainX` that is initialized via the `np.linspace()` API. Next, the array variable `trainY` is defined via the "perturbation technique" that you have seen in previous code samples. The output from Listing 5.4 is here:

```
trainX:  [-1.  -0.6 -0.2  0.2  0.6  1. ]
trainY:  [-2.9008553  -2.26684745 -0.59516253
0.66452207  1.82669051  2.30549295]
trainX:  [-1.  -0.6 -0.2  0.2  0.6  1. ]
trainY:  [-2.9008553  -2.26684745 -0.59516253
0.66452207  1.82669051  2.30549295]
```

Now that we know how to generate `(x,y)` values for a linear equation, let's learn how to calculate the MSE, which is discussed in the next section.

The next example generates a set of data values using the `np.linspace()` method and the `np.random.randn()` method in order to introduces some randomness in the data points.

5.19 Calculating MSE with np.linspace() API

The code sample in this section differs from many of the earlier code samples in this chapter: it uses a hard-coded array of values for `X` and also for `Y` instead of the "perturbation" technique. Hence, you will *not* know the correct value for the slope and y-intercept (and you probably will not be able to guess their correct values). Listing 5.5 displays the contents of `plain_linreg1.py` that illustrates how to compute the MSE with simulated data.

Listing 5.5: 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 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 5.5 initializes the array variables `X` and `Y` with hard-coded values, and then initializes the scalar variables `W` and `b`. The next portion of Listing 5.5 contains a `for` loop that iterates 100 times. After each iteration of the loop, the variables `Y_pred`, `Loss_error`, and `cost` are calculated. Next, the values for `dw` and `db` are calculated, based on the sum of the terms in the array `Y_pred-Y`, and the inner product of `Y_pred-y` and `X`, respectively.

Notice how `W` and b are updated: their values are decremented by the term `0.01*dw` and `0.01*db`, respectively. This calculation ought to look somewhat familiar: the code is programmatically calculating an approximate value of the gradient for `W` and `b`, both of which are multiplied by the learning rate (the hard-coded value 0.01), and the resulting term is decremented from the current values of `W` and `b` in order to produce a new approximation for `W` and b. Although this technique is very simple, it does calculate reasonable values for `W` and `b`.

The final block of code in Listing 5.5 displays the intermediate approximations for `W` and `b`, along with a plot of the cost (vertical axis) versus the number of iterations (horizontal axis). The output from Listing 5.5 is here:

```
Cost at 10 iteration =  0.04114630674619492
Cost at 20 iteration =  0.026706242729839392
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 5.9 displays a scatter plot of points generated by the code in Listing 5.5.

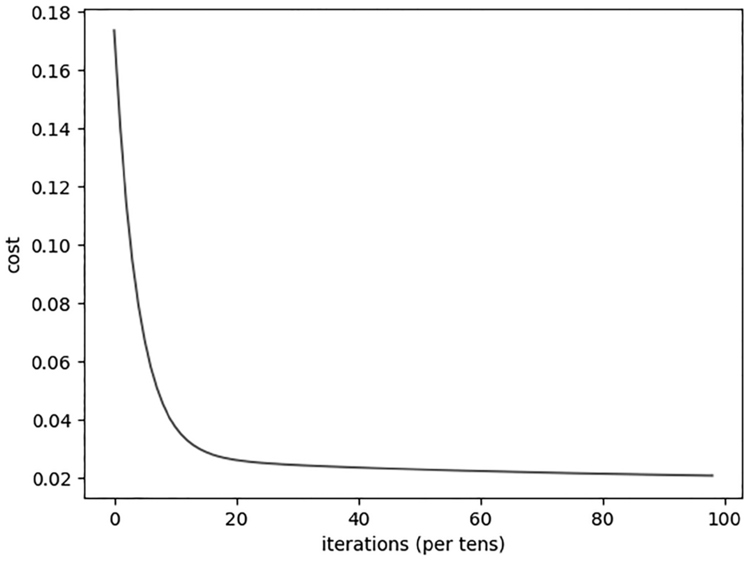

FIGURE 5.9: MSE values with linear regression.

The code sample `plain-linreg2.py` is similar to the code in Listing 5.5: the difference is that instead of a single loop with 100 iterations, there is an outer loop that execute 100 times, and during each iteration of the outer loop, the inner loop also execute 100 times.

5.20 Linear Regression with `Keras`

The code sample in this section contains primarily Keras code in order to perform linear regression. If you have read the previous examples in this chapter, this section will be easier for you to understand because the steps for linear regression are the same.

Listing 5.6 displays the contents of `keras_linear_regression.py` that illustrates how to perform linear regression in `Keras`.

Listing 5.6: keras_linear_regression.py

```
#######################################################
#######
#Keep in mind the following important points:
```

```
#1) Always standardize both input features and target
variable:
#doing so only on input feature produces incorrect
predictions
#2) Data might not be normally distributed: check the
data and
#based on the distribution apply StandardScaler,
MinMaxScaler,
#Normalizer or RobustScaler
#########################################################
#######

import tensorflow as tf
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split

df = pd.read_csv('housing.csv')
X  = df.iloc[:,0:13]
y  = df.iloc[:,13].values

mmsc = MinMaxScaler()
X  = mmsc.fit_transform(X)
y  = y.reshape(-1,1)
y  = mmsc.fit_transform(y)

X_train, X_test, y_train, y_test = train_test_split(X,
y, test_size=0.3)

# this Python method creates a Keras model
def build_keras_model():
  model = tf.keras.models.Sequential()
  model.add(tf.keras.layers.Dense(units=13, input_
dim=13))
  model.add(tf.keras.layers.Dense(units=1))
  model.compile(optimizer='adam',loss='mean_squared_erro
r',metrics=['mae','accuracy'])
  return model

batch_size=32
epochs = 40

# specify the Python method 'build_keras_model' to
create a Keras model
```

```
# using the implementation of the scikit-learn regressor
API for Keras
model = tf.keras.wrappers.scikit_learn.
KerasRegressor(build_fn=build_keras_model, batch_
size=batch_size,epochs=epochs)

# train ('fit') the model and then make predictions:
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
#print("y_test:",y_test)
#print("y_pred:",y_pred)

# scatter plot of test values-vs-predictions
fig, ax = plt.subplots()
ax.scatter(y_test, y_pred)
ax.plot([y_test.min(), y_test.max()], [y_test.min(), y_
test.max()], 'r*--')
ax.set_xlabel('Calculated')
ax.set_ylabel('Predictions')
plt.show()
```

Listing 5.6 starts with multiple `import` statements and then initializes the dataframe `df` with the contents of the `CSV` file `housing.csv` (a portion of which is shown in Listing 5.7). Notice that the training set `X` is initialized with the contents of the first 13 columns of the dataset `housing.csv`, and the variable y contains the rightmost column of the dataset `housing.csv`.

The next section in Listing 5.6 uses the `MinMaxScaler` class to calculate the mean and standard deviation, and then invokes the `fit_transform()` method in order to update the `X` values and the `y` values so that they have a mean of 0 and a standard deviation of 1.

Next, the `build_keras_mode()` Python method creates a Keras-based model with two dense layers. Notice that the input layer has size 13, which is the number of columns in the dataframe `X`. The next code snippet compiles the model with an `adam` optimizer, the MSE loss function, and also specifies the MAE and accuracy for the metrics. The compiled model is then returned to the caller.

The next portion of Listing 5.6 initializes the `batch_size` variable to 32 and the epochs variable to 40, and specifies them in the code snippet that creates the model, as shown here:

```
model = tf.keras.wrappers.scikit_learn.
KerasRegressor(build_fn=build_keras_model, batch_
size=batch_size,epochs=epochs)
```

The short comment block that appears in Listing 5.6 explains the purpose of the preceding code snippet, which constructs our `Keras` model.

The next portion of Listing 5.6 invokes the `fit()` method to train the model and then invokes the `predict()` method on the `X_test` data to calculate a set of predictions and initialize the variable `y_pred` with those predictions.

The final portion of Listing 5.6 displays a scatter plot in which the horizontal axis is the values in `y_test` (the actual values from the CSV file `housing.csv`) and the vertical axis is the set of predicted values.

Figure 5.10 displays a scatter plot of points based on the test values and the predictions for those test values.

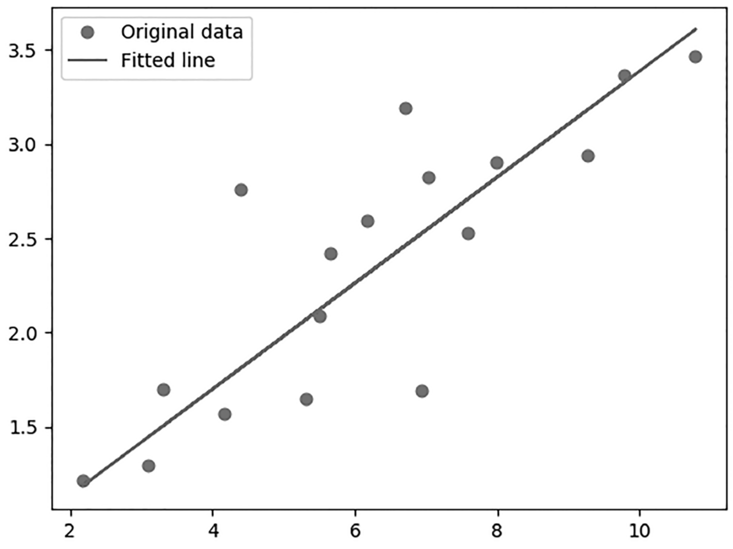

FIGURE 5.10: A scatter plot and a best-fitting line.

Listing 5.7 displays the first four rows of the CSV file `housing.csv` that is used in the Python code in Listing 5.6.

Listing 5.7: housing.csv

```
0.00632,18,2.31,0,0.538,6.575,65.2,4.09,1,296,15.3,396.9
,4.98,24
```

```
0.02731,0,7.07,0,0.469,6.421,78.9,4.9671,2,242,17.8,396.
9,9.14,21.6
0.02729,0,7.07,0,0.469,7.185,61.1,4.9671,2,242,17.8,392.
83,4.03,34.7
0.03237,0,2.18,0,0.458,6.998,45.8,6.0622,3,222,18.7,394.
63,2.94,33.4
```

5.21 Summary

This chapter introduced you to machine learning and concepts such as feature selection, feature engineering, data cleaning, training sets, and test sets. Next you learned about supervised, unsupervised, and semisupervised learning. Then you learned regression tasks, classification tasks, and clustering, as well as the steps that are typically required in order to prepare a dataset. These steps include *feature selection* or *feature extraction* that can be performed using various algorithms. Then you learned about issue that can arise with the data in datasets, and how to rectify them.

In addition, you also learned about linear regression, along with a brief description of how to calculate a best-fitting line for a dataset of values in the Euclidean plane. You saw how to perform linear regression using `NumPy` in order to initialize arrays with data values, along with a "perturbation" technique that introduces some randomness for the y values. This technique is useful because you will know the correct values for the slope and y-intercept of the best-fitting line, which you can then compare with the trained values.

You then learned how to perform linear regression in code samples that involve `Keras`. In addition, you saw how to use Matplotlib in order to display line graphs for best-fitting lines and graphs that display the cost versus the number of iterations during the training-related code blocks.

CHAPTER 6

CLASSIFIERS IN MACHINE LEARNING

- What is Classification?
- What Are Linear Classifiers?
- What is kNN?
- What Are Decision Trees?
- What Are Random Forests?
- What Are SVMs?
- What is Bayesian Inference?
- What is a Bayesian Classifier?
- Training Classifiers
- Evaluating Classifiers
- What Are Activation Functions?
- Common Activation Functions
- The ReLU and ELU Activation Functions
- Sigmoid, Softmax, and Hardmax Similarities
- Sigmoid, Softmax, and HardMax Differences
- What is Logistic Regression?
- Keras and Logistic Regression
- Keras, Logistic Regression, and Iris Dataset
- Summary

This chapter presents numerous classification algorithms in machine learning. This includes algorithms such as the k nearest neighbor (kNN) algorithm, logistic regression (despite its name it *is* a classifier), decision trees, random forests, SVMs, and Bayesian classifiers. The emphasis on algorithms is intended to introduce you to machine learning, which includes a tree-based code sample that relies on `scikit-learn`. The latter portion of this chapter contains `Keras`-based code samples for standard datasets.

Due to space constraints, this chapter does not cover other well-known algorithms, such as linear discriminant analysis and the kMeans algorithm (which is for unsupervised learning and clustering). However, there are many online tutorials available that discuss these and other algorithms in machine learning.

With the preceding points in mind, the first section of this chapter briefly discusses the classifiers that are mentioned in the introductory paragraph. The second section of this chapter provides an overview of activation functions, which will be very useful if you decide to learn about deep neural networks. In this section you will learn how and why they are used in neural networks. This section also contains a list of the TensorFlow APIs for activation functions, followed by a description of some of their merits.

The third section introduces logistic regression, which relies on the sigmoid function, which is also used in recurrent neural networks (RNNs) and long short term memory (LSTMs). The fourth part of this chapter contains a code sample involving logistic regression and the `MNIST` dataset.

In order to give you some context, classifiers are one of three major types of algorithms: regression algorithms (such as linear regression in Chapter 4), classification algorithms (discussed in this chapter), and clustering algorithms (such as kMeans, which is not discussed in this book).

Another point: the section pertaining to activation functions does involve a basic understanding of hidden layers in a neural network. Depending on your comfort level, you might benefit from reading some preparatory material before diving into this section (there are many articles available online).

6.1 What is Classification?

Given a dataset that contains observations whose class membership is known, classification is the task of determining the class to which a new data point belongs. Classes refer to categories and are also called targets or labels. For example, spam detection in email service providers involves binary classification (only 2 classes). The `MNIST` dataset contains a set of images, where each image is a single digit, which means there are 10 labels. Some applications in classification include: credit approval, medical diagnosis, and target marketing.

6.1.1 What Are Classifiers?

In the previous chapter, you learned that linear regression uses supervised learning in conjunction with numeric data: the goal is to train a model that can make numeric predictions (e.g., the price of stock tomorrow, the temperature of a system, its barometric pressure, and so forth). By contrast, classifiers use supervised learning in conjunction with nonnumeric classes of data: the goal is to train a model that can make categorical predictions.

For instance, suppose that each row in a dataset is a specific wine, and each column pertains to a specific wine feature (tannin, acidity, and so forth). Suppose further that there are five classes of wine in the dataset: for simplicity, let's label them A, B, C, D, and E. Given a new data point, which is to say a new row of data, a classifier for this dataset attempts to determine the label for this wine.

Some of the classifiers in this chapter can perform categorical classification and also make numeric predictions (i.e., they can be used for regression as well as classification).

6.1.2 Common Classifiers

Some of the most popular classifiers for machine learning are listed here (in no particular order):

- linear classifiers
- kNN
- logistic regression
- decision trees
- random forests
- SVMs
- Bayesian classifiers
- CNNs (deep learning)

Keep in mind that different classifiers have different advantages and disadvantages, which often involves a trade-off between complexity and accuracy, similar to algorithms in fields that are outside of AI.

In the case of deep learning, convolutional neural networks (CNNs) perform image classification, which makes them classifiers (they can also be used for audio and text processing).

The upcoming sections provide a brief description of the ML classifiers that are listed in the previous list.

6.1.3 Binary versus Multiclass Classification

Binary classifiers work with dataset that have two classes, whereas multiclass classifiers (sometimes called multinomial classifiers) distinguish more than two classes. Random forest classifiers and naïve Bayes classifiers support multiple classes, whereas SVMs and linear classifiers are binary classifiers (but multi-class extensions exist for SVM).

In addition, there are techniques for multiclass classification that are based on binary classifiers: one-versus-all (OvA) and one-versus-one (OvO).

The OvA technique (also called one-versus-the-rest) involves multiple binary classifiers that is equal to the number of classes. For example, if a dataset has five classes, then OvA uses five binary classifiers, each of which detects one of the five classes. In order to classify a data point in this particular dataset, select the binary classifier that has output the highest score.

The OvO technique also involves multiple binary classifiers, but in this case a binary classifier is used to train on a pair of classes. For instance, if the classes are A, B, C, D, and E, then 10 binary classifiers are required: one for A and B, one for A and C, one for A and D, and so forth, until the last binary classifier for D and E.

In general, if there are `n` classes, then `n*(n-1)/2` binary classifiers are required. Although the OvO technique requires considerably more binary classifiers (e.g., 190 are required for 20 classes) than the OvA technique (e.g., a mere 20 binary classifiers for 20 classes), the OvO technique has the advantage that each binary classifier is only trained on the portion of the dataset that pertains to its two chosen classes.

6.1.4 Multilabel Classification

Multilabel classification involves assigning multiple labels to an instance from a dataset. Hence, multilabel classification generalizes multiclass classification (discussed in the previous section), where the latter involves assigning a single label to an instance belonging to a dataset that has multiple classes. An article involving multilabel classification that contains `Keras`-based code is here:

https://medium.com/@vijayabhaskar96/multi-label-image-classification-tutorial-with-keras-imagedatagenerator-cd541f8eaf24

You can also perform an online search for articles that involve SKLearn or PyTorch for multilabel classification tasks.

6.2 What are Linear Classifiers?

A linear classifier separates a dataset into two classes. A linear classifier is a line for 2D points, a plane for 3D points, and a hyperplane (a generalization of a plane) for higher dimensional points.

Linear classifiers are often the fastest classifiers, so they are often used when the speed of classification is of high importance. Linear classifiers usually work well when the input vectors are sparse (i.e., mostly zero values) or when the number of dimensions is large.

6.3 What is kNN?

The k nearest neighbor (kNN) algorithm is a classification algorithm. In brief, data points that are "near" each other are classified as belonging to the same class. When a new point is introduced, it's added to the class of the majority of its nearest neighbor. For example, suppose that k equals 3, and a new data point is introduced. Look at the class of its 3 nearest neighbors: let's say they are A, A, and B. Then by majority vote, the new data point is labeled as a data point of class A.

The kNN algorithm is essentially a heuristic and not a technique with complex mathematical underpinnings, and yet it's still an effective and useful algorithm.

Try the kNN algorithm if you want to use a simple algorithm, or when you believe that the nature of your dataset is highly unstructured. The kNN algorithm can produce highly nonlinear decisions despite being very simple. You can use kNN in search applications where you are searching for "similar" items.

Measure similarity by creating a vector representation of the items, and then compare the vectors using an appropriate distance metric (such as Euclidean distance).

Some concrete examples of kNN search include searching for semantically similar documents.

6.3.1 How to Handle a Tie in kNN

An odd value for k is less likely to result in a tie vote, but it's not impossible. For example, suppose that k equals 7, and when a new data point is introduced, its 7 nearest neighbors belong to the set {A,B,A,B,A,B,C}. As you can see, there is no majority vote, because there are 3 points in class A, 3 points in class B, and 1 point in class C.

There are several techniques for handling a tie in kNN, as listed here:

- Assign higher weights to closer points
- Increase the value of k until a winner is determined
- Decrease the value of k until a winner is determined
- Randomly select one class

If you reduce k until it equals 1, it's still possible to have a tie vote: there might be two points that are equally distant from the new point, so you need a mechanism for deciding which of those two points to select as the 1-neighbor.

If there is a tie between classes A and B, then randomly select either class A or class B. Another variant is to keep track of the "tie" votes, and alternate round-robin style to make ensure a more even distribution.

6.4 What are Decision Trees?

Decision trees are another type of classification algorithm that involves a tree-like structure. In a "generic" tree, the placement of a data point is determined by simple conditional logic. As a simple illustration, suppose that a dataset contains a set of numbers that represent ages of people, and let's also suppose that the first number is 50. This number is chosen as the root of the tree, and all numbers that are smaller than 50 are added on the left branch of the tree, whereas all numbers that are greater than 50 are added on the right branch of the tree.

For example, suppose we have the sequence of numbers is {50, 25, 70, 40}. Then we can construct a tree as follows: 50 is the root node; 25 is the left child of 50; 70 is the right child of 50; and 40 is the right child of 20. Each additional numeric value that we add to this dataset is processed to determine which direction to proceed ("left or right") at each node in the tree.

Listing 6.1 displays the contents of `sklearn_tree2.py` that defines a set of 2D points in the Euclidean plane, along with their labels, and then predicts the label (i.e., the class) of several other 2D points in the Euclidean plane.

Listing 6.1: sklearn_tree2.py

```
from sklearn import tree

# X = pairs of 2D points and Y = the class of each point
X = [[0, 0], [1, 1], [2,2]]
Y = [0, 1, 1]

tree_clf = tree.DecisionTreeClassifier()
tree_clf = tree_clf.fit(X, Y)

#predict the class of samples:
print("predict class of [-1., -1.]:")
print(tree_clf.predict([[-1., -1.]]))

print("predict class of [2., 2.]:")
print(tree_clf.predict([[2., 2.]]))

# the percentage of training samples of the same class
# in a leaf note equals the probability of each class
print("probability of each class in [2.,2.]:")
print(tree_clf.predict_proba([[2., 2.]]))
```

Listing 6.1 imports the tree class from `sklearn` and then initializes the arrays `X` and `y` with data values. Next, the variable `tree_clf` is initialized as an instance of the `DecisionTreeClassifier` class, after which it is trained by invoking the `fit()` method with the values of `X` and `y`.

Now launch the code in Listing 6.3 and you will see the following output:

```
predict class of [-1., -1.]:
[0]
predict class of [2., 2.]:
[1]
probability of each class in [2.,2.]:
[[0. 1.]]
```

As you can see, the points [-1,-1] and [2,2] are correctly labeled with the values 0 and 1, respectively, which is probably what you expected.

Listing 6.2 displays the contents of `sklearn_tree3.py` that extends the code in Listing 6.1 by adding a third label, and also by predicting the label of three points instead of two points in the Euclidean plane (the modifications are shown in bold).

Listing 6.2: sklearn_tree3.py

```
from sklearn import tree

# X = pairs of 2D points and Y = the class of each point
X = [[0, 0], [1, 1], [2,2]]
Y = [0, 1, 2]

tree_clf = tree.DecisionTreeClassifier()
tree_clf = tree_clf.fit(X, Y)

#predict the class of samples:
print("predict class of [-1., -1.]:")
print(tree_clf.predict([[-1., -1.]]))

print("predict class of [0.8, 0.8]:")
print(tree_clf.predict([[0.8, 0.8]]))

print("predict class of [2., 2.]:")
print(tree_clf.predict([[2., 2.]]))

# the percentage of training samples of the same class
# in a leaf note equals the probability of each class
print("probability of each class in [2.,2.]:")
print(tree_clf.predict_proba([[2., 2.]]))
```

Now launch the code in Listing 6.2 and you will see the following output:

```
predict class of [-1., -1.]:
[0]
predict class of [0.8, 0.8]:
[1]
predict class of [2., 2.]:
[2]
probability of each class in [2.,2.]:
[[0. 0. 1.]]
```

As you can see, the points [-1,-1], [0.8, 0.8], and [2,2] are correctly labeled with the values 0, 1, and 2, respectively, which again is probably what you expected.

Listing 6.3 displays a portion of the dataset `partial_wine.csv`, which contains two features and a label column (there are three classes). The total row count for this dataset is 178.

Listing 6.3: partial_wine.csv

```
Alcohol, Malic acid, class
14.23,1.71,1
13.2,1.78,1
13.16,2.36,1
14.37,1.95,1
13.24,2.59,1
14.2,1.76,1
```

Listing 6.4 displays contents of `tree_classifier.py` that uses a decision tree in order to train a model on the dataset `partial_wine.csv`.

Listing 6.4: tree_classifier.py

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

# Importing the dataset
dataset = pd.read_csv('partial_wine.csv')
X = dataset.iloc[:, [0, 1]].values
y = dataset.iloc[:, 2].values

# split the dataset into a training set and a test set
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X,
y, test_size = 0.25, random_state = 0)

# Feature Scaling
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)

# ====> INSERT YOUR CLASSIFIER CODE HERE <====
from sklearn.tree import DecisionTreeClassifier
classifier = DecisionTreeClassifier(criterion='entropy',
random_state=0)
classifier.fit(X_train, y_train)
# ====> INSERT YOUR CLASSIFIER CODE HERE <====

# predict the test set results
```

```
y_pred = classifier.predict(X_test)

# generate the confusion matrix
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)
print("confusion matrix:")
print(cm)
```

Listing 6.4 contains some `import` statements and then populates the `Pandas` DataFrame `dataset` with the contents of the CSV file `partial_wine.csv`. Next, the variable `X` is initialized with the first two columns (and all the rows) of dataset, and the variable `y` is initialized with the third column (and all the rows) of dataset.

Next, the variables `X_train, X_test, y_train, y_test` are populated with data from `X` and `y` using a 75/25 split proportion. Notice that the variable `sc` (which is an instance of the `StandardScalar` class) performs a scaling operation on the variables `X_train` and `X_test`.

The code block shown in bold in Listing 6.4 is where we create an instance of the `DecisionTreeClassifier` class and then train the instance with the data in the variables `X_train` and `X_test`.

The next portion of Listing 6.4 populates the variable `y_pred` with a set of predictions that are generated from the data in the `X_test` variable. The last portion of Listing 6.4 creates a confusion matrix based on the data in `y_test` and the predicted data in `y_pred`.

Remember that all the diagonal elements of a confusion matrix are correct predictions (such as true positive and true negative); all the other cells contain a numeric value that specifies the number of predictions that are incorrect (such as false positive and false negative).

Now launch the code in Listing 6.4 and you will see the following output for the confusion matrix in which there are 36 correct predictions and 9 incorrect predictions (with an accuracy of 80%):

```
confusion matrix:
[[13  1  2]
 [ 0 17  4]
 [ 1  1  6]]
from sklearn.metrics import confusion_matrix
```

There is a total of 45 entries in the preceding 3x3 matrix, and the diagonal entries are correctly identified labels. Hence the accuracy is 36/45 = 0.80.

6.5 What are Random Forests?

Random forests are a generalization of decision trees: this classification algorithm involves multiple trees (the number is specified by you). If the data involves making a numeric prediction, the average of the predictions of the trees is computed. If the data involves a categorical prediction, the mode of the predictions of the trees is determined.

By way of analogy, random forests operate in a manner similar to financial portfolio diversification: the goal is to balance the losses with higher gains. Random forests use a “majority vote” to make predictions, which operates under the assumption that selecting the majority vote is more likely to be correct (and more often) than any individual prediction from a single tree.

You can easily modify the code in Listing 6.4 to use a random forest by replacing the two lines shown in bold with the following code:

```
from sklearn.ensemble import RandomForestClassifier
classifier = RandomForestClassifier(n_estimators = 10,
criterion='entropy', random_state = 0)
```

Make this code change, launch the code, and examine the confusion matrix to compare its accuracy with the accuracy of the decision tree in Listing 6.4.

6.6 What are SVMs?

Support vector machines (SVM) involve a supervised ML algorithm and can be used for classification or regression problems. SVM can work with nonlinearly separable data as well as linearly separable data. SVM uses a technique called the "kernel trick" to transform data and then finds an optimal boundary the transform involves higher dimensionality. This technique results in a separation of the transformed data, after which it’s possible to find a hyperplane that separates the data into two classes.

SVMs are more common in classification tasks than regression tasks. Some use cases for SVMs include:

- text classification tasks: category assignment
- detecting spam / sentiment analysis
- used for image recognition: aspect-based recognition color-based classification

- handwritten digit recognition (postal automation)

6.6.1 Tradeoffs of SVMs

Although SVMs are extremely powerful, there are tradeoffs involved. Some of the advantages of SVMs are listed here:

- high accuracy
- works well on smaller cleaner datasets
- can be more efficient because it uses a subset of training points
- an alternative to CNNs in cases of limited datasets
- captures more complex relationships between data points

Despite the power of SVMS, there are some disadvantages of SVMs, which are listed here:

- not suited to larger datasets: training time can be lengthy
- less effective on noisier datasets with overlapping classes

SVMs involve more parameters than decision trees and random forests

Suggestion: modify Listing 6.4 to use an SVM by replacing the two lines shown in bold with the following two lines shown in bold:

```
from sklearn.svm import SVC
classifier = SVC(kernel = 'linear', random_state = 0)
```

You now have an SVM-based model, simply by making the previous code update. Make the code change, then launch the code and examine the confusion matrix in order to compare its accuracy with the accuracy of the decision tree model and the random forest model earlier in this chapter.

6.7 What is Bayesian Inference?

Bayesian inference is an important technique in statistics that involves statistical inference and Bayes' theorem to update the probability for a hypothesis as more information becomes available. Bayesian inference is often called *Bayesian probability*, and it's important in dynamic analysis of sequential data.

6.7.1 Bayes Theorem

Given two sets A and B, let's define the following numeric values (all of them are between 0 and 1):

```
P(A) = probability of being in set A
P(B) = probability of being in set B
P(Both) = probability of being in A intersect B
P(A|B) = probability of being in A (given you're in B)
P(B|A) = probability of being in B (given you're in A)
```

Then the following formulas are also true:

```
P(A|B) = P(Both)/P(B) (#1)
P(B|A) = P(Both)/P(A) (#2)
```

Multiply the preceding pair of equations by the term that appears in the denominator and we get these equations:

```
P(B)*P(A|B) = P(Both) (#3)
P(A)*P(B|A) = P(Both) (#4)
```

Now set the left-side of equations #3 and #4 equal to each another and that gives us this equation:

```
P(B)*P(A|B) = P(A)*P(B|A) (#5)
```

Divide both sides of #5 by P(B) and we get this well-known equation:

```
P(A|B) = P(A)*P(A|B)/P(B) (#6)
```

6.7.2 Some Bayesian Terminology

In the previous section, we derived the following relationship:

```
P(h|d) = (P(d|h) * P(h)) / P(d)
```

There is a name for each of the four terms in the preceding equation, and they are:

First, the *posterior probability* is `P(h|d)`, which is the probability of hypothesis `h` given the data `d`.

Second, `P(d|h)` is the probability of data `d` given that the hypothesis `h` was true.

Third, the *prior probability* of `h` is `P(h)`, which is the probability of hypothesis `h` being true (regardless of the data).

Finally, `P(d)` is the probability of the data (regardless of the hypothesis).

We are interested in calculating the posterior probability of P(h|d) from the prior probability p(h) with P(d) and P(d|h).

6.7.3 What Is MAP?

The maximum a posteriori (MAP) hypothesis is the hypothesis with the highest probability, which is the maximum probable hypothesis. This can be written as follows:

```
MAP(h) = max(P(h|d))
or:
MAP(h) = max((P(d|h) * P(h)) / P(d))
or:
MAP(h) = max(P(d|h) * P(h))
```

6.7.4 Why Use Bayes Theorem?

Bayes' theorem describes the probability of an event based on the prior knowledge of the conditions that might be related to the event. If we know the conditional probability, we can use Bayes rule to find out the reverse probabilities. The previous statement is the general representation of the Bayes rule.

6.8 What is a Bayesian Classifier?

A naïve Bayes (NB) classifier is a probabilistic classifier inspired by the Bayes theorem. An NB classifier assumes the attributes are conditionally independent and it works well even when assumption is not true. This assumption greatly reduces computational cost, and it's a simple algorithm to implement that only requires linear time. Moreover, an NB classifier is easily scalable to larger datasets and good results are obtained in most cases. Other advantages of an NB classifier include:

- can be used for binary and multiclass classification
- provides different types of NB algorithms
- good choice for text classification problems
- a popular choice for spam email classification
- can be easily trained on small datasets

As you can probably surmise, NB classifiers do have some disadvantages, as listed here:

- all features are assumed unrelated
- it cannot learn relationships between features
- it can suffer from "the zero probability problem"

The *zero probability problem* refers to the case when the conditional probability is zero for an attribute, it fails to give a valid prediction. However, can be fixed explicitly using a Laplacian estimator.

6.8.1 Types of Naïve Bayes Classifiers

There are three major types of NB classifiers:

- Gaussian Naïve Bayes
- multinomialNB Naïve Bayes
- Bernoulli Naïve Bayes

Details of these classifiers are beyond the scope of this chapter, but you can perform an online search for more information.

6.9 Training Classifiers

Some common techniques for training classifiers are here:

- holdout method
- k-fold cross-validation

The *holdout method* is the most common method, which starts by dividing the dataset into two partitions called train and test (80% and 20%, respectively). The train set is used for training the model, and the test data tests its predictive power.

The *k-fold cross-validation* technique is used to verify that the model is not over-fitted. The dataset is randomly partitioned into k mutually exclusive subsets, where each partition has equal size. One partition is for testing and the other partitions are for training. Iterate throughout the whole of the k-folds.

6.10 Evaluating Classifiers

Whenever you select a classifier for a dataset, it's obviously important to evaluate the accuracy of that classifier. Some common techniques for evaluating classifiers are listed here:

- precision and recall
- receiver operating characteristics (ROC) curve

Precision and recall are discussed in Chapter 2 and reproduced here for your convenience. Let's define the following variables:

```
TP = the number of true positive results
FP = the number of false positive results
TN = the number of true negative results
FN = the number of false negative results
```

Then the definitions of precision, accuracy, and recall are given by the following formulas:

```
precision = TP/(TN + FP)
accuracy  = (TP + TN)/[P + N]
recall    = TP/[TP + FN]
```

The *receiver operating characteristics* (ROC) curve is used for visual comparison of classification models that shows the trade-off between the true positive rate and the false positive rate. The area under the ROC curve is a measure of the accuracy of the model. When a model is closer to the diagonal, it is less accurate and the model with perfect accuracy will have an area of 1.0.

The ROC curve plots true positive rate versus false positive rate. Another type of curve is the precision-recall (PR) curve that plots rrecision versus recall. When dealing with highly skewed datasets (strong class imbalance), PR curves give better results.

Later in this chapter you will see many of the `Keras`-based classes (located in the `tf.keras.metrics` namespace) that correspond to common statistical terms, which includes some of the terms in this section.

This concludes the portion of the chapter pertaining to statistical terms and techniques for measuring the validity of a dataset. Now let's look at activation functions in machine learning, which is the topic of the next section.

6.11 What are Activation Functions?

A one-sentence description: an activation function is (usually) a nonlinear function that introduces nonlinearity into a neural network, thereby preventing a *consolidation* of the hidden layers in neural network. Specifically, suppose that every pair of adjacent layers in a neural network involves just a matrix transformation and no activation function. *Such a network is a linear system, which means that its layers can be consolidated into a much smaller system.*

First, the weights of the edges that connect the input layer with the first hidden layer can be represented by a matrix: let's call it `W1`. Next, the weights of the edges that connect the first hidden layer with the second hidden layer can also be represented by a matrix: let's call it `W2`. Repeat this process until we reach the edges that connect the final hidden layer with the output layer: let's call this matrix `Wk`. Since we do not have an activation function, we can simply multiply the matrices `W1`, `W2`, …, `Wk` together and produce one matrix: let's call it `W`. We have now replaced the original neural network with an equivalent neural network that contains one input layer, a single matrix of weights `W`, and an output layer. In other words, we no longer have our original multilayered neural network.

Fortunately, we can prevent the previous scenario from happening when we specify an activation function between every pair of adjacent layers. In other words, an activation function at each layer prevents this *matrix consolidation*. Hence, we can maintain all the intermediate hidden layers during the process of training the neural network.

For simplicity, let's assume that we have the same activation function between every pair of adjacent layers (we'll remove this assumption shortly). The process for using an activation function in a neural network is initially a "three step," after which it's a "two-step," as described here:

1. Start with an input vector `x1` of numbers.
2. Multiply `x1` by the matrix of weights W1 that represents the edges that connect the input layer with the first hidden layer: the result is a new vector `x2`.
3. "Apply" the activation function to each element of `x2` to create another vector `x3`.

Now repeat steps 2 and 3, except that we use the "starting" vector x3 and the weights matrix `W2` for the edges that connect the first hidden layer with the second hidden layer (or just the output layer if there is only one hidden layer).

After completing the preceding process, we have *preserved* the neural network, which means that it can be trained on a dataset. One other thing: instead of using the same activation function at each step, you can replace each activation function by a different activation function (the choice is yours).

6.11.1 Why Do We Need Activation Functions?

The previous section outlines the process for transforming an input vector from the input layer and then through the hidden layers until it reaches the output layer. The purpose of activation functions in neural networks is vitally important, so it's worth repeating here: activation functions "maintain" the structure of neural networks and prevent them from being reduced to an input layer and an output layer. In other words, if we specify a nonlinear activation function between every pair of consecutive layers, then the neural network cannot be replaced with a neural network that contains fewer layers.

Without a nonlinear activation function, we simply multiply a weight matrix for a given pair of consecutive layers with the output vector that is produced from the previous pair of consecutive layers. We repeat this simple multiplication until we reach the output layer of the neural network. After reaching the output layer, we have effectively replaced multiple matrices with a single matrix that connects the numbers in the input layer with the numbers in the output layer.

6.11.2 How Do Activation Functions Work?

If this is the first time you have encountered the concept of an activation function, it's probably confusing, so here's an analogy that might be helpful. Suppose you're driving your car late at night and there's nobody else on the highway. You can drive at a constant speed for as long as there are no obstacles (stop signs, traffic lights, and so forth). However, suppose you drive into the parking lot of a large grocery store. When you approach a speed bump you must slow down, cross the speed bump, and increase speed again, and repeat this process for every speed bump.

Think of the nonlinear activation functions in a neural network as the counterpart to the speed bumps: you simply cannot maintain a constant speed, which (by analogy) means that you cannot first multiply all the weight matrices together and "collapse" them into a single weight matrix. Another analogy involves a road with multiple toll booths: you must slow down, pay the toll, and then resume driving until you reach the next toll booth. These are only analogies (and hence imperfect) to help you understand the need for nonlinear activation functions.

6.12 Common Activation Functions

Although there are many activation functions (and you can define your own if you know how to do so), here is a list of common activation functions, followed by brief descriptions:

- Sigmoid
- Tanh
- ReLU
- ReLU6
- ELU
- SELU

The `sigmoid` activation function is based on Euler's constant e, with a range of values between 0 and 1, and its formula is shown here:

```
1/[1+e^(-x)]
```

The `tanh` activation function is also based on Euler's constant e, and its formula is shown here:

```
[e^x – e^(-x)]/[e^x+e^(-x)]
```

One way to remember the preceding formula is to note that the numerator and denominator have the same pair of terms: they are separated by a "-" sign in the numerator and a "+" sign in the denominator. The `tanh` function has a range of values between -1 and 1.

The rectified linear unit (ReLU) activation function is straightforward: if x is negative then ReLU(x) is 0; for all other values of x, ReLU(x) equals x. ReLU6 is specific to TensorFlow, and it's a variation of ReLU(x): the additional constraint is that ReLU(x) equals 6 when x >= 6 (hence its name).

exponential linear unit (ELU) is the exponential "envelope" of ReLU, which replaces the two linear segments of ReLU with an Exponential activation function that is differentiable for all values of x (including x = 0).

Scaled exponential linear unit (SELU)is slightly more complicated than the other activation functions (and used less frequently). For a thorough explanation of these and other activation functions (along with graphs that depict their shape), navigate to the following Wikipedia link:

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

The preceding link provides a long list of activation functions as well as their derivatives.

6.12.1 Activation Functions in Python

Listing 6.5 displays contents of the file `activations.py` that contains the formulas for various activation functions.

Listing 6.5: activations.py

```
import numpy as np

# Python sigmoid example:
z = 1/(1 + np.exp(-np.dot(W, x)))

# Python tanh example:
z = np.tanh(np.dot(W,x))

# Python ReLU example:
z = np.maximum(0, np.dot(W, x))
```

Listing 6.5 contains Python code that use `NumPy` methods in order to define a sigmoid function, a `tanh` function, and a ReLU function. Note that you need to specify values for `x` and `W` in order to launch the code in Listing 6.5.

6.12.2 `Keras` Activation Functions

TensorFlow (and many other frameworks) provide implementations for many activation functions, which saves you the time and effort from writing your own implementation of activation functions.

Here is a list of TF 2/Keras APIs for activation functions that are located in the tf.keras.layers namespace:

- tf.keras.layers.leaky_relu
- tf.keras.layers.relu
- tf.keras.layers.relu6
- tf.keras.layers.selu
- tf.keras.layers.sigmoid

- tf.keras.layers.sigmoid_cross_entropy_with_logits
- tf.keras.layers.softmax
- tf.keras.layers.softmax_cross_entropy_with_logits_v2
- tf.keras.layers.softplus
- tf.keras.layers.softsign
- tf.keras.layers.softmax_cross_entropy_with_logits
- tf.keras.layers.tanh
- tf.keras.layers.weighted_cross_entropy_with_logits

The following subsections provide additional information regarding some of the activation functions in the preceding list. Keep the following point in mind: for simple neural networks, use ReLU as your first preference.

6.13 The ReLU and ELU Activation Functions

Currently ReLU is often the recommended activation function: previously the preferred activation function was `tanh` (and before `tanh` it was `sigmoid`). ReLU behaves close to a linear unit and provides the best training accuracy and validation accuracy.

ReLU is like a switch for linearity: it's "off" if you don't need it, and its derivative is 1 when it's active, which makes ReLU the simplest of all the current activation functions. Note that the second derivative of the function is 0 everywhere: it's a very simple function that simplifies optimization. In addition, the gradient is large whenever you need large values, and it never saturates (i.e., it does not shrink to zero on the positive horizontal axis).

Rectified linear units and generalized versions are based on the principle that linear models are easier to optimize. Use the ReLU activation function or one of its related alternatives (discussed later).

6.13.1 The Advantages and Disadvantages of ReLU

The following list contains the advantages of the ReLU activation function:

- It does not saturate in the positive region.
- It's very efficient in terms of computation.
- Models with ReLU typically converge faster those with other activation functions.

However, ReLU does have a disadvantage when the activation value of a ReLU neuron becomes 0: then the gradients of the neuron will also be 0 during back-propagation. You can mitigate this scenario by judiciously assigning the values for the initial weights as well as the learning rate.

6.13.2 ELU

Exponential linear unit (ELU) is based on ReLU: the key difference is that ELU is differentiable at the origin (ReLU is a continuous function but *not* differentiable at the origin). However, keep in mind several points. First, ELU's trade computational efficiency for "immortality" (immunity to dying): read the following paper for more details: arxiv.org/abs/1511.07289. Secondly, RELUs are still popular and preferred over ELU because the use of ELU introduces an additional new hyper-parameter.

6.14 Sigmoid, Softmax, and Hardmax Similarities

The `sigmoid` activation function has a range in (0,1), and it saturates and "kills" gradients. Unlike the `tanh` activation function, `sigmoid` outputs are not zero-centered. In addition, both `sigmoid` and `softmax` (discussed later) are discouraged for vanilla feed forward implementation. (See Chapter 6 of the online book *Deep Learning* by Ian Goodfellow et al. 2015). However, the `sigmoid` activation function is still used in LSTMs (specifically for the forget gate, input gate, and the output gate), gated recurrent units (GRUs), and probabilistic models. Moreover, some autoencoders have additional requirements that preclude the use of piecewise linear activation functions.

6.14.1 Softmax

The `softmax` activation function maps the values in a dataset to another set of values that are between 0 and 1, and whose sum equals 1. Thus, `softmax` creates a probability distribution. In the case of image classification with convolutional neural networks (CNNs), the `softmax` activation function maps the values in the final hidden layer to the 10 neurons in the output layer. The index of the position that contains the largest probability is matched with the index of the number 1 in the one-hot encoding of the input image. If the index values are equal, then the image has been classified, otherwise it's considered a mismatch.

6.14.2 Softplus

The `softplus` activation function is a smooth (i.e., differentiable) approximation to the ReLU activation function. Recall that the origin is the only nondifferentiable point of the ReLU function, which is "smoothed" by the `softmax` activation whose equation is here:

```
f(x) = ln(1 + e^x)
```

6.14.3 Tanh

The `tanh` activation function has a range in (-1,1), whereas the `sigmoid` function has a range in (0,1). Both of these two activations saturate, but unlike the `sigmoid` neuron the `tanh` output is zero-centered. Therefore, in practice the `tanh` nonlinearity is always preferred to the `sigmoid` nonlinearity.

The `sigmoid` and `tanh` activation functions appear in LSTMs (sigmoid for the three gates and tanh for the internal cell state) as well as GRUs during the calculations pertaining to input gates, forget gates, and output gates (discussed in more detail in the next chapter).

6.15 Sigmoid, Softmax, and HardMax Differences

This section briefly discusses some of the differences among these three functions. First, the `sigmoid` function is used for binary classification in logistic regression model, as well as the gates in LSTMs and GRUs. The `sigmoid` function is used as activation function while building neural networks, but keep in mind that the sum of the probabilities is *not* necessarily equal to 1.

Second, the `softmax` function generalizes the `sigmoid` function: it's used for multiclassification in logistic regression model. The `softmax` function is the activation function for the *fully connected layer* in CNNs, which is the right-most hidden layer and the output layer. Unlike the sigmoid function, the sum of the probabilities *must* equal 1. You can use either the sigmoid function or `softmax` for binary (n=2) classification.

Third, the so-called "`hardmax`" function assigns 0 or 1 to output values (similar to a step function). For example, suppose that we have three classes `{c1, c2, c3}` whose scores are `[1, 7, 2]`, respectively. The `hardmax` probabilities are `[0, 1, 0]`, whereas the `softmax` probabilities are `[0.1,` `0.7, 0.2]`. Notice that the sum of the `hardmax` probabilities is 1, which is also true of the sum of the `softmax` probabilities. However, the `hard-max` probabilities are all-or-nothing, whereas the `softmax` probabilities are analogous to receiving "partial credit."

6.16 What is Logistic Regression?

Despite its name, logistic regression is a classifier and a linear model with a binary output. Logistic regression works with multiple independent variables and involves a sigmoid function for calculating probabilities. 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), or ordinal (ordered dependent variables), but mainly used for binomial cases. 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.

6.16.1 Setting a Threshold Value

The *threshold value* is a numeric value that determines which data points belong to class A and which points belong to class B. For instance, a pass/fail threshold might be 0.70. A pass/fail threshold for passing a writing driver's test in California is 0.85.

As another example, suppose that $p = 0.5$ is the "cutoff" probability. Then we can assign class A to the data points that occur with probability > 0.5 and assign class B to data points that occur with probability $<= 0.5$. Since there are only two classes, we do have a classifier.

A similar (yet slightly different) scenario involves tossing a well-balanced coin. We know that there is a 50% chance of throwing heads (let's label this outcome as class A) and a 50% chance of throwing tails (let's label this outcome as class B). If we have a dataset that consists of labeled outcomes, then we have the expectation that approximately 50% of them are class A and class B.

However, we have no way to determine (in advance) what percentage of people will pass their written driver's test, or the percentage of people who will pass their course. Datasets containing outcomes for these types of scenarios need to be trained, and logistic regression can be a suitable technique for doing so.

6.16.2 Logistic Regression: Important Assumptions

Logistic regression requires the observations to be independent of each other. In addition, logistic regression requires little or no multicollinearity among the independent variables. Logistic regression handles numeric, categorical, and continuous variables, and also assumes linearity of independent variables and log odds, which is defined here:

```
odds = p/(1-p) and logit = log(odds)
```

This analysis does not require the dependent and independent variables to be related linearly; however, another requirement is that independent variables are linearly related to the log odds.

Logistic regression is used to obtain odds ratio in the presence of more than one explanatory variable. The procedure is quite similar to multiple linear regression, with the exception that the response variable is binomial. The result is the impact of each variable on the odds ratio of the observed event of interest.

6.16.3 Linearly Separable Data

Linearly separable data is data that can be separated by a line (in 2D), a plane (in 3D), or a hyperplane (in higher dimensions). Linearly nonseparable data is data (clusters) that cannot be separated by a line or a hyperplane. For example, the XOR function involves data points that cannot be separated by a line. If you create a truth table for an `XOR` function with two inputs, the points (0,0) and (1,1) belong to class 0, whereas the points (0,1) and (1,0) belong to class 1 (draw these points in a 2D plane to convince yourself). The solution involves transforming the data in a higher dimension so that it becomes linearly separable, which is the technique used in SVMS (discussed earlier in this chapter).

6.17 `Keras`, Logistic Regression, and Iris Dataset

Listing 6.6 displays the contents of `tf2_keras_iris.py` that defines a `Keras`-based model to perform logistic regression.

Listing 6.6: tf2_keras_iris.py

```
import tensorflow as tf
import matplotlib.pyplot as plt

from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder,
StandardScaler

iris = load_iris()
X = iris['data']
y = iris['target']

#you can view the data and the labels:
#print("iris data:",X)
#print("iris target:",y)

# scale the X values so they are between 0 and 1
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

X_train, X_test, y_train, y_test = train_test_split(X_
scaled, y, test_size = 0.2)

model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Dense(activation='relu',
input_dim=4,
           units=4, kernel_initializer='uniform'))

model.add(tf.keras.layers.Dense(activation='relu',
units=4,
                     kernel_initializer='uniform'))
model.add(tf.keras.layers.Dense(activation='sigmoid',
units=1,
                     kernel_initializer='uniform'))
#model.add(tf.keras.layers.Dense(1,
activation='softmax'))

model.compile(optimizer='adam', loss='mean_squared_
error', metrics=['accuracy'])
model.fit(X_train, y_train, batch_size=10, epochs=100)

# Predicting values from the test set
y_pred = model.predict(X_test)

# scatter plot of test values-vs-predictions
fig, ax = plt.subplots()
```

```
ax.scatter(y_test, y_pred)
ax.plot([y_test.min(), y_test.max()], [y_test.min(), y_
test.max()], 'r*--')

ax.set_xlabel('Calculated')
ax.set_ylabel('Predictions')
plt.show()
```

Listing 6.6 starts with an assortment of `import` statements, and then initializes the variable `iris` with the `Iris` dataset. The variable X contains the first three columns (and all the rows) of the `Iris` dataset, and the variable `y` contains the fourth column (and all the rows) of the `Iris` dataset.

The next portion of Listing 6.6 initializes the training set and the test set using an 80/20 data split. Next, the `Keras`-based model contains three `Dense` layers, where the first two specify the relu activation function and the third layer specifies the sigmoid activation function.

The next portion of Listing 6.6 compiles the model, trains the model, and then calculates the accuracy of the model via the test data. Launch the code in Listing 6.6 and you will see the following output:

```
Train on 120 samples
Epoch 1/100120/120 [==============================] - 0s
980us/sample - loss: 0.9819 - accuracy: 0.3167
Epoch 2/100
120/120 [==============================] - 0s 162us/
sample - loss: 0.9789 - accuracy: 0.3083
Epoch 3/100
120/120 [==============================] - 0s 204us/
sample - loss: 0.9758 - accuracy: 0.3083
Epoch 4/100
120/120 [==============================] - 0s 166us/
sample - loss: 0.9728 - accuracy: 0.3083
Epoch 5/100
120/120 [==============================] - 0s 160us/
sample - loss: 0.9700 - accuracy: 0.3083
// details omitted for brevity
Epoch 96/100
120/120 [==============================] - 0s 128us/
sample - loss: 0.3524 - accuracy: 0.6500
Epoch 97/100
120/120 [==============================] - 0s 184us/
sample - loss: 0.3523 - accuracy: 0.6500
```

```
Epoch 98/100
120/120 [==============================] - 0s 128us/
sample - loss: 0.3522 - accuracy: 0.6500
Epoch 99/100
120/120 [==============================] - 0s 187us/
sample - loss: 0.3522 - accuracy: 0.6500
Epoch 100/100
120/120 [==============================] - 0s 167us/
sample - loss: 0.3521 - accuracy: 0.6500
```

Figure 6.1 displays a scatter plot of points based on the test values and the predictions for those test values.

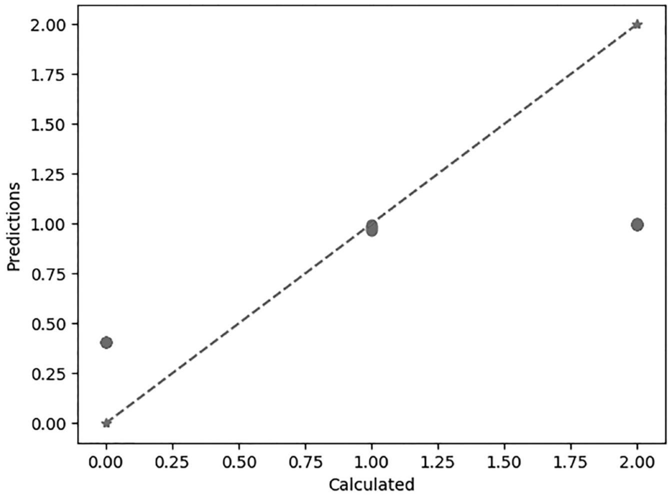

FIGURE 6.1: A scatter plot and a best-fitting line.

The accuracy is admittedly poor (abysmal?), and yet it's quite possible that you will encounter this type of situation. Experiment with a different number of hidden layers and replace the final hidden layer with a `Dense` layer that specifies a `softmax` activation function—or some other activation function—to see if this change improves the accuracy.

6.18 Summary

This chapter started with an explanation of classification and classifiers, followed by a brief explanation of commonly used classifiers in machine learning.

Next you learned about activation functions, why they are important in neural networks, and also how they are used in neural networks. Then you saw a list of the TensorFlow/`Keras` APIs for various activation functions, followed by a description of some of their merits.

You also learned about logistic regression that involves the sigmoid activation function, followed by a `Keras`-based code sample involving logistic regression.

CHAPTER 7

NATURAL LANGUAGE PROCESSING AND REINFORCEMENT LEARNING

- Working with Natural Language Processing (NLP)
- Popular NLP Algorithms
- What Are Word Embeddings?
- ELMo, ULMFit, OpenAI, BERT, and ERNIE 2.0 (optional)
- What is Translatotron?
- Deep Learning and NLP (optional)
- NLU versus NLG (optional)
- What is Reinforcement Learning (RL)?
- From NFAs to MDPs
- The Epsilon-Greedy Algorithm
- The Bellman Equation
- RL Toolkits and Frameworks
- Deep Reinforcement Learning (optional)
- Summary

This chapter provides a casual introduction you to natural language processing (NLP) and reinforcement learning (RL). Both topics can easily fill entire books, often involving complex topics, which means that this chapter provides a limited introduction to these topics. If you want to acquire a thorough grasp of BERT (discussed briefly later in the chapter), you need to learn about "attention" and the transformer architecture. Similarly, if you want to acquire a solid understanding of deep RL, then you need to understand deep learning architectures. After you finish reading the cursory introduction to NLP and RL in this chapter, you can find additional online information about the facets of NLP or RL that interest you.

The first section discusses NLP, along with some code samples in Keras. This section also discusses natural language understanding (NLU) and natural language generation (NLG).

The second section introduces RL, along with a description of the types of tasks that are well-suited to RL. You will learn about the `nchain` task and the epsilon-greedy algorithm that can solve problems that you cannot solve using a "pure" greedy algorithm. In this section you will also learn about the Bellman equation, which is a cornerstone of RLRL.

The third section discusses the TF-Agents toolkit from Google, deep RLRL (deep learning combined with RL), and the Google Dopamine toolkit.

7.1 Working with NLP

This section highlights some concepts in NLP, and depending on your background, you might need to perform an online search to learn more about some of the concepts (try Wikipedia). Although the concepts are treated in a very superficial manner, you will know what to pursue in order to further your study of NLP.

NLP is currently the focus of significant interest in the machine learning (ML) community. Some of the use cases for NLP are listed here:

- chatbots
- search (text and audio)
- text classification
- sentiment analysis
- recommendation systems
- question answering
- speech recognition
- NLU
- NLG

You encounter many of these use cases in every day life: when you visit Web pages, or perform an online search for books, or recommendations regarding movies.

7.1.1 NLP Techniques

The earliest approach for solving NLP tasks involved rule-based approaches, which dominated the industry for decades. Examples of techniques using rule-based approaches include regular expressions (RegExs) and context-free grammars (CFGs). RegExs are sometimes used in order to remove HTML tags from text that has been "scraped" from a Web page, or unwanted special characters from a document.

The second approach involved training a ML model with some data that is based on some user-defined features. This technique requires a considerable amount of feature engineering (a nontrivial task), and includes analyzing the text to remove undesired and superfluous content (including "stop" words), as well as transforming the words (e.g., converting uppercase to lowercase).

The most recent approach involves deep learning, whereby a neural network learns the features instead of relying on humans to perform feature engineering. One of the key ideas involves "mapping" words to numbers, which enables us to map sentences to vectors of numbers. After transforming documents to vectors, we can perform a myriad of operations on those vectors. For example, we can use the notion of vector spaces to define vector space models, where the distance between two vectors can be measured by the angle between them (related to cosine similarity). If two vectors are "close" to each other, then it's likelier that the corresponding sentences are similar in meaning. Their similarity is based on the *distributional hypothesis*, which asserts that words in the same contexts tend to have similar meanings.

A nice article that discusses vector representations of words, along with links to code samples, is here:

https://www.tensorflow.org/tutorials/representation/word2vec

7.1.2 The Transformer Architecture and NLP

In 2017, Google introduced the `Transformer` neural network architecture, which is based on a "self-attention" mechanism that is well-suited for language understanding.

Google showed that the Transformer outperforms earlier benchmarks for both RNNs and CNNs involving the translation of academic English to German as well as English to French. Moreover, the Transformer required less computation to train, and also improved the training time by as much as an order of magnitude.

The Transformer can process the sentence "I arrived at the bank after crossing the river" and correctly determine that the word "bank" refers to the shore of a river and not a financial institution. The Transformer makes this determination in a single step by making the association between "bank" and "river." As another example, the Transformer can determine the different meanings of “it” in these two sentences:

“The horse did not cross the street because it was too tired.”

“The horse did not cross the street because it was too narrow.”

The Transformer computes the next representation for a given word by comparing the word to every other word in the sentence, which results in an "attention score" for the words in the sentence. The Transformer uses these scores to determine the extent to which other words will contribute to the next representation of a given word.

The result of these comparisons is an attention score for every other word in the sentence. As a result, "river" received a high attention score when computing a new representation for "bank."

Although LSTMs and bidirectional LSTMs are heavily utilized in NLP tasks, the Transformer has gained a lot of traction in the AI community, not only for translation between languages, but also the fact that for some tasks it can outperform both RNNs and CNNs. The Transformer architecture requires much less computation time in order to train a model, which explain why some people believe that the Transformer has already begun to supplant RNNs and LSTMs.

The following link contains a TF 2 code sample of a Transformer neural network that you can launch in Google Colaboratory:

https://www.tensorflow.org/alpha/tutorials/text/transformer

Another interesting and recent architecture is called “attention augmented convolutional networks,” which is a combination of CNNs with self-attention. This combination achieves better accuracy than “pure” CNNs, and you can find more details in this paper: *https://arxiv.org/abs/1904.09925*

7.1.3 Transformer-XL Architecture

The `Transformer-XL` combines a `Transformer` architecture with two techniques called recurrence mechanism and relative positional encoding to obtain better results than a `Transformer`. `Transformer-XL` works with word-level and character-level language modeling.

The `Transformer-XL` and `Transformer` both process the first segment of tokens, and the former also keeps the outputs of the hidden layers. Consequently, each hidden layer receives two inputs from the previous hidden layer, and then concatenates them to provide additional information to the neural network.

According to the following article, `Transformer-XL` significantly outperforms `Transformer`, and its dependency is 80% longer than "vanilla" RNNs:

https://hub.packtpub.com/transformer-xl-a-google-architecture-with-80-longer-dependency-than-rnns/

7.1.4 Reformer Architecture

Recently the `Reformer` architecture was released, which uses two techniques to improve the efficiency (i.e., lower memory and faster performance on long sequences) of the `Transformer` architecture. As a result, the `Reformer` architecture also has lower complexity than the `Transformer`. More details regarding the `Reformer` are here:

https://openreview.net/pdf?id=rkgNKkHtvB

Some Reformer-related code is here: *https://pastebin.com/62r5FuEW*

7.1.5 NLP and Deep Learning

The `NLP` models that use deep learning can comprise `CNN`s, `RNN`s, `LSTM`s, and bi-directional `LSTM`s. For example, Google released `BERT` in 2018, which is an extremely powerful framework for `NLP`. `BERT` is quite sophisticated, and involves bidirectional transformers and so-called "attention" (discussed briefly later in this chapter).

Deep learning for NLP often yields higher accuracy than other techniques, but keep in mind that sometimes it's not as fast as rule-based and classical ML methods. In case you're interested, a code sample that uses TensorFlow and `RNN`s for text classification is here:

https://www.tensorflow.org/alpha/tutorials/text/text_classification_rnn

A code sample that uses TensorFlow and RNNs for text generation is here:

https://www.tensorflow.org/alpha/tutorials/text/text_generation

7.1.6 Data Preprocessing Tasks in NLP

There are some common preprocessing tasks that are performed on documents, as listed here:

- [1] lowercasing
- [1] noise removal
- [2] normalization
- [3] text enrichment
- [3] stopword removal
- [3] stemming
- [3] lemmatization

The preceding tasks can be classified as follows:

- [1]: mandatory tasks
- [2]: recommended tasks
- [3]: task dependent

In brief, preprocessing tasks involve at least the removal of redundant words ("a," "the," and so forth), removing the endings of words ("running," "runs," and "ran" are treated the same as "run"), and converting text from uppercase to lowercase.

7.2 Popular NLP Algorithms

Some of the popular NLP algorithms appear in the following list, and in some cases they are the foundation for more sophisticated NLP toolkits:

- BoW: Bag of Words
- n-grams and skip-grams
- TF-IDF: basic algorithm in extracting keywords
- Word2Vector (Google): O/S project to describe text

- GloVe (Stanford NLP Group)
- LDA: text classification
- CF (collaborative filtering): an algorithm in news recommend system (Google News and Yahoo News)

The topics in the first half of the preceding list are discussed briefly in subsequent sections.

7.2.1 What is an n-gram?

An n-gram is a technique for creating a vocabulary that is based on adjacent words that are grouped together. This technique retains some word positions (unlike BoW). You need to specify the value of "n" that in turn specifies the size of the group.

The idea is simple: for each word in a sentence, construct a vocabulary term that contains the n words on the left side of the given word and n words that are on the right side of the given word. As a simple example, "This is a sentence" has the following 2-grams:

```
(this, is), (is, a), (a, sentence)
```

As another example, we can use the same sentence "This is a sentence" to determine its 3-grams:

```
(this, is, a), (is, a, sentence)
```

The notion of n-grams is surprisingly powerful, and it’s used heavily in popular open source toolkits such as `ELMo` and `BERT` when they pre-train their models.

7.2.2 What is a skip-gram?

Given a word in a sentence, a skip gram creates a vocabulary term by constructing a list that contains the n words on both sides of a given word, followed by the word itself. For example, consider the following sentence:

```
the quick brown fox jumped over the lazy dog
```

A skip-gram of size 1 yields the following vocabulary terms:

```
([the,brown], quick), ([quick,fox], brown),
([brown,jumped], fox),...
```

A skip-gram of size 2 yields the following vocabulary terms:

```
([the,quick,fox,jumped], brown),
([quick,brown,jumped,over], fox), ([brown,fox,over,the],
jumped),...
```

More details regarding skip-grams are discussed here:

https://www.tensorflow.org/tutorials/representation/word2vec#the_skip-gram_model

7.2.3 What is BoW?

BoW (Bag of Words) assigns a numeric value to each word in a sentence and treats those words as a set (or bag). Hence, BoW does not keep track of adjacent words, so it's a very simple algorithm.

Listing 7.1 displays the contents of the Python script `bow_to_vector.py` that illustrates how to use the BoW algorithm.

Listing 7.1: bow_to_vector.py

```
VOCAB = ['dog', 'cheese', 'cat', 'mouse']
TEXT1 = 'the mouse ate the cheese'
TEXT2 = 'the horse ate the hay'

 def to_bow(text):
  words = text.split(" ")
  return [1 if w in words else 0 for w in VOCAB]

print("VOCAB: ",VOCAB)
print("TEXT1:",TEXT1)
print("BOW1: ",to_bow(TEXT1))  # [0, 1, 0, 1]
print("")

print("TEXT2:",TEXT2)
print("BOW2: ",to_bow(TEXT2))  # [0, 0, 0, 0]
```

Listing 7.1 initializes a list `VOCAB` and two text strings `TEXT1` and `TEXT2`. The next portion of Listing 7.1 defines the Python function `to_bow()` that returns an array containing 0s and 1s: if a word in the current sentence appears in the vocabulary, then a 1 is returned (otherwise a 0 is returned). The last portion of Listing 7.1 invokes the Python function with two different sentences. The output from launching the code in Listing 7.1 is here:

```
('VOCAB: ', ['dog', 'cheese', 'cat', 'mouse'])
('TEXT1:', 'the mouse ate the cheese')
('BOW1: ', [0, 1, 0, 1])

('TEXT2:', 'the horse ate the hay')
('BOW2: ', [0, 0, 0, 0])
```

7.2.4 What is Term Frequency?

Term frequency is the number of times that a word appears in a document, which can vary among different documents. Consider the following simple example that consists of two "documents" `Doc1` and `Doc2`:

```
Doc1 = "This is a short sentence"
Doc2 = "yet another short sentence"
```

The term frequency for the word "is" and the word "short" is given here:

```
tf(is) = 1/5 for doc1
tf(is) = 0 for doc2
tf(short) = 1/5 for doc1
tf(short) = 1/4 for doc2
```

The preceding values will be used in the calculation of `tf-idf` that is explained in a later section.

7.2.5 What is Inverse Document Frequency (idf)?

Given a set of N documents and given a word in a document, let's define `dc` and `idf` of each word as follows:

```
dc = # of documents containing a given word
idf = log(N/dc)
```

Now let's use the same two documents Doc1 and Doc2 from a previous section:

```
Doc1 = "This is a short sentence"
Doc2 = "yet another short sentence"
```

The calculations of the `idf` value for the word "is" and the word "short" are shown here:

```
idf(is) = log(2/1) = log(2)
idf(short) = log(2/2) = 0
```

The following link provides more detailed information about inverse document frequency: https://en.wikipedia.org/wiki/Tf–idf#Example_of_tf–idf

7.2.6 What is `tf-idf`?

The term `tf-idf` is an abbreviation for "term frequency, inverse document frequency," and it's the product of the `tf` value and the `idf` value of a word, as shown here:

```
tf-idf = tf * idf
```

A high frequency word has a higher `tf` value but a lower `idf` value. In general, "rare" words are more relevant than "popular" ones, so they help to extract "relevance." For example, suppose you have a collection of 10 documents (real documents, not the toy documents we used earlier). The word “the” occurs frequently in English sentences, but it does not provide any indication of the topics in any of the documents. On the other hand, if you determine that the word “universe” appears multiple times in a single document, this information can provide some indication of the theme of that document, and with the help of NLP techniques, assist in determining the topic (or topics) in that document.

7.3 What are Word Embeddings?

An *embedding* is a fixed-length vector to encode and represent an entity (document, sentence, word, graph). Each word is represented by a real-valued vector, which can result in hundreds of dimensions. Furthermore, such an encoding can result in sparse vectors: one example is one-hot encoding, where one position has the value 1 and all other positions have the value 0.

Three popular word embedding algorithms are Word2vec, GloVe, and FastText. Keep in mind that these three algorithms involve unsupervised approaches. They are also based on the distributional hypothesis: words in the same contexts tend to have similar meanings: *https://aclweb.org/aclwiki/Distributional_Hypothesis*.

A good article regarding Word2Vec in TensorFlow is here:

https://towardsdatascience.com/learn-word2vec-by-implementing-it-in-tensorflow-45641adaf2ac

This article is useful if you want to see Word2Vec with FastText in gensim:

https://towardsdatascience.com/word-embedding-with-word2vec-and-fasttext-a209c1d3e12c

Another good article, and this one pertains to the skip-gram model:

https://towardsdatascience.com/word2vec-skip-gram-model-part-1-intuition-78614e4d6e0b

A useful article that describes how FastText works “under the hood”:

https://towardsdatascience.com/fasttext-under-the-hood-11efc57b2b3

Along with the preceding popular algorithms there are also some popular embedding models, some of which are listed here:

- baseline averaged sentence embeddings
- Doc2Vec
- neural-net language models
- skip-thought vectors
- quick-thought vectors
- inferSent
- universal sentence encoder

Perform an online search for more information about the preceding embedding models.

7.4 ELMo, ULMFit, OpenAI, BERT, and ERNIE 2.0

During 2018 there were some significant advances in NLP-related research, resulting in the following toolkits and frameworks:

- `ELMo:      released in 02/2018`
- `ULMFit:    released in 05/2018`
- `OpenAI:    released in 06/2018`
- `BERT:      released in 10/2018`
- `MT-DNN:    released in 01/2019`
- `ERNIE 2.0: released in 08/2019`

`ELMo` is an acronym for "embeddings from language models," which provides deep contextualized word representations and state-of-the-art contextual word vectors, resulting in noticeable improvements in word embeddings.

Jeremy Howard and Sebastian Ruder created universal language model fine-tuning (ULMFit), which is a transfer learning method that can be applied to any task in NLP. ULMFit significantly outperforms the state-of-the-art on six text classification tasks, reducing the error by 18–24% on the majority of datasets.

Furthermore, with only 100 labeled examples, it matches the performance of training from scratch on 100x more data. ULMFit is downloadable from GitHub:

https://github.com/jannenev/ulmfit-language-model

OpenAI developed GPT-2 (a successor to GPT), which is a model that was trained to predict the next word in 40GB of Internet text. OpenAI chose not to release the trained model due to concerns regarding malicious applications of their technology.

GPT-2 is a large transformer-based language model with 1.5 billion parameters, trained on a dataset of 8 million Web pages (curated by humans), with an emphasis on diversity of content. GPT-2 is trained to predict the next word, given all of the previous words within some text. The diversity of the dataset causes this goal to contain naturally occurring demonstrations of many tasks across diverse domains. GPT-2 is a direct scale-up of GPT, with more than 10X the parameters and trained on more than 10X the amount of data.

`BERT` is an acronym for "bidirectional encoder representations from transformers." `BERT` can pass this simple English test (i.e., `BERT` can determine the correct choice among multiple choices):

```
On stage, a woman takes a seat at the piano. She:

a) sits on a bench as her sister plays with the doll.

b) smiles with someone as the music plays.

c) is in the crowd, watching the dancers.

d) nervously sets her fingers on the keys.
```

Details of `BERT` and this English test are here:

https://www.lyrn.ai/2018/11/07/explained-bert-state-of-the-art-language-model-for-nlp/

The `BERT` (TensorFlow) source code is available here on GitHub:

https://github.com/google-research/bert

https://github.com/hanxiao/bert-as-service

Another interesting development is MT-DNN from Microsoft, which asserts that MT-DNN can outperform Google `BERT`:

https://medium.com/syncedreview/microsofts-new-mt-dnn-outperforms-google-bert-b5fa15b1a03e

A Jupyter notebook with `BERT` is available, and you need the following in order to run the notebook in Google Colaboratory:

- a GCP (Google Compute Engine) account
- a GCS (Google Cloud Storage) bucket

Here is the link to the notebook in Google Colaboratory:

https://colab.research.google.com/github/tensorflow/tpu/blob/master/tools/colab/bert_finetuning_with_cloud_tpus.ipynb

In March, 2019 Baidu open sourced ERNIE 1.0 (Enhanced Representation through kNowledge IntEgration) that (according to Baidu) outperformed BERT in tasks involving Chinese language understanding. In August, 2019 Baidu open sourced ERNIE 2.0, which is downloadable here:

https://github.com/PaddlePaddle/ERNIE/

An article with additional information about ERNIE 2.0 (including its architecture) is here:

https://hub.packtpub.com/baidu-open-sources-ernie-2-0-a-continual-pre-training-nlp-model-that-outperforms-bert-and-xlnet-on-16-nlp-tasks/

7.5 What is Translatotron?

Translatotron is an end-to-end speech-to-speech translation model (from Google) whose output retains the original speaker's voice; moreover it's trained with less data.

Speech-to-speech translation systems have been developed over the past several decades with the goal of helping people who speak different languages to communicate with each other. Such systems have three parts:

- automatic speech recognition to transcribe the source speech as text
- machine translation to translate the transcribed text into the target language
- text-to-speech synthesis (TTS) to generate speech in the target language from the translated text

The preceding approach has been successful in commercial products (including Google Translate). However, Translatatron does not require separate stages, resulting in the following advantages:

- faster inference speed
- avoids compounding errors between recognition and translation
- easier to retain the voice of the original speaker after translation
- better handling of untranslated words (names and proper nouns)

This concludes the portion of this chapter that pertains to NLP. Another area of great interest in the AI community is RL, which is introduced later in this chapter.

7.6 Deep Learning and NLP

In Chapter 4, you learned about CNNs and how they are well-suited for image classification tasks. You might be surprised to discover that CNNs also work with NLP tasks. However, you must first "map" each word in a dictionary (which can be a subset of the words in English or some other language) to numeric values and then construct a vector of numeric values from the words in a sentence. A document can be transformed into a set of numeric vectors (involving various techniques that are not discussed here) in order to create a dataset that's suitable for input to a `CNN`.

Another option involves the use of `RNN`s and `LSTM`s instead of `CNN`s for `NLP`-related tasks. In fact, a "bidirectional LSTM" is being used successfully in ELMo (Embeddings from Language Models), whereas `BERT` is based on a bi-directional transformer architecture. The Google AI team developed `BERT` (open sourced in 2018) and it's considered a breakthrough in its ability to solve `NLP` problems. The source code is here: *https://github.com/google-research/bert*

7.7 NLU versus NLG

NLU is an acronym for natural language understanding. NLU pertains to machine reading comprehension, and it's considered a difficult problem. At the same time, NLU is relevant to machine translation, question answering, and text categorization (among others). NLU attempts to discern the meaning of fragmented sentences and run-on sentences, after which some type of action can be performed (e.g., respond to voice queries).

NLG is an acronym for natural language generation, which involves generating documents. The Markov chain (discussed later in this chapter) was one of the first algorithms for NLG. Another technique involves RNNs (discussed in Chapter 5) that can retain some history of previous words, and the probability of the next word in a sequence is calculated. Recall that RNNs suffer from limited memory, which limits the length of the sentences that can be generated. A third technique involves LSTMs, which can maintain state for a long period of time, and also avoid the "exploding gradient" problem.

Recently (circa 2017) Google introduced the transformer architecture, which involves a stack of encoders for processing inputs and a set of decoders to produce generated sentences. A transformer-based architecture is more efficient than an LSTM because a transformer requires a small and fixed number of steps in order to apply the so-called "self-attention mechanism" in order to simulate the relationship among all the words in a sentence.

In fact, the transformer differs from previous models in one important way: it uses the representation of all words in context without compressing all the information into a single fixed-length representation. This technique enables a transformer to handle longer sentences without high computational costs.

The transformer architecture is the foundation for the GPT-2 language model (from OpenAI). The model learns to predict the next word in a sentence by focusing on words that were previously seen in the model and related to predicting the next word. In 2018, Google released the BERT architecture for NLP, which is based on transformers with a two-way encoder representation.

7.8 What is Reinforcement Learning (RL)?

RL is a subset of machine learning that attempts to find the maximum reward for a so-called "agent" that interacts with an "environment." RL is suitable for solving tasks that involve deferred rewards, especially when those rewards are greater than intermediate rewards.

In fact, RL can handle tasks that involve a combination of negative, zero, and positive rewards. For example, if you decide to leave your job in order to attend school on a full-time basis, you are spending money (a negative reward) with the believe that your investment of time and money will lead to a higher paying position (a positive reward) that outweighs the cost of school and lost earnings.

One thing that might surprise you is that RL agents are susceptible to `GAN`s. Chapter 5 contains a section devoted to `GAN`s, and you can find additional details (along with related links) in this article:

https://openai.com/blog/adversarial-example-research/

7.8.1 RL Applications

There are many RL applications, some of which are listed here:

- game theory
- control theory
- operations research
- information theory
- simulation-based optimization
- multi-agent systems
- swarm intelligence
- statistics and genetic algorithms
- resources management in computer clusters
- traffic light control (congestion problems)
- robotics operations
- autonomous cars/helicopters
- Web system Configuration/Web-page indexing
- personalized recommendations
- bidding and advertising
- robot legged locomotion
- marketing strategy selection
- factory control

RL refers to goal-oriented algorithms for reaching a complex goal, such as winning games that involve multiple moves (e.g., chess or Go). RL algorithms are penalized for incorrect decisions and rewarded for correct decisions: this reward mechanism is reinforcement.

7.8.2 NLP and RL

More recently RL with `NLP` has become a successful area of research. One technique for `NLP`-related tasks involves `RNN`-based encoder-decoder models that have achieved good results for short input and output sequences. Another technique involves a neural network, supervised word prediction, and RL. This particular combination avoids exposure bias, which can occur in models that use only supervised learning. More details are here: *https://arxiv.org/pdf/1705.04304.pdf*

Yet another interesting technique involves deep reinforcement learning (i.e., DL combined with RL) with NLP. In case you don't already know, DRL has achieved success in various areas, such as Atari games, defeating Lee Sedol (the world champion Go player), and robotics. In addition, DRL is also applicable to NLP-related tasks, which involves the key challenge of designing of a suitable model. Perform an online search for more information about solving NLP-related tasks with RL and DRL.

7.8.3 Values, Policies, and Models in RL

There are three main approaches in RL. *value-based* RL estimates the optimal value function `Q(s,a)`, which is the maximum value achievable under any policy. *Policy-based* RL searches directly for the optimal policy π, which is the policy achieving maximum future reward. *Model-based* RL builds a model of the environment and plans (by lookahead) using the model.

In addition to the preceding approaches to RL (value functions, policies, and models), you will need to learn the following RL concepts:

- Markov decision processes (MDPs)
- A policy (a sequence of actions)
- The state/value function
- The action/value function
- Bellman equation (for calculating rewards)

The RL material in this chapter only addresses the following list of topics (after which you can learn the concepts in the previous list):

- Nondeterministic finite automata (NFAs)
- Markov chains
- MDPs

- Epsilon-Greedy algorithm
- Bellman equation

Another key point: almost all RL problems can be formulated as *Markov decision processes*, (MDPs) which in turn are based on Markov chains. Let's take a look at `NFA`s and Markov chains and then we can define MDPs.

7.9 From NFAs to MDPs

Let's start with the two-minute summary. The underlying structure for an MDP is an NFA (nondeterministic finite automata), which is studied in great detail in an automata theory course (as part of a computer science degree). An NFA is a collection of states and transitions, each of which has equal probability. An NFA also has a start state and one or more end states.

Now add probabilities to transitions in an NFA, in such a way that the sum of the probabilities of the outgoing transitions of any state equals one. The result is a Markov chain. A Markov decision process is a Markov chain with several additional properties.

The following subsections expand the two-minute summary by providing additional explanatory details.

7.9.1 What Are `NFA`s?

An `NFA` is a nondeterministic finite automata, which is a generalization of a `DFA` (deterministic finite automata). Figure 7.1 displays an example of an NFA.

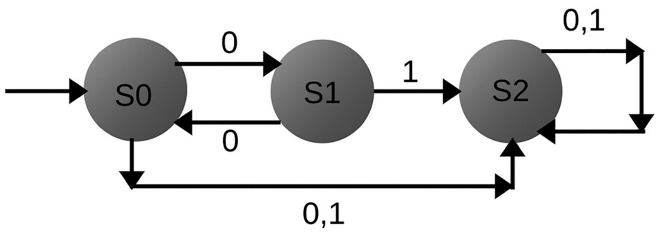

FIGURE 7.1: An example of an `NFA`. Image adapted from *https://math.stackexchange.com/questions/1240601/what-is-the-easiest-way-to-determine-the-accepted-language-of-a-deterministic-fi?rq=1*.

An `NFA` enables you to define multiple transitions from a given state to other states. By way of analogy, consider the location of many (most?) gas stations. Usually they are located at an intersection of two streets, which means there are at least two entrances to the gas station. After you make your purchase, you can exit from the same entrance or from the second entrance. In some cases, you might even be able to exit from one location and return to the gas station from the other entrance: this would be comparable to a "loop" transition of a state in a state machine.

The next step involves adding probabilities to `NFA`s in order to create a Markov Chain, which is described in more detail in the next section.

7.9.2 What Are Markov Chains?

Markov Chains are `NFA`s with an additional constraint: the sum of the probabilities of the outgoing edges of every state equals one. Figure 7.2 displays a Markov chain.

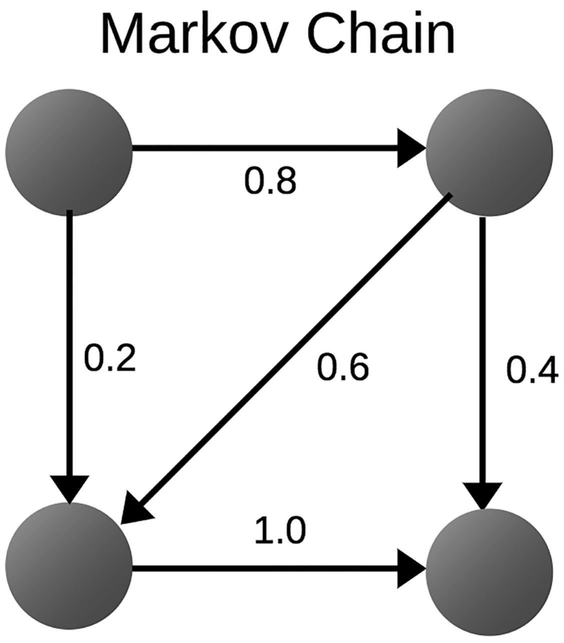

FIGURE 7.2: An example of a Markov chain. Image adapted from *https://en.wikipedia.org/wiki/Markov_chain*.

As you can see in Figure 7.2, a Markov chain is an `NFA` because a state can have multiple transitions. The constraint involving probabilities ensures that we can perform statistical sampling in `MDP`s that are described in the next section.

7.9.3 MDPs

In high-level terms, an MDP is a method that samples from a complex distribution to infer its properties. More specifically, `MDPs` are an extension of Markov chains, which involves the addition of actions (allowing choice) and rewards (giving motivation). Conversely, if only one action exists for each state (e.g., "wait") and all rewards are the same (e.g., "zero"), an `MDP` reduces to a Markov chain. Figure 7.3 displays an example of an `MDP`.

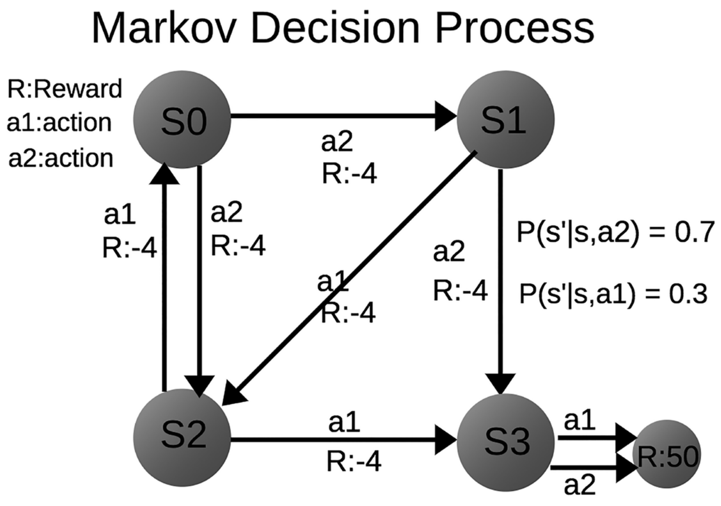

FIGURE 7.3: An example of an `MDP`.

Thus, an `MDP` consists of a set of states and actions, and also the rules for transitioning from one state to another. One episode of this process (e.g., a single "game") produces a finite sequence of states, actions, and rewards. A key property of `MDPs`: history does not affect future decisions. In other words, the process of selecting the next state is independent of everything that happened before reaching the current state.

`MDPs` are nondeterministic search problems that are solved via dynamic programming and RL, where outcomes are partly random and partly under control. As you learned earlier in this section, almost all RL problems can be formulated as `MDPs`; consequently, RL can solve tasks that cannot be solved by greedy algorithms. However, the epsilon-greedy algorithm is a clever algorithm that *can* solve such tasks. In addition, the Bellman equation enables us to compute rewards for states. Both are discussed in subsequent sections.

7.10 The Epsilon-Greedy Algorithm

There are three fundamental problems that arise in RL:

- the exploration-exploitation tradeoff
- the problem of delayed reward (credit assignment)
- the need to generalize

The term *exploration* refers to trying something new or different, whereas the term *exploitation* refers to leveraging existing knowledge or information. For instance, going to a favorite restaurant is an example of exploitation (you are "exploiting" your knowledge of good restaurants), whereas going to an untried restaurant is an example of exploration (you are "exploring" a new venue). When people move to a new city, they tend to explore new restaurants, whereas people who are moving away from a city tend to exploit their knowledge of good restaurants.

In general, exploration refers to making random choices, whereas exploitation refers to using a greedy algorithm. The epsilon-greedy algorithm is an example of exploration and exploitation, where the "epsilon" portion of the algorithm refers to making random selections, and "exploitation" involves a greedy algorithm.

An example of a simple task that can be solved via the epsilon-greedy algorithm is Open AI Gym's NChain environment, as shown in Figure 7.4.

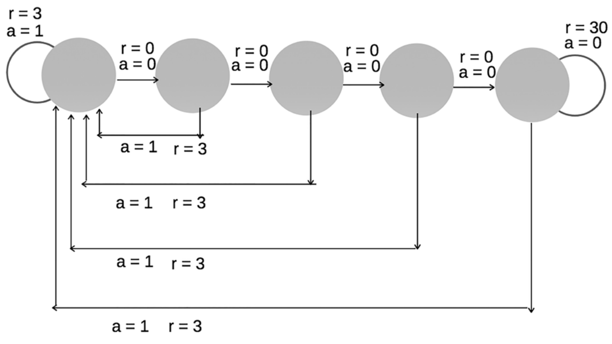

FIGURE 7.4: The Open AI Gym's NChain environment.

Image adapted from [*http://ceit.aut.ac.ir/~shiry/lecture/machine-learning/papers/BRL-2000.pdf*]

Each state in Figure 7.4 has two actions, and each action has an associated reward. For each state, its "forward" action has reward 0, whereas its "backward" action has reward 3. Since a greedy algorithm will always select the larger reward at any state, this means that the "backward" action is always selected. Hence, we can never move toward the final state 4 that has a reward of 10. Indeed, we can never leave state 0 (the initial state) if we adhere to the greedy algorithm.

Here is the key question: how do we go from the initial state 0 to the final state, which contains a large reward? *We need a modified or hybrid algorithm in order to go through intermediate low-reward states that lead to the high reward state.*

The hybrid algorithm is simple to describe: adhere to the greedy algorithm about 90% of the time and randomly select a state for the remaining 10% of the time. This technique is simple, elegant, and effective, and it's called the *epsilon-greedy* algorithm (there are additional details required for a complete implementation).

Incidentally, a Python-based solution for OpenAI's NChain task is here:

https://github.com/openai/gym/blob/master/gym/envs/toy_text/nchain.py

Another central concept in RL involves the Bellman equation, which is the topic of the next section.

7.11 The Bellman Equation

The Bellman equations are named after Richard Bellman who derived these equations that are ubiquitous in RL. There are several Bellman equations, including one for the state value function and one for the action value function. Figure 7.5 displays the Bellman equation for the state value function.

$$V^{\pi}(s) = E_{\pi}[\sum_{k=0}^{\infty} \gamma^{k} r_{t+k+1} | s_t = s]$$

FIGURE 7.5: The Bellman equation.

As you can see in Figure 7.5, the value of a given state depends on the discounted value of future states. The following analogy might help you understand the purpose of the discounted value called *gamma* in this equation. Suppose that you have USD 100 that you invest at a 5% annual interest rate. After one year you will have USD 105 (`=100 + 5%*100 = 100*(1+0.05)`), after two years you will have USD 110.25 (`=100*(1+0.05)*(1+0.05)`), and so forth.

Conversely, if you have a future value of USD 100 (with a 5% annual investment rate) that is two years in the future, what is its present value? The answer involves dividing 100 by powers of `(1+0.05)`. Specifically, the present value of USD 100 from two years in the future equals `100/[(1+0.05)*(1+0.05)]`.

In analogous fashion, the Bellman equation enables us to calculate the current value of a state by calculating the discounted reward of subsequent states. The discount factor is called *gamma*, and it's often a value between 0.9 and 0.99. In the preceding example involving USD 100, the value of gamma is 0.9523.

7.11.1 Other Important Concepts in RL

After you have studied the basic concepts in RL, you can delve into the topics that are listed here:

- policy gradient (rules for "best" actions)
- Q-value
- Monte Carlo
- dynamic programming
- temporal difference (TD)
- Q-learning
- Deep Q network

The preceding topics are explained in online articles (suggestion: use Wikipedia as a starting point for RL concepts), and they will be much more relevant after you grasp the introductory concepts in RL that are discussed in earlier sections. Be prepared to spend some time learning these topics because some of them are quite challenging in nature.

7.12 RL Toolkits and Frameworks

There are many toolkits and libraries for RL, typically based on Python, Keras, Torch, or Java. Some of them are listed here:

- OpenAI gym: A toolkit for developing and comparing RL algorithms
- OpenAI universe: A software platform for measuring and training an AI's general intelligence across the world's supply of games, websites and other applications
- DeepMind Lab: A customisable 3D platform for agent-based AI research
- rllab: A framework for developing and evaluating RL algorithms, fully compatible with OpenAI Gym
- TensorForce: Practical deep RL on TensorFlow with Gitter support and OpenAI Gym/Universe/DeepMind Lab integration
- tf-TRFL: A library built on top of TensorFlow that exposes several useful building blocks for implementing RL agents
- OpenAI lab: An experimentation system for RL using OpenAI Gym, Tensorflow, and Keras
- MAgent: A Platform for Many-agent RL
- Intel Coach: Coach is a python RL research framework containing implementation of many state-of-the-art algorithms</BL>

As you can see from the preceding list, there is a considerable variety of available RL toolkits, and visit their homepages to determine which ones have the features that meet your specific requirements.

7.12.1 TF-Agents

Google created the TF-Agents library for RL in TensorFlow. Google TF-Agents is open source and downloadable from Github:

https://github.com/tensorflow/agents

The core elements of RL algorithms are implemented as *agents*. An agent encompasses two main responsibilities: defining a *policy* to interact with the *environment*, and how to learn/train that policy from collected experience. TF-Agents implements the following algorithms:

- DQN: Human level control through deep RL Mnih et al., 2015
- DDQN: Deep RL with Double Q-learning Hasselt et al., 2015
- DDPG: Continuous control with deep RL Lillicrap et al., 2015
- TD3: Addressing Function Approximation Error in Actor-Critic Methods Fujimoto et al., 2018
- REINFORCE: Simple Statistical Gradient-Following Algorithms for Connectionist RL Williams, 1992
- PPO: Proximal Policy Optimization Algorithms Schulman et al., 2017
- SAC: Soft Actor Critic Haarnoja et al., 2018

Before you can use TF-Agents, first install the nightly build version of TF-Agents with this command (`pip` or `pip3`):

```
# the --upgrade flag ensures you'll get the latest
version
pip install --user --upgrade tf-nightly
pip install --user --upgrade tf-agents-nightly #
requires tf-nightly
```

There are "end-to-end" examples training agents under each agent directory, an example of which is here for DQN:

```
tf_agents/agents/dqn/examples/v1/train_eval_gym.py
```

Keep in mind that TF-Agents is in prerelease status and therefore under active development, which means that interfaces may change at any time.

7.13 What is Deep RL (DRL)?

Deep RL is a surprisingly effective combination of deep learning and RL that has shown remarkable results in a variety of tasks. For example, DRL has won game competitions such as Go (Alpha Go versus world champion Lee Sedol) and even prevailed in the complexity of StarCraft (AlphaStar of DeepMind) and Dota.

With the release of ELMo and `BERT` in 2018 (discussed earlier in this chapter), DRL made significant advances in `NLP` with these toolkits, surpassing previous benchmarks in `NLP`.

Google released the Dopamine toolkit for DRL, which is downloadable here from GitHub: *https://github.com/google/dopamine*.

The `keras-rl` toolkit supports state-of-the-art Deep RL algorithms in Keras, which are also designed for compatibility with OpenAI (discussed earlier in this chapter). This toolkit includes the following:

- Deep Q Learning (DQN)
- Double DQN
- Deep Deterministic Policy Gradient (DDPG)
- Continuous DQN (CDQN or NAF)
- Cross-Entropy Method (CEM)
- Dueling network DQN (Dueling DQN)
- Deep SARSA
- Asynchronous Advantage Actor-Critic (A3C)
- Proximal Policy Optimization Algorithms (PPO)

Please keep in mind that the details of the algorithms in the preceding list require a decent understanding of RL. The `keras-rl` toolkit is downloadable here from GitHub: *https://github.com/keras-rl/keras-rl*

7.14 Summary

This chapter introduced you to NLP, along with some code samples in Keras, as well as NLU and NLG. In addition, you learned about some basic concepts in `NLP`, such as n-grams, BoW, tf-idf, and word embeddings.

Then you got an introduction to RL, along with a description of the types of tasks that are well-suited to RL. You will learn about the nchain task and the epsilon-greedy algorithm that can solve problems that you cannot solve using a "pure" greedy algorithm. You also learned about the Bellman equation, which is a cornerstone of RL.

Next, you were exposed to the TF-Agents toolkit from Google, deep RL (deep learning combined with RL), and the Google Dopamine toolkit.

Congratulations! You have reached the end of this book, which has covered many ML concepts. You also learned about Keras, as well as linear regression, logistic regression, and deep learning. You are now in a good position to delve further into ML algorithms or proceed with deep learning, and good luck in your journey!

APPENDIX A

INTRODUCTION TO REGULAR EXPRESSIONS

- What Are Regular Expressions?
- Meta Characters in Python
- Character Sets in Python
- Character Classes in Python
- Matching Character Classes with the re Module
- Using the `re.match()` Method
- Options for the `re.match()` Method
- Matching Character Classes with the `re.search()` Method
- Matching Character Classes with the `findAll()` Method
- Additional Matching Function for Regular Expressions
- Grouping with Character Classes in Regular Expressions
- Using Character Classes in Regular Expressions
- Modifying Text Strings with the re Module
- Splitting Text Strings with the `re.split()` Method
- Splitting Text Strings Using Digits and Delimiters
- Substituting Text Strings with the `re.sub()` Method
- Matching the Beginning and the End of Text Strings
- Compilation Flags
- Compound Regular Expressions

This appendix introduces you to regular expressions, which is a very powerful language feature in Python. Since regular expressions are available in other programming languages (such as JavaScript and Java), the knowledge that you gain from the material in this appendix will be useful to you outside of Python. This appendix contains a mixture of code blocks and complete code samples, with varying degrees of complexity, that are suitable for beginners as well as people who have had some exposure to regular expressions. In fact, you have probably used (albeit simple) regular expressions in a command line on a laptop, whether it be Windows, Unix, or Linux-based systems. In this appendix you will learn how to define and use more complex regular expressions than the regular expressions that you have used from the command line. Recall that in Chapter 1 you learned about some basic metacharacters, and you can use them as part of regular expressions in order to perform sophisticated search-and-replace operations involving text strings and text files.

The first part of this appendix shows you how to define regular expressions with digits and letters (uppercase as well as lowercase), and also how to use character classes in regular expressions. You will also learn about character sets and character classes. The second portion discusses the Python `re` module, which contains several useful methods, such as the `re.match()` method for matching groups of characters, the `re.search()` method to perform searches in character strings, and the `findAll()` method. You will also learn how to use character classes (and how to group them) in regular expressions.

The final portion of this appendix contains an assortment of code samples, such as modifying text strings, splitting text strings with the `re.split()` method, and substituting text strings with the `re.sub()` method.

As you read the code samples in this appendix, some concepts and facets of regular expressions might make you feel overwhelmed with the density of the material if you are a novice. However, practice and repetition will help you become more comfortable with regular expressions.

Finally, please keep in mind that the code samples were originally written for Python 2.7.5, which means that you might encounter some code samples that need to be updated to work in Python 3.x.

A.1 What Are Regular Expressions?

Regular expressions are referred to as REs, or regexes, or regex patterns, and they enable you to specify expressions that can match specific "parts" of a string. For instance, you can define a regular expression to match a single character or digit, a telephone number, a zip code, or an email address. You can use metacharacters and character classes (defined in the next section) as part of regular expressions to search text documents for specific patterns. As you learn how to use RE you will find other ways to use them as well.

The `re` module (added in Python 1.5) provides Perl-style regular expression patterns. Note that earlier versions of Python provided the `regex` module that was removed in Python 2.5. The `re` module provides an assortment of methods (discussed later in this appendix) for searching text strings or replacing text strings, which is similar to the basic search and/or replace functionality that is available in word processors (but usually without regular expression support). The `re` module also provides methods for splitting text strings based on regular expressions.

Before delving into the methods in the `re` module, you need to learn about metacharacters and character classes, which are the topic of the next section.

A.2 Metacharacters in Python

Python supports a set of metacharacters, most of which are the same as the metacharacters in other scripting languages such as Perl, as well as programming languages such as JavaScript and Java. The complete list of metacharacters in Python is here:

```
. ^ $ * + ? { } [ ] \ | ( )
```

The meaning of the preceding metacharacters is here:

`?` (matches 0 or 1): the expression `a?` matches the string `a` (but not `ab`)

`*` (matches 0 or more): the expression `a*` matches the string `aaa` (but not `baa`)

+ (matches 1 or more): the expression a+ matches aaa (but not baa)

`^` (beginning of line): the expression `^[a]` matches the string `abc` (but not `bc`)

`$` (end of line): `[c]$` matches the string `abc` (but not `cab`)

`.` (a single dot): matches any character (except newline)

Sometimes you need to match the metacharacters themselves rather than their representation, which can be done in two ways. The first way involves "escaping" their symbolic meaning with the backslash ("\") character. Thus, the sequences `\?, \*, \+, \^, \$, and \.` represent the literal characters instead of their symbolic meaning. You can also "escape" the backslash character with the sequence "`\\`". If you have two consecutive backslash characters, you need an additional backslash for each of them, which means that "`\\\\`" is the "escaped" sequence for "`\\`".

The second way is to list the metacharacters inside a pair of square brackets. For example, `[+?]` treats the two characters "+" and "?" as literal characters instead of metacharacters. The second approach is obviously more compact and less prone to error (it's easy to forget a backslash in a long sequence of metacharacters). As you might surmise, the methods in the `re` module support metacharacters.

Note: The "`^`" character that is to the left (and outside) of a sequence in square brackets (such as `^[A-Z]`) "anchors" the regular expression to the beginning of a line, whereas the "^" character that is the first character inside a pair of square brackets *negates* the regular expression (such as `[^A-Z]`) inside the square brackets.

The interpretation of the "`^`" character in a regular expression depends on its location in a regular expression, as shown here:

- "`^[a-z]`" means any string that starts with any lowercase letter
- "`[^a-z]`" means any string that does *not* contain any lowercase letters
- "`^[^a-z]`" means any string that starts with anything *except* a lowercase letter
- "`^[a-z]$`" means a single lowercase letter
- "`^[^a-z]$`" means a single character (including digits) that is *not* a lowercase letter

As a quick preview of the `re` module that is discussed later in this appendix, the `re.sub()` method enables you to remove characters (including metacharacters) from a text string. For example, the following code snippet removes all occurrences of a forward slash ("/") and the plus sign ("+") from the variable `str`:

```
>>> import re
>>> str  = "this string has a / and + in it"
>>> str2 = re.sub("[/]+","",str)
>>> print 'original:',str
original: this string has a / and + in it
>>> print 'replaced:',str2
replaced: this string has a  and + in it
```

We can easily remove occurrences of other metacharacters in a text string by listing them inside the square brackets, just as we have done in the preceding code snippet.

Listing A.1 displays the contents of `RemoveMetaChars1.py` that illustrates how to remove other metacharacters from a line of text.

Listing A.1: RemoveMetaChars1.py

```
import re

text1 = "meta characters ? and / and + and ."
text2 = re.sub("[/\.*?=+]+","",text1)

print 'text1:',text1
print 'text2:',text2
```

The regular expression in Listing A.1 might seem daunting if you are new to regular expressions, but let's demystify its contents by examining the entire expression and then the meaning of each character. First of all, the term `[/\.*?=+]` matches a forward slash ("/"), a dot ("."), a question mark ("?"), an equals sign ("="), or a plus sign ("+"). Notice that the dot "." is preceded by a backslash character "\". Doing so "escapes" the meaning of the "." metacharacter (which matches any single nonwhitespace character) and treats it as a literal character.

Thus the term `[/\.*?=+]+` means "one or more occurrences of any of the metacharacters—treated as literal characters—inside the square brackets."

Consequently, the expression `re.sub("[/\.*?=+]+","",text1)` matches any occurrence of the previously listed metacharacters, and then replaces them with an empty string in the text string specified by the variable `text1`. The output from Listing A.1 is here:

```
text1: meta characters ? and / and + and .
text2: meta characters  and  and  and
```

Later in this appendix you will learn about other functions in the `re` module that enable you to modify and split text strings.

A.3 Character Sets in Python

A single digit in base 10 is a number between 0 and 9 inclusive, which is represented by the sequence `[0-9]`. Similarly, a lowercase letter can be any letter between a and z, which is represented by the sequence `[a-z]`. An uppercase letter can be any letter between `A` and `Z`, which is represented by the sequence `[A-Z]`.

The following code snippets illustrate how to specify sequences of digits and sequences of character strings using a shorthand notation that is much simpler than specifying every matching digit:

- `[0-9]` matches a single digit
- `[0-9][0-9]` matches 2 consecutive digits
- `[0-9]{3}` matches 3 consecutive digits
- `[0-9]{2,4}` matches 2, 3, or 4 consecutive digits
- `[0-9]{5,}` matches 5 or more consecutive digits
- `^[0-9]+$` matches a string consisting solely of digits

You can define similar patterns using uppercase or lowercase letters in a way that is much simpler than explicitly specifying every lowercase letter or every uppercase letter:

- `[a-z][A-Z]` matches a single lowercase letter that is followed by 1 uppercase letter
- `[a-zA-Z]` matches any upper- or lowercase letter
- [2] Working with "^" and "\"

The purpose of the "^" character depends on its context in a regular expression. For example, the following expression matches a text string that starts with a digit:

```
^[0-9].
```

However, the following expression matches a text string that does *not* start with a digit because of the "^" metacharacter that is at the beginning of an expression in square brackets as well as the "^" metacharacter that is to the left (and outside) the expression in square brackets (which you learned in a previous note):

```
^[^0-9]
```

Thus, the "^" character inside a pair of matching square brackets ("[]") negates the expression immediately to its right that is also located inside the square brackets.

The backslash ("\") allows you to "escape" the meaning of a metacharacter. Consequently, a dot "." matches a single character (except for whitespace characters), whereas the sequence "\." matches the dot "." character. Other examples involving the backslash metacharacter are here

- `\.H.*` matches the string `.Hello`
- `H.*` matches the string `Hello`
- `H.*\.` matches the string `Hello.`
- `.ell.` matches the string `Hello`
- `.*` matches the string `Hello`
- `\..*` matches the string `.Hello`

A.4 Character Classes in Python

Character classes are convenient expressions that are shorter and simpler than their "bare" counterparts that you saw in the previous section. Some convenient character sequences that express patterns of digits and letters are as follows:

- `\d` matches a single digit
- `\w` matches a single character (digit or letter)
- `\s` matches a single whitespace (space, newline, return, or tab)
- `\b` matches a boundary between a word and a nonword

- `\n, \r, \t` represent a newline, a return, and a tab, respectively
- `\` "escapes" any character

Based on the preceding definitions, `\d+` matches one or more digits and `\w+` matches one or more characters, both of which are more compact expressions than using character sets. In addition, we can reformulate the expressions in the previous section:

- `\d` is the same as `[0-9]` and `\D` is the same as `[^0-9]`
- `\s` is the same as `[ \t\n\r\f\v]` and it matches any nonwhitespace character, whereas
- \S is the opposite (it matches `[^ \t\n\r\f\v]`)
- `\w` is the same as `[a-zA-Z0-9_]` and it matches any alphanumeric character, whereas `\W` is the opposite (it matches `[^a-zA-Z0-9_]`)

Additional examples are here:

- `\d{2}` is the same as `[0-9][0-9]`
- `\d{3}` is the same as `[0-9]{3}`
- `\d{2,4}` is the same as `[0-9]{2,4}`
- `\d{5,}` is the same as `[0-9]{5,}`
- `^\d+$` is the same as `^[0-9]+$`

The curly braces ("`{}`") are called quantifiers, and they specify the number (or range) of characters in the expressions that precede them.

A.5 Matching Character Classes with the `re` Module

The `re` module provides the following methods for matching and searching one or more occurrences of a regular expression in a text string:

- `match()`: Determine if the RE matches at the *beginning* of the string
- `search()`: Scan through a string, looking for *any* location where the RE matches
- `findall()`: Find *all* substrings where the RE matches and return them as a list

- `finditer()`: Find all substrings where the RE matches and return them as an iterator
- **Note**: The `match()` function only matches pattern to the start of string.

The next section shows you how to use the `match()` function in the `re` module.

A.6 Using the `re.match()` Method

The `re.match()` method attempts to match RE pattern in a text string (with optional flags), and it has the following syntax:

```
re.match(pattern, string, flags=0)
```

The `pattern` parameter is the regular expression that you want to match in the `string` parameter. The flags parameter allows you to specify multiple flags using the bitwise OR operator that is represented by the pipe "|" symbol.

The `re.match` method returns a match object on success and `None` on failure. Use `group(num)` or `groups()` function of the match object to get a matched expression.

`group(num=0)`: This method returns entire match (or specific subgroup `num`)

`groups()`: This method returns all matching subgroups in a tuple (empty if there weren't any)

Note: The `re.match()` method only matches patterns from the start of a text string, which is different from the `re.search()` method discussed later in this appendix.

The following code block illustrates how to use the `group()` function in regular expressions:

```
>>> import re
>>> p = re.compile('(a(b)c)de')
>>> m = p.match('abcde')
>>> m.group(0)
'abcde'
>>> m.group(1)
'abc'
>>> m.group(2)
```

```
'b'
```

Notice that the higher numbers inside the `group()` method match more deeply nested expressions that are specified in the initial regular expression.

Listing A.2 displays the contents of `MatchGroup1.py` that illustrates how to use the `group()` function to match an alphanumeric text string and an alphabetic string.

Listing A.2: MatchGroup1.py

```
import re

line1 = 'abcd123'
line2 = 'abcdefg'
mixed = re.compile(r"^[a-z0-9]{5,7}$")
line3 = mixed.match(line1)
line4 = mixed.match(line2)

print 'line1:',line1
print 'line2:',line2
print 'line3:',line3
print 'line4:',line4
print 'line5:',line4.group(0)

line6 = 'a1b2c3d4e5f6g7'
mixed2 = re.compile(r"^([a-z]+[0-9]+){5,7}$")
line7 = mixed2.match(line6)

print 'line6:',line6
print 'line7:',line7.group(0)
print 'line8:',line7.group(1)

line9 = 'abc123fgh4567'
mixed3 = re.compile(r"^([a-z]*[0-9]*){5,7}$")
line10 = mixed3.match(line9)
print 'line9:',line9
print 'line10:',line10.group(0)
```

The output from Listing A.2 is here:

```
line1: abcd123
line2: abcdefg
line3: <_sre.SRE_Match object at 0x100485440>
line4: <_sre.SRE_Match object at 0x1004854a8>
line5: abcdefg
```

```
line6: a1b2c3d4e5f6g7
line7: a1b2c3d4e5f6g7
line8: g7
line9: abc123fgh4567
line10: abc123fgh4567
```

Notice that `line3` and `line7` involve two similar but different regular expressions. The variable mixed specifies a sequence of lowercase letters followed by digits, where the length of the text string is also between 5 and 7. The string `'abcd123'` satisfies all of these conditions.

On the other hand, `mixed2` specifies a pattern consisting of one or more pairs, where each pair contains one or more lowercase letters followed by one or more digits, where the length of the matching pairs is also between 5 and 7. In this case, the string `'abcd123'`as well as the string `'a1b2c3d4e5f6g7'` both satisfy these criteria.

The third regular expression `mixed3` specifies a pair such that each pair consists of zero or more occurrences of lowercase letters and zero or more occurrences of a digit, and also that the number of such pairs is between 5 and 7. As you can see from the output, the regular expression in `mixed3` matches lowercase letters and digits in any order.

In the preceding example, the regular expression specified a range for the length of the string, which involves a lower limit of 5 and an upper limit of 7.

However, you can also specify a lower limit without an upper limit (or an upper limit without a lower limit).

Listing A.3 displays the contents of `MatchGroup2.py` that illustrates how to use a regular expression and the `group()` function to match an alphanumeric text string and an alphabetic string.

Listing A.3: MatchGroup2.py

```
import re

alphas = re.compile(r"^[abcde]{5,}")

line1 = alphas.match("abcde").group(0)
line2 = alphas.match("edcba").group(0)
line3 = alphas.match("acbedf").group(0)
line4 = alphas.match("abcdefghi").group(0)
line5 = alphas.match("abcdefghi abcdef")
```

```
print 'line1:',line1
print 'line2:',line2
print 'line3:',line3
print 'line4:',line4
print 'line5:',line5
```

Listing A.3 initializes the variable alphas as a regular expression that matches any string that starts with one of the letters a through e, and consists of at least 5 characters. The next portion of Listing A.3 initializes the 4 variables `line1`, `line2`, `line3`, and `line4` by means of the `alphas` RE that is applied to various text strings. These 4 variables are set to the first matching group by means of the expression `group(0)`.

The output from Listing A.3 is here:

```
line1: abcde
line2: edcba
line3: acbed
line4: abcde
line5: <_sre.SRE_Match object at 0x1004854a8>
```

Listing A.4 displays the contents of `MatchGroup3.py` that illustrates how to use a regular expression with the `group()` function to match words in a text string.

Listing A.4: MatchGroup3.py

```
import re

line = "Giraffes are taller than elephants";

matchObj = re.match( r'(.*) are(\.*)', line, re.M|re.I)

if matchObj:
   print "matchObj.group()  : ", matchObj.group()
   print "matchObj.group(1) : ", matchObj.group(1)
   print "matchObj.group(2) : ", matchObj.group(2)
else:
   print "matchObj does not match line:", line
```

The code in Listing A.4 produces the following output:

```
matchObj.group()  :  Giraffes are
matchObj.group(1) :  Giraffes
matchObj.group(2) :
```

Listing A.4 contains a pair of delimiters separated by a pipe ("|") symbol. The first delimiter is `re.M` for "multi-line" (this example contains only a single line of text), and the second delimiter `re.I` means "ignore case" during the pattern matching operation. The `re.match()` method supports additional delimiters, as discussed in the next section.

A.7 Options for the `re.match()` Method

The `match()` method supports various optional modifiers that affect the type of matching that will be performed. As you saw in the previous example, you can also specify multiple modifiers separated by the OR ("|") symbol. Additional modifiers that are available for RE are shown here:

- `re.I` performs case-insensitive matches (see previous section)
- `re.L` interprets words according to the current locale
- `re.M` makes $ match the end of a line and makes ^ match the start of any line
- `re.S` makes a period (".") match any character (including a newline)
- `re.U` interprets letters according to the Unicode character set
- Experiment with these modifiers by writing Python code that uses them in conjunction with different text strings.

A.8 Matching Character Classes with the `re.search()` Method

As you saw earlier in this appendix, the `re.match()` method only matches from the beginning of a string, whereas the `re.search()` method can successfully match a substring anywhere in a text string.

The `re.search()` method takes two arguments: a regular expression pattern and a string and then searches for the specified pattern in the given string. The `search()` method returns a match object (if the search was successful) or `None`.

As a simple example, the following searches for the pattern `tasty` followed by a 5 letter word:

```
import re

str = 'I want a tasty pizza'
match = re.search(r'tasty \w\w\w\w\w', str)

if match:
  ## 'found tasty pizza'
  print 'found', match.group()
else:
  print 'Nothing tasty here'
```

The output of the preceding code block is here:

```
found tasty pizza
```

The following code block further illustrates the difference between the `match()` method and the `search()` methods:

```
>>> import re
>>> print re.search('this', 'this is the one').span()
(0, 4)
>>>
>>> print re.search('the', 'this is the one').span()
(8, 11)
>>> print re.match('this', 'this is the one').span()
(0, 4)
>>> print re.match('the', 'this is the one').span()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'NoneType' object has no attribute
'span'
```

A.9 Matching Character Classes with the `findAll()` Method

Listing A.5 displays the contents of the Python script `RegEx1.py` that illustrates how to define simple character classes that match various text strings.

Listing A.5: RegEx1.py

```
import re

str1 = "123456"
matches1 = re.findall("(\d+)", str1)
print 'matches1:',matches1

str1 = "123456"
matches1 = re.findall("(\d\d\d)", str1)
print 'matches1:',matches1

str1 = "123456"
matches1 = re.findall("(\d\d)", str1)
print 'matches1:',matches1
```

```
print
str2 = "1a2b3c456"
matches2 = re.findall("(\d)", str2)
print 'matches2:',matches2

print
str2 = "1a2b3c456"
matches2 = re.findall("\d", str2)
print 'matches2:',matches2

print
str3 = "1a2b3c456"
matches3 = re.findall("(\w)", str3)
print 'matches3:',matches3
```

Listing A.5 contains simple regular expressions (which you have seen already) for matching digits in the variables `str1` and `str2`. The final code block of Listing A.5 matches every character in the string `str3`, effectively "splitting" `str3` into a list where each element consists of one character. The output from Listing A.5 is here (notice the blank lines after the first three output lines):

```
matches1: ['123456']
matches1: ['123', '456']
matches1: ['12', '34', '56']

matches2: ['1', '2', '3', '4', '5', '6']

matches2: ['1', '2', '3', '4', '5', '6']

matches3: ['1', 'a', '2', 'b', '3', 'c', '4', '5', '6']
```

A.9.1 Finding Capitalized Words in a String

Listing A.6 displays the contents of the Python script `FindCapitalized.py` that illustrates how to define simple character classes that match various text strings.

Listing A.6: FindCapitalized.py

```
import re

str = "This Sentence contains Capitalized words"
caps = re.findall(r'[A-Z][\w\.-]+', str)

print 'str: ',str
print 'caps:',caps
```

Listing A.6 initializes the string variable `str` and the `RE caps` that matches any word that starts with a capital letter because the first portion of `caps` is the pattern `[A-Z]` that matches any capital letter between `A` and `Z` inclusive.

The output of Listing A.6 is here:

```
str:  This Sentence contains Capitalized words
caps: ['This', 'Sentence', 'Capitalized']
```

A.10 Additional Matching Function for Regular Expressions

After invoking any of the methods `match()`, `search()`, `findAll()`, `or finditer()`, you can invoke additional methods on the "matching object." An example of this functionality using the `match()` method is here:

```
import re

p1 = re.compile('[a-z]+')
m1 = p1.match("hello")
```

In the preceding code block, the `p1` object represents the compiled regular expression for one or more lowercase letters, and the "matching object" `m1` object supports the following methods:

- `group()` return the string matched by the RE
- `start()` return the starting position of the match
- `end()` return the ending position of the match
- `span()` return a tuple containing the (start, end) positions of the match

As a further illustration, Listing A.7 displays the contents of `SearchFunction1.py` that illustrates how to use the `search()` method and the `group()` method.

Listing A.7: SearchFunction1.py

```
import re

line = "Giraffes are taller than elephants";

searchObj = re.search( r'(.*) are(\.*)', line,
re.M|re.I)

if searchObj:
   print "searchObj.group()  : ", searchObj.group()
   print "searchObj.group(1) : ", searchObj.group(1)
   print "searchObj.group(2) : ", searchObj.group(2)
else:
   print "searchObj does not match line:", line
```

Listing A.7 contains the variable line that represents a text string and the variable `searchObj` is an RE involving the `search()` method and pair of pipe-delimited modifiers (discussed in more detail in the next section). If `searchObj` is not null, the if/else conditional code in Listing A.7 displays the contents of the three groups resulting from the successful match with the contents of the variable line. The output from Listing A.7 is here:

```
searchObj.group()  :  Giraffes are
searchObj.group(1) :  Giraffes
searchObj.group(2) :
```

A.11 Grouping with Character Classes in Regular Expressions

In addition to the character classes that you have seen earlier in this appendix, you can specify subexpressions of character classes.

Listing A.8 displays the contents of `Grouping1.py` that illustrates how to use the `search()` method.

Listing A.8: Grouping1.py

```
import re

p1 = re.compile('(ab)*')
print 'match1:',p1.match('ababababab').group()
print 'span1: ',p1.match('ababababab').span()

p2 = re.compile('(a)b')
m2 = p2.match('ab')
print 'match2:',m2.group(0)
print 'match3:',m2.group(1)
```

Listing A.8 starts by defining the RE `p1` that matches zero or more occurrences of the string `ab`. The first `print` statement displays the result of using the `match()` function of `p1` (followed by the `group()` function) against a string, and the result is a string. This illustrates the use of "method chaining," which eliminates the need for an intermediate object (as shown in the second code block). The second `print` statement displays the result of using the `match()` function of `p1`, followed by applying the `span()` function, against a string. In this case the result is a numeric range (see following output).

The second part of Listing A.8 defines the RE `p2` that matches an optional letter a followed by the letter `b`. The variable `m2` invokes the match method on `p2` using the string `ab`. The third `print` statement displays the result of invoking `group(0)` on `m2`, and the fourth `print` statement displays the result of involving group(1) on m2. Both results are substrings of the input string `ab`. Recall that `group(0)` returns the highest level match that occurred, and `group(1)` returns a more "specific" match that occurred, such as one that involves the parentheses in the definition of `p2`. The higher the value of the integer in the expression `group(n)`, the more specific the match.

The output from Listing A.8 is here:

```
match1: abababababab
span1:  (0, 10)
match2: ab
match3: a
```

A.12 Using Character Classes in Regular Expressions

This section contains some examples that illustrate how to use character classes to match various strings and also how to use delimiters in order to split a text string. For example, one common date string involves a date format of the form MM/DD/YY. Another common scenario involves records with a delimiter that separates multiple fields. Usually such records contain one delimiter, but as you will see, Python makes it very easy to split records using multiple delimiters.

A.12.1 Matching Strings with Multiple Consecutive Digits

Listing A.9 displays the contents of the Python script `MatchPatterns1.py` that illustrates how to define simple regular expressions in order to split the contents of a text string based on the occurrence of one or more consecutive digits.

Although the regular expressions `\d+/\d+/\d+` and `\d\d/\d\d/\d\d\d\d` both match the string `08/13/2014`, the first regular expression matches more patterns than the second regular expression which is an "exact match" with respect to the number of matching digits that are allowed.

Listing A.9: MatchPatterns1.py

```
import re

date1 = '02/28/2013'
date2 = 'February 28, 2013'

# Simple matching: \d+ means match one or more digits
if re.match(r'\d+/\d+/\d+', date1):
  print('date1 matches this pattern')
else:
  print('date1 does not match this pattern')

if re.match(r'\d+/\d+/\d+', date2):
  print('date2 matches this pattern')
else:
  print('date2 does not match this pattern')
```

The output from launching Listing A.9 is here:

```
date1 matches this pattern
date2 does not match this pattern
```

A.12.2 Reversing Words in Strings

Listing A.10 displays the contents of the Python script `ReverseWords1.py` that illustrates how to reverse a pair of words in a string.

Listing A.10: ReverseWords1.py

```
import re

str1 = 'one two'
match = re.search('([\w.-]+) ([\w.-]+)', str1)
str2 = match.group(2) + ' ' + match.group(1)
print 'str1:',str1
print 'str2:',str2
```

The output from Listing A.10 is here:

```
str1: one two
str2: two one
```

Now that you understand how to define regular expressions for digits and letters, let's look at some more sophisticated regular expressions.

For example, the following expression matches a string that is any combination of digits, uppercase letters, or lowercase letters (i.e., no special characters):

```
^[a-zA-Z0-9]$
```

Here is the same expression rewritten using character classes:

```
^[\w\W\d]$
```

A.13 Modifying Text Strings with the `re` Module

The Python `re` module contains several methods for modifying strings. The `split()` method uses a regular expression to "split" a string into a list. The `sub()` method finds all substrings where the regular expression matches, and then replaces them with a different string. The `subn()` performs the same functionality as `sub()`, and also returns the new string and the number of replacements. The following subsections contain examples that illustrate how to use the functions `split()`, `sub()`, and `subn()` in regular expressions.

A.14 Splitting Text Strings with the `re.split()` Method

Listing A.11 displays the contents of the Python script `RegEx2.py` that illustrates how to define simple regular expressions in order to split the contents of a text string.

Listing A.11: RegEx2.py

```
import re

line1 = "abc def"
result1 = re.split(r'[\s]', line1)
print 'result1:',result1

line2 = "abc1,abc2:abc3;abc4"
result2 = re.split(r'[,:;]', line2)
print 'result2:',result2

line3 = "abc1,abc2:abc3;abc4 123 456"
result3 = re.split(r'[,:;\s]', line3)
print 'result3:',result3
```

Listing A.11 contains three blocks of code, each of which uses the `split()` method in the `re` module in order to tokenize three different strings. The first regular expression specifies a whitespace, the second regular expression specifies three punctuation characters, and the third regular expression specifies the combination of the first two regular expressions.

The output from launching `RegEx2.py` is here:

```
result1: ['abc', 'def']
result2: ['abc1', 'abc2', 'abc3', 'abc4']
result3: ['abc1', 'abc2', 'abc3', 'abc4', '123', '456']
```

A.15 Splitting Text Strings Using Digits and Delimiters

Listing A.12 displays the contents of `SplitCharClass1.py` that illustrates how to use regular expression consisting of a character class, the "." character, and a whitespace to split the contents of two text strings.

Listing A.12: SplitCharClass1.py

```
import re

line1 = '1. Section one 2. Section two 3. Section three'
line2 = '11. Section eleven 12. Section twelve 13.
Section thirteen'

print re.split(r'\d+\. ', line1)
print re.split(r'\d+\. ', line2)
```

Listing A.12 contains two text strings that can be split using the same regular expression `'\d+\. '`. Note that if you use the expression `'\d\. '` only the first text string will split correctly. The result of launching Listing A.12 is here:

```
['', 'Section one ', 'Section two ', 'Section three']
['', 'Section eleven ', 'Section twelve ', 'Section
thirteen']
```

A.16 Substituting Text Strings with the `re.sub()` Method

Earlier in this appendix you saw a preview of using the `sub()` method to remove all the metacharacters in a text string. The following code block illustrates how to use the `re.sub()` method to substitute alphabetic characters in a text string.

```
>>> import re
>>> p = re.compile( '(one|two|three)')
>>> p.sub( 'some', 'one book two books three books')
'some book some books some books'
>>>
>>> p.sub( 'some', 'one book two books three books',
count=1)
'some book two books three books'
```

The following code block uses the `re.sub()` method in order to insert a line feed after each alphabetic character in a text string:

```
>>> line = 'abcde'
>>> line2 = re.sub('', '\n', line)
>>> print 'line2:',line2
line2:
a
b
c
d
e
```

A.17 Matching the Beginning and the End of Text Strings

Listing A.13 displays the contents of the Python script `RegEx3.py` that illustrates how to find substrings using the `startswith()` function and `endswith()` function.

Listing A.13: RegEx3.py

```
import re

line2 = "abc1,Abc2:def3;Def4"
result2 = re.split(r'[,:;]', line2)

for w in result2:
  if(w.startswith('Abc')):
    print 'Word starts with Abc:',w
  elif(w.endswith('4')):
    print 'Word ends with 4:',w
  else:
    print 'Word:',w
```

Listing A.13 starts by initializing the string `line2` (with punctuation characters as word delimiters) and the RE `result2` that uses the `split()`

function with a comma, colon, and semicolon as “split delimiters” in order to tokenize the string variable `line2`.

The output after launching Listing A.13 is here:

```
Word: abc1
Word starts with Abc: Abc2
Word: def3
Word ends with 4: Def4
```

Listing A.14 displays the contents of the Python script `MatchLines1.py` that illustrates how to find substrings using character classes.

Listing A.14 MatchLines1.py

```
import re

line1 = "abcdef"
line2 = "123,abc1,abc2,abc3"
line3 = "abc1,abc2,123,456f"

if re.match("^[A-Za-z]*$", line1):
  print 'line1 contains only letters:',line1

# better than the preceding snippet:
line1[:-1].isalpha()
  print 'line1 contains only letters:',line1

if re.match("^[\w]*$", line1):
  print 'line1 contains only letters:',line1

if re.match(r"^[^\W\d_]+$", line1, re.LOCALE):
  print 'line1 contains only letters:',line1
print

if re.match("^[0-9][0-9][0-9]", line2):
  print 'line2 starts with 3 digits:',line2

if re.match("^\d\d\d", line2):
  print 'line2 starts with 3 digits:',line2
print

# does not work: fixme
if re.match("[0-9][0-9][0-9][a-z]$", line3):
  print 'line3 ends with 3 digits and 1 char:',line3

# does not work: fixme
if re.match("[a-z]$", line3):
  print 'line3 ends with 1 char:',line3
```

Listing A.14 starts by initializing 3 string variables `line1`, `line2`, and `line3`. The first RE contains an expression that matches any line containing uppercase or lowercase letters (or both):

```
if re.match("^[A-Za-z]*$", line1):
```

The following two snippets also test for the same thing:

```
line1[:-1].isalpha()
```

The preceding snippet starts from the right-most position of the string and checks if each character is alphabetic.

The next snippet checks if `line1` can be tokenized into words (a word contains only alphabetic characters):

```
if re.match("^[\w]*$", line1):
```

The next portion of Listing A.14 checks if a string contains three consecutive digits:

```
if re.match("^[0-9][0-9][0-9]", line2):
  print 'line2 starts with 3 digits:',line2

if re.match("^\d\d\d", line2):
```

The first snippet uses the pattern `[0-9]` to match a digit, whereas the second snippet uses the expression `\d` to match a digit.

The output from Listing A.14 is here:

```
line1 contains only letters: abcdef
line1 contains only letters: abcdef
line1 contains only letters: abcdef
line1 contains only letters: abcdef

line2 starts with 3 digits: 123,abc1,abc2,abc3
line2 starts with 3 digits: 123,abc1,abc2,abc3
```

A.18 Compilation Flags

Compilation flags modify the manner in which regular expressions work. Flags are available in the re module as a long name (such as IGNORECASE) and a short, one-letter form (such as I). The short form is the same as the flags in pattern modifiers in Perl. You can specify multiple flags by using the "|" symbol. For example, `re.I | re.M` sets both the `I` and `M` flags.

You can check the online Python documentation regarding all the available compilation flags in Python.

A.19 Compound Regular Expressions

Listing A.15 displays the contents of `MatchMixedCase1.py` that illustrates how to use the pipe ("|") symbol to specify two regular expressions in the same `match()` function.

Listing A.15: MatchMixedCase1.py

```
import re

line1 = "This is a line"
line2 = "That is a line"

if re.match("^[Tt]his", line1):
  print 'line1 starts with This or this:'
  print line1
else:
  print 'no match'

if re.match("^This|That", line2):
  print 'line2 starts with This or That:'
  print line2
else:
  print 'no match'
```

Listing A.15 starts with two string variables `line1` and `line2`, followed by an if/else conditional code block that checks if `line1` starts with the RE `[Tt]his`, which matches the string `This` as well as the string `this`.

The second conditional code block checks if `line2` starts with the string `This` or the string `That`. Notice the "^" metacharacter, which in this context anchors the RE to the beginning of the string. The output from Listing A.15 is here:

```
line1 starts with This or this:
This is a line
line2 starts with This or That:
That is a line
```

A.20 Counting Character Types in a String

You can use a regular expression to check whether a character is a digit, a letter, or some other type of character. Listing A.16 displays the contents of `CountDigitsAndChars.py` that performs this task.

Listing A.16: CountDigitsAndChars.py

```
import re

charCount  = 0
digitCount = 0
otherCount = 0

line1 = "A line with numbers: 12 345"

for ch in line1:
   if(re.match(r'\d', ch)):
     digitCount = digitCount + 1
   elif(re.match(r'\w', ch)):
     charCount = charCount + 1
   else:
     otherCount = otherCount + 1

print 'charcount:',charCount
print 'digitcount:',digitCount
print 'othercount:',otherCount
```

Listing A.16 initializes three numeric counter-related variables, followed by the string variable `line1`. The next part of Listing A.16 contains a `for` loop that processes each character in the string `line1`. The body of the `for` loop contains a conditional code block that checks whether the current character is a digit, a letter, or some other nonalphanumeric character. Each time there is a successful match, the corresponding "counter" variable is incremented.

The output from Listing A.16 is here:

```
charcount: 16
digitcount: 5
othercount: 6
```

A.21 Regular Expressions and Grouping

You can also "group" sub-expressions and even refer to them symbolically. For example, the following expression matches zero or 1 occurrences of 3 consecutive letters or digits:

```
^([a-zA-Z0-9]{3,3})?
```

The following expression matches a telephone number (such as 650-555-1212) in the USA:

```
^\d{3,3}[-]\d{3,3}[-]\d{4,4}
```

The following expression matches a zip code (such as `67827` or `94343-04005`) in the USA:

```
^\d{5,5}([-]\d{5,5})?
```

The following code block partially matches an email address:

```
str = 'john.doe@google.com'
  match = re.search(r'\w+@\w+', str)
  if match:
    print match.group()  ## 'doe@google'
```

Exercise: Use the preceding code block as a starting point in order to define a regular expression for email addresses.

A.22 Simple String Matches

Listing A.17 displays the contents of the Python script `RegEx4.py` that illustrates how to define regular expressions that match various text strings.

Listing A.17: RegEx4.py

```
import re

searchString = "Testing pattern matches"

expr1 = re.compile( r"Test" )
expr2 = re.compile( r"^Test" )
expr3 = re.compile( r"Test$" )
expr4 = re.compile( r"\b\w*es\b" )
expr5 = re.compile( r"t[aeiou]", re.I )

if expr1.search( searchString ):
   print '"Test" was found.'

if expr2.match( searchString ):
   print '"Test" was found at the beginning of the
line.'

if expr3.match( searchString ):
   print '"Test" was found at the end of the line.'

result = expr4.findall( searchString )
```

```
if result:
   print 'There are %d words(s) ending in "es":' % \
      ( len( result ) ),

   for item in result:
      print " " + item,

print

result = expr5.findall( searchString )
if result:
   print 'The letter t, followed by a vowel, occurs %d
times:' % \
      ( len( result ) ),

   for item in result:
      print " "+item,

print
```

Listing A.17 starts with the variable `searchString` that specifies a text string, followed by the REs `expr1`, `expr2`, `expr3`. The RE `expr1` matches the string `Test` that occurs anywhere in `searchString`, whereas `expr2` matches `Test` if it occurs at the beginning of `searchString`, and `expr3` matches `Test` if it occurs at the end of `searchString`. The RE `expr` matches words that end in the letters `es`, and the RE `expr5` matches the letter `t` followed by a vowel.

The output from Listing A.17 is here:

```
"Test" was found.
"Test" was found at the beginning of the line.
There are 1 words(s) ending in "es":  matches
The letter t, followed by a vowel, occurs 3 times:
Te ti te
```

A.23 Additional Topics for Regular Expressions

In addition to the Python-based search/replace functionality that you have seen in this appendix, you can also perform a greedy search and substitution. Perform an Internet search to learn what these features are and how to use them in Python code.

A.24 Summary

This appendix showed you how to create various types of regular expressions. First you learned how to define primitive regular expressions using sequences of digits, lowercase letters, and uppercase letters. Next you learned how to use character classes, which are more convenient and simpler expressions that can perform the same functionality. You also learned how to use the Python `re` library in order to compile regular expressions and then use them to see if they match substrings of text strings.

A.25 Exercises

Exercise 1: Given a text string, find the list of words (if any) that start or end with a vowel, and treat upper- and lowercase vowels as distinct letters. Display this list of words in alphabetical order, and also in descending order based their frequency.

Exercise 2: Given a text string, find the list of words (if any) that contain lowercase vowels or digits or both, but no uppercase letters. Display this list of words in alphabetical order, and also in descending order based their frequency.

Exercise 3: There is a spelling rule in English specifying that "the letter i is before e, except after c," which means that "receive" is correct but "recieve" is incorrect. Write a Python script that checks for incorrectly spelled words in a text string.

Exercise 4: Subject pronouns cannot follow a preposition in the English language. Thus, "between you and me" and "for you and me" are correct, whereas "between you and I" and "for you and I" are incorrect. Write a Python script that checks for incorrect grammar in a text string, and search for the prepositions "between," "for," and "with." In addition, search for the subject pronouns "I," "you," "he," and "she." Modify and display the text with the correct grammar usage.

Exercise 5: Find the words in a text string whose length is at most 4 and then print all the substrings of those characters. For example, if a text string contains the word "text," print the strings "t," "te," "tex," and "text."

APPENDIX B

INTRODUCTION TO KERAS

- What is `Keras`?
- Creating a `Keras`-based Model
- `Keras` and Linear Regression
- `Keras`, MLPs, and MNIST
- `Keras`, CNNs, and cifar10
- Resizing Images in `Keras`
- `Keras` and Early Stopping (1)
- `Keras` and Early Stopping (2)
- `Keras` and Metrics
- Saving and Restoring `Keras` Models
- Summary

This appendix introduces you to `Keras`, along with code samples that illustrate how to define basic neural networks as well as and deep neural networks with various datasets with as `MNIST` and `Cifar10`.

The first part of this appendix briefly discusses some of the important namespaces (such as `tf.keras.layers`) and their contents, as well as a simple `Keras`-based model.

The second section contains an example of performing linear regression with `Keras` and a simple CSV file. You will also see a `Keras`-based MLP neural network that is trained on the `MNIST` dataset.

The third section contains a simple example of training a neural network with the cifar10 dataset. This code sample is similar to training a neural network on the `MNIST` dataset, and requires a very small code change.

The final section contains two examples of `Keras`-based models that perform "early stopping," which is convenient when the model exhibits minimal improvement (that is specified by you) during the training process.

B.1 What is `Keras`?

If you are already comfortable with `Keras`, you can skim this section to learn about the new namespaces and what they contain, and then proceed to the next section that contains details for creating a `Keras`-based model.

If you are new to `Keras`, you might be wondering why this section is included in this appendix. First, `Keras` is well-integrated into TF 2, and it's in the `tf.keras` namespace. Second, `Keras` is well-suited for defining models to solve a myriad of tasks, such as linear regression and logistic regression, as well as deep learning tasks involving CNNs, RNNs, and LSTMs that are discussed in the appendix.

The next several subsections contain lists of bullet items for various `Keras`-related namespaces, and they will be very familiar if you have worked with TF 1.x. If you are new to TF 2, you'll see examples of some of the classes in subsequent code samples.

B.1.1 Working with `Keras` Namespaces in TF 2

TF 2 provides the `tf.keras` namespace, which in turn contains the following namespaces:

- tf.keras.layers
- tf.keras.models
- tf.keras.optimizers
- tf.keras.utils
- tf.keras.regularizers

The preceding namespaces contain various layers in `Keras` models, different types of `Keras` models, optimizers (`Adam` et al.), utility classes, and regularizers (such as L1 and L2), respectively.

Currently there are three ways to create Keras-based models:

- The sequential API
- The functional API
- The model API

The Keras-based code samples in this book use primarily the sequential API (it's the most intuitive and straightforward). The sequential API enables you to specify a list of layers, most of which are available in the tf.keras.layers namespace (discussed later).

The Keras-based models that use the functional API involve specifying layers that are passed as function-like elements in a pipeline-like fashion. Although the functional API provides some additional flexibility, you will probably use the sequential API to define Keras-based models if you are a TF 2 beginner.

The model-based API provides the greatest flexibility, and it involves defining a Python class that encapsulates the semantics of your Keras model. This class is a subclass of the tf.model.Model class, and you must implement the two methods __init__ and call in order to define a Keras model in this subclass.

Perform an online search for more details regarding the functional API and the model API.

B.1.2 Working with the tf.keras.layers Namespace

The most common (and also the simplest) Keras-based model is the Sequential() class that is in the tf.keras.models namespace. This model is comprised of various layers that belong to the tf.keras.layers namespace, as shown here:

- tf.keras.layers.Conv2D()
- tf.keras.layers.MaxPooling2D()
- tf.keras.layers.Flatten()
- tf.keras.layers.Dense()
- tf.keras.layers.Dropout()
- tf.keras.layers.BatchNormalization()

- tf.keras.layers.embedding()
- tf.keras.layers.RNN()
- tf.keras.layers.LSTM()
- tf.keras.layers.Bidirectional (ex: BERT)

The `Conv2D()` and `MaxPooling2D()` classes are used in `Keras`-based models for CNNs, which are discussed in Chapter 5. Generally speaking, the next six classes in the preceding list can appear in models for CNNs as well as models for machine learning. The `RNN()` class is for simple RNNS and the LSTM class is for LSTM-based models. The `Bidirectional()` class is a bidirectional LSTM that you will often see in models for solving natural language processing (NLP) tasks. Two very important NLP frameworks that use bidirectional LSTMs were released as open source (on GitHub) in 2018: ELMo from Facebook and BERT from Google.

B.1.3 Working with the `tf.keras.activations` Namespace

Machine learning and deep learning models require activation functions. For Keras-based models, the activation functions are in the `tf.keras.activations` namespace, some of which are listed here:

- `tf.keras.activations.relu`
- `tf.keras.activations.selu`
- `tf.keras.activations.linear`
- `tf.keras.activations.elu`
- `tf.keras.activations.sigmoid`
- `tf.keras.activations.softmax`
- `tf.keras.activations.softplus`
- `tf.keras.activations.tanh`
- `Others …`

The ReLU/SELU/ELU functions are closely related, and they often appear in artificial neural networks (ANNs) and CNNs. Before the `relu()` function became popular, the `sigmoid()` and `tanh()` functions were used in ANNs and CNNs. However, they are still important and they are used in various gates in GRUs and LSTMs. The `softmax()` function is typically used in the pair of layers consisting of the right-most hidden layer and the output layer.

B.1.4 Working with the `keras.tf.datasets` Namespace

For your convenience, TF 2 provides a set of built-in datasets in the `tf.keras.datasets` namespace, some of which are listed here:

- `tf.keras.datasets.boston_housing`
- `tf.keras.datasets.cifar10`
- `tf.keras.datasets.cifar100`
- `tf.keras.datasets.fashion_mnist`
- `tf.keras.datasets.imdb`
- `tf.keras.datasets.mnist`
- `tf.keras.datasets.reuters`

The preceding datasets are popular for training models with small datasets. The `mnist` dataset and `fashion_mnist` dataset are both popular when training CNNs, whereas the `boston_housing` dataset is popular for linear regression. The `Titanic` dataset is also popular for linear regression, but it's not currently supported as a default dataset in the `tf.keras.datasets` namespace.

B.1.5 Working with the `tf.keras.experimental` Namespace

The `contrib` namespace in TF 1.x has been deprecated in TF 2, and it's "successor" is the `tf.keras.experimental` namespace, which contains the following classes (among others):

- `tf.keras.experimental.CosineDecay`
- `tf.keras.experimental.CosineDecayRestarts`
- `tf.keras.experimental.LinearCosineDecay`
- `tf.keras.experimental.NoisyLinearCosineDecay`
- `tf.keras.experimental.PeepholeLSTMCell`

If you are a beginner, you probably won't use any of the classes in the preceding list. Although the `PeepholeLSTMCell` class is a variation of the LSTM class, there are limited use cases for this class.

B.1.6 Working with Other `tf.keras` Namespaces

TF 2 provides a number of other namespaces that contain useful classes, some of which are listed here:

- `tf.keras.callbacks     (early stopping)`
- `tf.keras.optimizers    (Adam et al)`
- `tf.keras.regularizers  (L1 and L2)`
- `tf.keras.utils        (to_categorical)`

The `tf.keras.callbacks` namespace contains a class that you can use for "early stopping," which is to say that it's possible to terminate the training process if there is insufficient reduction in the cost function in two successive iterations.

The `tf.keras.optimizers` namespace contains the various optimizers that are available for working in conjunction with cost functions, which includes the popular Adam optimizer.

The `tf.keras.regularizers` namespace contains two popular regularizers: the L1 regularizer (also called `LASSO` in machine learning) and the L2 regularizer (also called the `Ridge` regularizer in machine learning). L1 is for mean absolute error (MAE) and L2 is for mean squared error (MSE). Both of these regularizers act as "penalty" terms that are added to the chosen cost function in order to reduce the "influence" of features in a machine learning model. Note that `LASSO` can drive values to zero, with the result that features are actually eliminated from a model, and hence is related to something called *feature selection* in machine learning.

The `tf.keras.utils` namespace contains an assortment of functions, including the `to_categorical()` function for converting a class vector into a binary class.

Although there are other namespaces in TF 2, the classes listed in all the preceding subsections will probably suffice for the majority of your tasks if you are a beginner in TF 2 and machine learning.

B.1.7 TF 2 `Keras` versus "Standalone" `Keras`

The original `Keras` is actually a specification, with various "backend" frameworks such as TensorFlow, Theano, and CNTK. Currently `Keras` standalone does not support TF 2, whereas the implementation of `Keras` in `tf.keras` has been optimized for performance.

`Keras` standalone will live in perpetuity in the `keras.io` package, which is discussed in detail at the `Keras` website: `keras.io`.

Now that you have a high-level view of the TF 2 namespaces for `Keras` and the classes that they contain, let's find out how to create a `Keras`-based model, which is the subject of the next section.

B.2 Creating a `Keras`-based Model

The following list of steps describe the high-level sequence involved in creating, training, and testing a `Keras` model:

- Step 1: Determine a model architecture (the number of hidden layers, various activation functions, and so forth).
- Step 2: Invoke the compile() method.
- Step 3: Invoke the fit() method to train the model.
- Step 4: Invoke the evaluate() method to evaluate the trained model.
- Step 5: Invoke the predict() method to make predictions.

Step 1 involves determining the values of a number of hyperparameters, including:

- the number of hidden layers
- the number of neurons in each hidden layer
- the initial values of the weights of edges
- the cost function
- the optimizer
- the learning rate
- the dropout rate
- the activation function(s)

Steps 2 through 4 involve the training data, whereas step 5 involves the test data, which are included in the following more detailed sequence of steps for the preceding list:

- Specify a dataset (if necessary, convert data to numeric data)
- Split the dataset into training data and test data (usually 80/20 split)

- Define the Keras model (such as the `tf.keras.models.Sequential()` API)
- Compile the Keras model (the `compile()` API)
- Train (fit) the Keras model (the `fit()` API)
- Make a prediction (the `prediction()` API)

Note that the preceding bullet items skip some steps that are part of a real `Keras` model, such as evaluating the `Keras` model on the test data, as well as dealing with issues such as overfitting.

The first bullet item states that you need a dataset, which can be as simple as a CSV file with 100 rows of data and just 3 columns (or even smaller). In general, a dataset is substantially larger: it can be a file with 1,000,000 rows of data and 10,000 columns in each row. We'll look at a concrete dataset in a subsequent section.

Next, a `Keras` model is in the `tf.keras.models` namespace, and the simplest (and also very common) `Keras` model is `tf.keras.models.Sequential`. In general, a `Keras` model contains layers that are in the `tf.keras.layers` namespace, such as `tf.keras.Dense` (which means that two adjacent layers are completely connected).

The activation functions that are referenced in `Keras` layers are in the `tf.nn` namespace, such as the `tf.nn.ReLU` for the ReLU activation function.

Here's a code block of the `Keras` model that's described in the preceding paragraphs (which covers the first four bullet points):

```
import tensorflow as tf

model = tf.keras.models.Sequential([
  tf.keras.layers.Dense(512, activation=tf.nn.relu),

])
```

We have three more bullet items to discuss, starting with the compilation step. `Keras` provides a `compile()` API for this step, an example of which is here:

```
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

Next we need to specify a training step, and `Keras` provides the `fit()` API (as you can see, it's not called `train()`), an example of which is here:

```
model.fit(x_train, y_train, epochs=5)
```

The final step is the prediction that is performed via the `predict()` API, an example of which is here:

```
pred = model.predict(x)
```

Keep in mind that the `evaluate()` method is used for evaluating an trained model, and the output of this method is accuracy or loss. On the other hand, the `predict()` method makes predictions from the input data.

Listing B.1 displays the contents of `tf2_basic_keras.py` that combines the code blocks in the preceding steps into a single code sample.

Listing B.1: tf2_basic_keras.py

```
import tensorflow as tf

# NOTE: we need the train data and test data

model = tf.keras.models.Sequential([
  tf.keras.layers.Dense(1, activation=tf.nn.relu),
])

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test)
```

Listing B.1 contains no new code, and we've essentially glossed over some of the terms such as the optimizer (an algorithm that is used in conjunction with a cost function), the loss (the type of loss function) and the metrics (how to evaluate the efficacy of a model).

The explanations for these details cannot be condensed into a few paragraphs (alas), but the good news is that you can find a plethora of detailed online blog posts that discuss these terms.

B.3 `Keras` and Linear Regression

This section contains a simple example of creating a `Keras`-based model in order to solve a task involving linear regression: given a positive number representing kilograms of pasta, predict its corresponding price. Listing B.2 displays the contents of `pasta.csv` and Listing B.3 displays the contents of `keras_pasta.py` that performs this task.

Listing B.2: pasta.csv

```
weight,price
5,30
10,45
15,70
20,80
25,105
30,120
35,130
40,140
50,150
```

Listing B.3: keras_pasta.py

```
import tensorflow as tf
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# price of pasta per kilogram
df = pd.read_csv("pasta.csv")

weight = df['weight']
price  = df['price']

model = tf.keras.models.Sequential([
   tf.keras.layers.Dense(units=1,input_shape=[1])
])

# MSE loss function and Adam optimizer
model.compile(loss='mean_squared_error',
              optimizer=tf.keras.optimizers.Adam(0.1))

# train the model
history = model.fit(weight, price, epochs=100,
verbose=False)

# graph the # of epochs versus the loss
plt.xlabel('Number of Epochs')
plt.ylabel("Loss Values")
plt.plot(history.history['loss'])
plt.show()

print("Cost for 11kg:",model.predict([11.0]))
print("Cost for 45kg:",model.predict([45.0]))
```

Listing B.3 initializes the `Pandas Dataframe df` with the contents of the CSV file `pasta.csv`, and then initializes the variables `weight` and `cost` with the first and second columns, respectively, of `df`.

The next portion of Listing B.3 defines a `Keras`-based model that consists of a single `Dense` layer. This model is compiled and trained, and then a graph is displayed that shows the number of epochs on the horizontal axis and the corresponding value of the loss function for the vertical axis. Launch the code in Listing B.3 and you will see the following output:

```
Cost for 11kg: [[41.727108]]
Cost for 45kg: [[159.02121]]
```

Figure B.1 displays a graph of epochs versus loss during the training process.

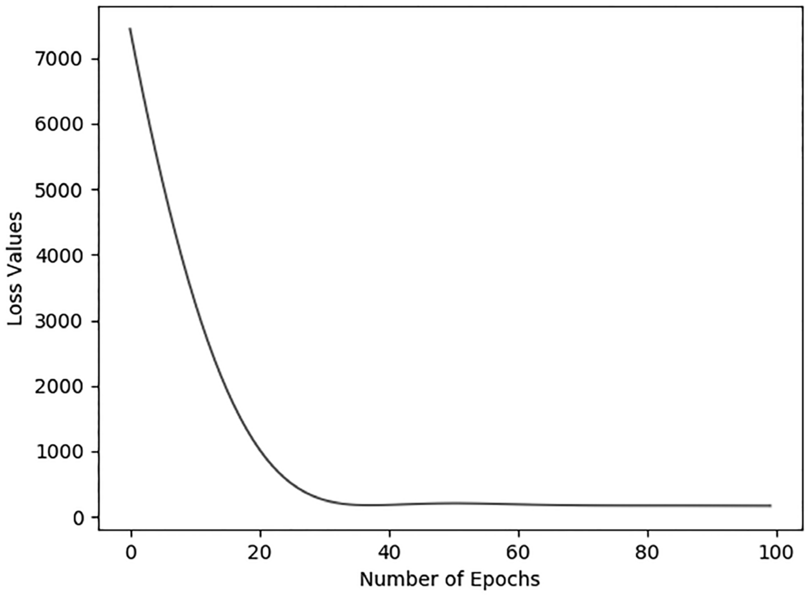

FIGURE B.1: A graph of epochs versus loss.

B.4 `Keras`, MLPs, and MNIST

This section contains a simple example of creating a `Keras`-based MLP neural network that will be trained with the MNIST dataset. Listing B.4 displays the contents of `keras_mlp_mnist.py` that performs this task.

Listing B.4: keras_mlp_mnist.py

```
import tensorflow as tf
import numpy as np

# instantiate mnist and load data:
```

```
mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# one-hot encoding for all labels to create 1x10
# vectors that are compared with the final layer:
y_train = tf.keras.utils.to_categorical(y_train)
y_test  = tf.keras.utils.to_categorical(y_test)

# resize and normalize the 28x28 images:
x_train = np.reshape(x_train, [-1, input_size])
x_train = x_train.astype('float32') / 255
x_test  = np.reshape(x_test, [-1, input_size])
x_test  = x_test.astype('float32') / 255

# initialize some hyper-parameters:
batch_size = 128
hidden_units = 218
dropout_rate = 0.3

# define a Keras based model:
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Dense(hidden_units, input_
dim=input_size))
model.add(tf.keras.layers.Activation('relu'))
model.add(tf.keras.layers.Dropout(dropout_rate))
model.add(tf.keras.layers.Dense(hidden_units))
model.add(tf.keras.layers.Activation('relu'))
model.add(tf.keras.layers.Dense(10))
model.add(tf.keras.layers.Activation('softmax'))

model.summary()

model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])

# train the network on the training data:
model.fit(x_train, y_train, epochs=10, batch_size=batch_
size)

# calculate and then display the accuracy:
loss, acc = model.evaluate(x_test, y_test, batch_
size=batch_size)
print("\nTest accuracy: %.1f%%" % (100.0 * acc))
```

Listing B.4 contains the usual `import` statements and then initializes the variable `mnist` as a reference to the MNIST dataset. The next portion of Listing B.4 contains some typical code that populates the training dataset and the test dataset and converts the labels to numeric values via the technique known as "one-hot" encoding.

Next, several hyperparameters are initialized, and a `Keras`-based model is defined that specifies three `Dense` layers and the `relu` activation function. This model is compiled and trained, and the accuracy on the test dataset is computed and then displayed. Launch the code in Listing B.4 and you will see the following output:

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                 (None, 256)              200960
_________________________________________________________________
activation (Activation)       (None, 256)              0
_________________________________________________________________
dropout (Dropout)             (None, 256)              0
_________________________________________________________________
dense_1 (Dense)               (None, 256)              65792
_________________________________________________________________
activation_1 (Activation)     (None, 256)              0
_________________________________________________________________
dense_2 (Dense)               (None, 10)               2570
_________________________________________________________________
activation_2 (Activation)     (None, 10)               0
=================================================================
Total params: 269,322
Trainable params: 269,322
Non-trainable params: 0

Train on 60000 samples
Epoch 1/10
60000/60000 [==============================] - 4s 74us/
sample - loss: 0.4281 - accuracy: 0.8683
Epoch 2/10
60000/60000 [==============================] - 4s 66us/
sample - loss: 0.1967 - accuracy: 0.9417
Epoch 3/10
60000/60000 [==============================] - 4s 63us/
sample - loss: 0.1507 - accuracy: 0.9547
Epoch 4/10
60000/60000 [==============================] - 4s 63us/
```

```
sample - loss: 0.1298 - accuracy: 0.9600
Epoch 5/10
60000/60000 [==============================] - 4s 60us/
sample - loss: 0.1141 - accuracy: 0.9651
Epoch 6/10
60000/60000 [==============================] - 4s 66us/
sample - loss: 0.1037 - accuracy: 0.9677
Epoch 7/10
60000/60000 [==============================] - 4s 61us/
sample - loss: 0.0940 - accuracy: 0.9702
Epoch 8/10
60000/60000 [==============================] - 4s 61us/
sample - loss: 0.0897 - accuracy: 0.9718
Epoch 9/10
60000/60000 [==============================] - 4s 62us/
sample - loss: 0.0830 - accuracy: 0.9747
Epoch 10/10
60000/60000 [==============================] - 4s 64us/
sample - loss: 0.0805 - accuracy: 0.9748
10000/10000 [==============================] - 0s 39us/
sample - loss: 0.0654 - accuracy: 0.9797

Test accuracy: 98.0%
```

B.5 `Keras`, CNNs, and cifar10

This section contains a simple example of training a neural network with the cifar10 dataset. This code sample is similar to training a neural network on the MNIST dataset, and requires a very small code change.

Keep in mind that images in MNIST have dimensions 28x28, whereas images in cifar10 have dimensions 32x32. Always ensure that images have the same dimensions in a dataset, otherwise the results can be unpredictable.

Note: Make sure that the images in your dataset have the same dimensions

Listing B.5 displays the contents of `keras_cnn_cifar10.py` that trains a CNN with the `cifar10` dataset.

Listing B.5: keras_cnn_cifar10.py

```
import tensorflow as tf

batch_size = 32
```

```
num_classes = 10
epochs = 100
num_predictions = 20

cifar10 = tf.keras.datasets.cifar10

# The data, split between train and test sets:
(x_train, y_train), (x_test, y_test) = cifar10.
load_data()
print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')

# Convert class vectors to binary class matrices
y_train = tf.keras.utils.to_categorical(y_train,
num_classes)
y_test = tf.keras.utils.to_categorical(y_test,
num_classes)

model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Conv2D(32, (3, 3),
padding='same', input_shape=x_train.shape[1:]))
model.add(tf.keras.layers.Activation('relu'))
model.add(tf.keras.layers.Conv2D(32, (3, 3)))
model.add(tf.keras.layers.Activation('relu'))
model.add(tf.keras.layers.MaxPooling2D
(pool_size=(2, 2)))
model.add(tf.keras.layers.Dropout(0.25))

# you can also duplicate the preceding code block here

model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(512))
model.add(tf.keras.layers.Activation('relu'))
model.add(tf.keras.layers.Dropout(0.5))
model.add(tf.keras.layers.Dense(num_classes))
model.add(tf.keras.layers.Activation('softmax'))

# use RMSprop optimizer to train the model
model.compile(loss='categorical_crossentropy',
              optimizer=opt,
              metrics=['accuracy'])

x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
```

```
x_test /= 255

model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=epochs,
          validation_data=(x_test, y_test),
          shuffle=True)

# evaluate and display results from test data
scores = model.evaluate(x_test, y_test, verbose=1)
print('Test loss:', scores[0])
print('Test accuracy:', scores[1])
```

Listing B.5 contains the usual `import` statement and then initializes the variable `cifar10` as a reference to the `cifar10` dataset. The next section of code is similar to the contents of Listing B.4: the main difference is that this `Keras`-based model defines a CNN instead of an MLP. Hence, the first layer is a convolutional layer, as shown here:

```
model.add(tf.keras.layers.Conv2D(32, (3, 3),
padding='same',
                  input_shape=x_train.shape[1:]))
```

Note that a "vanilla" CNN involves a convolutional layer (which is the purpose of the preceding code snippet), followed by the ReLU activation function, and a max pooling layer, both of which are displayed in Listing B.5. In addition, the final layer of the `Keras` model is the `softmax` activation function, which converts the 10 numeric values in the "fully connected" layer to a set of 10 non-negative numbers between 0 and 1, whose sum equals 1 (this gives us a probability distribution).

This model is compiled and trained, and then evaluated on the test dataset. The last portion of Listing B.5 displays the value of the test-related loss and accuracy, both of which are calculated during the preceding evaluation step. Launch the code in Listing B.5 and you will see the following output (note that the code was stopped after partially completing the second epoch):

```
x_train shape: (50000, 32, 32, 3)
50000 train samples
10000 test samples

Epoch 1/100
50000/50000 [==============================] - 285s 6ms/
sample - loss: 1.7187 - accuracy: 0.3802 - val_loss:
1.4294 - val_accuracy: 0.4926
```

```
Epoch 2/100
 1888/50000 [>.............................] - ETA: 4:39
- loss: 1.4722
- accuracy: 0.4635
```

B.6 Resizing Images in `Keras`

Listing B.6 displays the contents of `keras_resize_image.py` that illustrates how to resize an image in `Keras`.

Listing B.6: keras_resize_image.py

```
import tensorflow as tf
import numpy as np
import imageio
import matplotlib.pyplot as plt

# use any image that has 3 channels
inp = tf.keras.layers.Input(shape=(None, None, 3))
out = tf.keras.layers.Lambda(lambda image: tf.image.
resize(image, (128, 128)))(inp)

model = tf.keras.Model(inputs=inp, outputs=out)
model.summary()

# read the contents of a PNG or JPG
X = imageio.imread('sample3.png')

out = model.predict(X[np.newaxis, ...])

fig, axes = plt.subplots(nrows=1, ncols=2)
axes[0].imshow(X)
axes[1].imshow(np.int8(out[0,...]))

plt.show()
```

Listing B.6 contains the usual `import` statements and then initializes the variable `inp` so that it can accommodate a color image, followed by the variable `out` that is the result of resizing `inp` so that it has three color channels. Next, `inp` and `out` are specified as the values of `inputs` and `outputs`, respectively, for the `Keras` model, as shown in this code snippet:

```
model = tf.keras.Model(inputs=inp, outputs=out)
```

Next, the variable X is initialized as a reference to the result of reading the contents of the image `sample3.png`. The remainder of Listing B.6 involves displaying two images: the original image and the resized image.

Launch the code in Listing B.6 and you will see a graph of an image and its resized image as shown in Figure B.2.

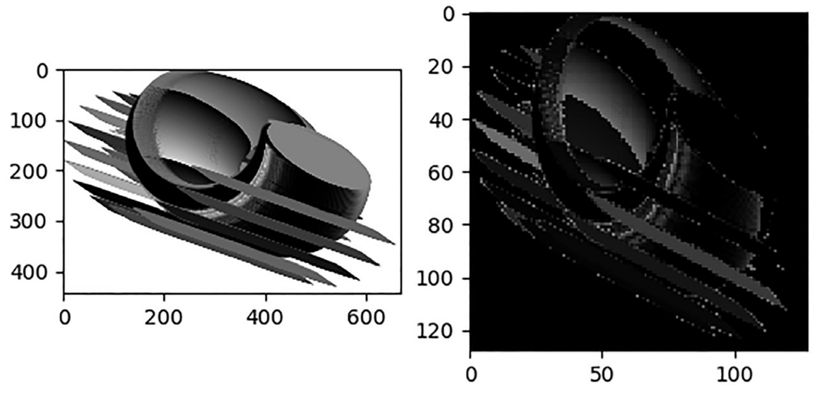

FIGURE B.2: A Graph of an image and its resized image.

B.7 `Keras` and Early Stopping (1)

After specifying the training set and the test set from a dataset, you also decide on the number of training epochs. A value that's too large can lead to overfitting, whereas a value that's too small can lead to underfitting. Moreover, model improvement can diminish and subsequent training iterations become redundant.

Early stopping is a technique that allows you to specify a large value for the number of epochs, and yet the training will stop if the model performance improvement drops below a threshold value.

There are several ways that you can specify early stopping, and they involve the concept of a *callback function*. Listing B.7 displays the contents of `tf2_keras_callback.py` that performs early stopping via a callback mechanism.

Listing B.7: tf2_keras_callback.py

```
import tensorflow as tf
import numpy as np

model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(64, activation='relu'))
model.add(tf.keras.layers.Dense(64, activation='relu'))
model.add(tf.keras.layers.Dense(10, activation='softmax'))

model.compile(optimizer=tf.keras.optimizers.Adam(0.01),
              loss='mse',       # mean squared error
              metrics=['mae'])  # mean absolute error

data   = np.random.random((1000, 32))
labels = np.random.random((1000, 10))

val_data   = np.random.random((100, 32))
val_labels = np.random.random((100, 10))

callbacks = [
  # stop training if "val_loss" stops improving for over 
2 epochs
  tf.keras.callbacks.EarlyStopping(patience=2, 
monitor='val_loss'),
  # write TensorBoard logs to the ./logs directory
  tf.keras.callbacks.TensorBoard(log_dir='./logs')
]

model.fit(data, labels, batch_size=32, epochs=50, 
callbacks=callbacks,
          validation_data=(val_data, val_labels))

model.evaluate(data, labels, batch_size=32)
```

Listing B.7 defines a `Keras`-based model with three hidden layers and then compiles the model. The next portion of Listing B.7 uses the `np.random.random` function in order to initialize the variables `data`, `labels`, `val_data`, and `val_labels`.

The interesting code involves the definition of the `callbacks` variable that specifies `tf.keras.callbacks.EarlyStopping` class with a value of 2 for `patience`, which means that the model will stop training if there is an insufficient reduction in the value of `val_loss`. The `callbacks` variable includes the `tf.keras.callbacks.TensorBoard` class to specify the logs subdirectory as the location for the TensorBoard files.

Next, the `model.fit()` method is invoked with a value of 50 for `epochs` (shown in bold), followed by the `model.evaluate()` method. Launch the code in Listing B.7 and you will see the following output:

```
Epoch 1/50
1000/1000 [==============================] - 0s 354us/
```

```
sample - loss: 0.2452 - mae: 0.4127 - val_loss: 0.2517 -
val_mae: 0.4205
Epoch 2/50
1000/1000 [==============================] - 0s 63us/
sample - loss: 0.2447 - mae: 0.4125 - val_loss: 0.2515 -
val_mae: 0.4204
Epoch 3/50
1000/1000 [==============================] - 0s 63us/
sample - loss: 0.2445 - mae: 0.4124 - val_loss: 0.2520 -
val_mae: 0.4209
Epoch 4/50
1000/1000 [==============================] - 0s 68us/
sample - loss: 0.2444 - mae: 0.4123 - val_loss: 0.2519 -
val_mae: 0.4205
1000/1000 [==============================] - 0s 37us/
sample - loss: 0.2437 - mae: 0.4119
(1000, 10)
```

Notice that the code stopped training after four epochs, even though 50 epochs are specified in the code.

B.8 `Keras` and Early Stopping (2)

The previous section contains a code sample with minimalistic functionality with respect to the use of callback functions in `Keras`. However, you can also define a custom class that provides finer-grained functionality that uses a callback mechanism.

Listing B.8 displays the contents of `tf2_keras_callback2.py` that performs early stopping via a callback mechanism (the new code is shown in bold).

Listing B.8: tf2_keras_callback2.py

```
import tensorflow as tf
import numpy as np

model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(64, activation='relu'))
model.add(tf.keras.layers.Dense(64, activation='relu'))
model.add(tf.keras.layers.Dense(10,
activation='softmax'))
```

```
model.compile(optimizer=tf.keras.optimizers.Adam(0.01),
              loss='mse',       # mean squared error
              metrics=['mae'])  # mean absolute error

data   = np.random.random((1000, 32))
labels = np.random.random((1000, 10))

val_data   = np.random.random((100, 32))
val_labels = np.random.random((100, 10))

class MyCallback(tf.keras.callbacks.Callback):
  def on_train_begin(self, logs={}):
    print("on_train_begin")

  def on_train_end(self, logs={}):
    print("on_train_begin")
    return

  def on_epoch_begin(self, epoch, logs={}):
    print("on_train_begin")
    return

  def on_epoch_end(self, epoch, logs={}):
    print("on_epoch_end")
    return

  def on_batch_begin(self, batch, logs={}):
    print("on_batch_begin")
    return

  def on_batch_end(self, batch, logs={}):
    print("on_batch_end")
    return

callbacks = [MyCallback()]

model.fit(data, labels, batch_size=32, epochs=50,
callbacks=callbacks,
          validation_data=(val_data, val_labels))

model.evaluate(data, labels, batch_size=32)
```

The new code in Listing B.8 differs from Listing B.7 is limited to the code block that is displayed in bold. This new code defines a custom Python class with several methods, each of which is invoked during the appropriate point during the `Keras` lifecycle execution. The six methods consists of three pairs of methods for the start event and end event associated with training, epochs, and batches, as listed here:

- `def on_train_begin()`
- `def on_train_end()`
- `def on_epoch_begin()`
- `def on_epoch_end()`
- `def on_batch_begin()`
- `def on_batch_end()`

The preceding methods contain just a `print()` statement in Listing B.8, and you can insert any code you wish in any of these methods. Launch the code in Listing B.8 and you will see the following output:

```
on_train_begin
on_train_begin
Epoch 1/50
on_batch_begin
on_batch_end
  32/1000 [..............................] - ETA: 4s -
loss: 0.2489 - mae: 0.4170on_batch_begin
on_batch_end
on_batch_begin on_batch_end
// details omitted for brevity
on_batch_begin
on_batch_end
on_batch_begin
on_batch_end
992/1000 [============================>.] - ETA: 0s -
loss: 0.2468 - mae: 0.4138on_batch_begin
on_batch_end
on_epoch_end
1000/1000 [==============================] - 0s 335us/
sample - loss: 0.2466 - mae: 0.4136 - val_loss: 0.2445 -
val_mae: 0.4126
on_train_begin
Epoch 2/50
on_batch_begin
on_batch_end
```

```
 32/1000 [..............................] - ETA: 0s -
loss: 0.2465 - mae: 0.4133on_batch_begin
on_batch_end
on_batch_begin
on_batch_end
// details omitted for brevity
on_batch_end
on_epoch_end
1000/1000 [==============================] - 0s 51us/
sample - loss: 0.2328 - mae: 0.4084 - val_loss: 0.2579 -
val_mae: 0.4241
on_train_begin
 32/1000 [..............................] - ETA: 0s -
loss: 0.2295 - mae: 0.4030
1000/1000 [==============================] - 0s 22us/
sample - loss: 0.2313 - mae: 0.4077
(1000, 10)
```

B.9 `Keras` and Metrics

Many `Keras`-based models only specify "accuracy" as the metric for evaluating a trained model, as shown here:

```
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

However, there are many other built-in metrics available, each of which is encapsulated in a `Keras` class in the `tf.keras.metrics` namespace. A list of many such metrics are displayed in the following list:

- class Accuracy: how often predictions matches labels
- class BinaryAccuracy: how often predictions matches labels
- class CategoricalAccuracy: how often predictions matches labels
- class FalseNegatives: the number of false negatives
- class FalsePositives: the number of false positives
- class Mean: the (weighted) mean of the given values
- class Precision: the precision of the predictions wrt the labels
- class Recall: the recall of the predictions wrt the labels

- class TrueNegatives: the number of true negatives
- class TruePositives: the number of true positives

Earlier in this chapter you learned about the "confusion matrix" that provides numeric values for TP, TN, FP, and FN; each of these values has a corresponding `Keras` class `TruePositive`, `TrueNegative`, `False-Positive`, and `FalseNegative`, respectively. Perform an online search for code samples that use the metrics in the preceding list.

B.10 Saving and Restoring `Keras` Models

Listing B.9 displays the contents of `tf2_keras_save_model.py` that creates, trains, and saves a `Keras`-based model, then creates a new model that is populated with the data from the saved model.

Listing B.9: tf2_keras_save_model.py

```
import tensorflow as tf
import os
def create_model():
  model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(512, activation=tf.nn.relu),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(10, activation=tf.nn.softmax)
  ])

  model.compile(optimizer=tf.keras.optimizers.Adam(),
               loss=tf.keras.losses.sparse_categorical_
crossentropy, metrics=['accuracy'])
  return model

# Create a basic model instance
model = create_model()
model.summary()

checkpoint_path = "checkpoint/cp.ckpt"
checkpoint_dir = os.path.dirname(checkpoint_path)

# Create checkpoint callback
cp_callback = tf.keras.callbacks.
ModelCheckpoint(checkpoint_path,
save_weights_only=True, verbose=1)
```

```
# => model #1: create the first model
model = create_model()

mnist = tf.keras.datasets.mnist
(X_train, y_train),(X_test, y_test) = mnist.load_data()

X_train, X_test = X_train / 255.0, X_test / 255.0
print("X_train.shape:",X_train.shape)

model.fit(X_train, y_train,  epochs = 2,
          validation_data = (X_test,y_test),
          callbacks = [cp_callback])  # pass callback to
training

# => model #2: create a new model and load saved model
model = create_model()
loss, acc = model.evaluate(X_test, y_test)
print("Untrained model, accuracy: {:5.2f}%".
format(100*acc))

model.load_weights(checkpoint_path)
loss,acc = model.evaluate(X_test, y_test)
print("Restored model, accuracy: {:5.2f}%".
format(100*acc))
```

Listing B.9 starts with the `create_model()` Python function that creates and compiles a `Keras`-based model. The next portion of Listing B.9 defines the location of the file that will be saved as well as the checkpoint callback, as shown here:

```
checkpoint_path = "checkpoint/cp.ckpt"
checkpoint_dir = os.path.dirname(checkpoint_path)

# Create checkpoint callback
cp_callback = tf.keras.callbacks.
ModelCheckpoint(checkpoint_path,
save_weights_only=True, verbose=1)
```

The next portion of Listing B.9 trains the current model using the `MNIST` dataset, and also specifies `cp_callback` so that the model can be saved.

The final code block in Listing B.8 creates a new `Keras`-based model by invoking the Python method `create_model()` again, evaluating this new model on the test-related data, and displaying the value of the accuracy. Next, the model is loaded with the saved model weights via the `load_weights()` API. The relevant code block is reproduced here:

```
model = create_model()
loss, acc = model.evaluate(X_test, y_test)
print("Untrained model, accuracy: {:5.2f}%".
format(100*acc))

model.load_weights(checkpoint_path)
loss,acc = model.evaluate(X_test, y_test)
print("Restored model, accuracy: {:5.2f}%".
format(100*acc))
```

Now launch the code in Listing B.9 and you will see the following output:

```
on_train_begin
Model: "sequential"
_________________________________________________________
Layer (type)             Output Shape        Param #
=========================================================
flatten (Flatten)        (None, 784)         0
_________________________________________________________
dense (Dense)            (None, 512)         401920
_________________________________________________________
dropout (Dropout)        (None, 512)         0
_________________________________________________________
dense_1 (Dense)          (None, 10)          5130
=========================================================
Total params: 407,050
Trainable params: 407,050
Non-trainable params: 0

Train on 60000 samples, validate on 10000 samples
Epoch 1/2
59840/60000 [============================>.] - ETA: 0s -
loss: 0.2173 - accuracy: 0.9351
Epoch 00001: saving model to checkpoint/cp.ckpt
60000/60000 [==============================] - 10s
168us/sample - loss: 0.2170 - accuracy: 0.9352 - val_
loss: 0.0980 - val_accuracy: 0.9696
Epoch 2/2
59936/60000 [============================>.] - ETA: 0s -
loss: 0.0960 - accuracy: 0.9707
Epoch 00002: saving model to checkpoint/cp.ckpt
60000/60000 [==============================] - 10s
174us/sample - loss: 0.0959 - accuracy: 0.9707 - val_
loss: 0.0735 - val_accuracy: 0.9761
```

```
10000/10000 [==============================] - 1s 86us/
sample - loss: 2.3986 - accuracy: 0.0777
Untrained model, accuracy:  7.77%
10000/10000 [==============================] - 1s 67us/
sample - loss: 0.0735 - accuracy: 0.9761
Restored model, accuracy: 97.61%
```

The directory where you launched this code sample contains a new subdirectory called `checkpoint` whose contents are shown here:

```
-rw-r--r--  1 owner  staff     1222 Aug 17 14:34 cp.ckpt.
index
-rw-r--r--  1 owner  staff  4886716 Aug 17 14:34 cp.ckpt.
data-00000-of-00001
-rw-r--r--  1 owner  staff       71 Aug 17 14:34
checkpoint
```

B.11 Summary

This appendix introduced you to some of the features of `Keras` and an assortment of `Keras`-based code samples involving basic neural networks with the `MNIST` and `Cifar10` datasets. You learned about some of the important namespaces (such as `tf.keras.layers`) and their contents.

Next, you saw an example of performing linear regression with `Keras` and a simple CSV file. Then you learned how to create a `Keras`-based MLP neural network that is trained on the `MNIST` dataset.

In addition, you saw examples of `Keras`-based models that perform “early stopping,” which is convenient when the model exhibits minimal improvement (that is specified by you) during the training process.

APPENDIX C

INTRODUCTION TO TF 2

- What is TF 2?
- Other TF 2-based Toolkits
- TF 2 Eager Execution
- TF2 Tensors, Data Types, and Primitive Types
- Constants in TF 2
- Variables in TF 2
- The `tf.rank()` API
- The `tf.shape()` API
- Variables in TF 2 (Revisited)
- What is `@tf.function` in TF 2?
- Working with `@tf.print()` in TF 2
- Arithmetic Operations in TF 2
- Caveats for Arithmetic Operations in TF 2
- TF 2 and Built-in Functions
- Calculating Trigonometric Values in TF 2
- Calculating Exponential Values in TF 2
- Working with Strings in TF 2
- Working with Tensors and Operations in TF 2
- 2nd Order Tensors in TF 2 (1)

- 2nd Order Tensors in TF 2 (2)
- Multiplying Two 2nd Order Tensors in TF
- Convert Python Arrays to TF Arrays
- Differentiation and `tf.GradientTape` in TF 2
- Examples of `tf.GradientTape`
- Google Colaboratory
- Other Cloud Platforms
- Summary

Welcome to TensorFlow 2! This chapter introduces you to various features of TensorFlow 2 (abbreviated as TF 2), as well as some of the TF 2 tools and projects that are covered under the TF 2 "umbrella." You will see TF 2 code samples that illustrate new TF 2 features (such as `tf.GradientTape` and the `@tf.function` decorator), plus an assortment of code samples that illustrate how to write code "the TF 2 way."

Despite the simplicity of many topics in this chapter, they provide you with a foundation for TF 2. This chapter also prepares you for Chapter 2 that delves into frequently used TF 2 APIs that you will encounter in other chapters of this book.

Keep in mind that the TensorFlow 1.x releases are considered legacy code after the production release of TF 2. Google will provide only security related updates for TF 1.x (i.e., no new code development), and support TensorFlow 1.x for at least another year beyond the initial production release of TF 2. For your convenience, TensorFlow provides a conversion script to facilitate the automatic conversion of TensorFlow 1.x code to TF 2 code in many cases (details provided later in this chapter).

As you saw in the Preface, this chapter contains several sections regarding TF 1.x, all of which are placed near the end of this chapter. If you do not have TF 1.x code, obviously these sections are optional (and they are labeled as such).

The first part of this chapter briefly discusses some TF 2 features and some of the tools that are included under the TF 2 "umbrella." The second section of this chapter shows you how to write TF 2 code involving TF constants and TF variables.

The third section digresses a bit: you will learn about the new TF 2 Python function decorator `@tf.function` that is used in many code samples in this chapter. Although this decorator is not always required, it's important to become comfortable with this feature, and there are some nonintuitive caveats regarding its use that are discussed in this section.

The fourth section of this chapter shows you how to perform typical arithmetic operations in TF 2, how to use some of the built-in TF 2 functions, and how to calculate trigonometric values. If you need to perform scientific calculations, see the code samples that pertain to the type of precision that you can achieve with floating point numbers in TF 2. This section also shows you how to use `for` loops and how to calculate exponential values.

The fifth section contains TF 2 code samples involving arrays, such as creating an identity matrix, a constant matrix, a random uniform matrix, and a truncated normal matrix, along with an explanation about the difference between a truncated matrix and a random matrix. This section also shows you how to multiply 2nd order tensors in TF 2 and how to convert Python arrays to 2nd order tensors in TF 2. The sixth section contains code samples that illustrate how to use some of the new features of TF 2, such as `tf.GradientTape`.

Although the TF 2 code samples in this book use Python 3.x, it's possible to modify the code samples in order to run under Python 2.7. Also make note of the following convention in this book (and only this book): TF 1.x files have a "tf_" prefix and TF 2 files have a "tf2_" prefix.

With all that in mind, the next section discusses a few details of TF 2, its architecture, and some of its features.

C.1 What is TF 2?

TF 2 is an open source framework from Google that is the newest version of TensorFlow. The TF 2 framework is a modern framework that's well-suited for machine learning and deep learning, and it's available through an Apache license. Interestingly, TensorFlow surprised many people, perhaps even members of the TF team, in terms of the creativity and plethora of use cases for TF in areas such as art, music, and medicine. For a variety of reasons, the TensorFlow team created TF 2 with the goal of consolidating the TF APIs, eliminating duplication of APIs, enabling rapid prototyping, and making debugging an easier experience.

There is good news if you are a fan of `Keras`: improvements in TF 2 are partially due to the adoption of `Keras` as part of the core functionality of

TF 2. In fact, TF 2 extends and optimizes `Keras` so that it can take advantage of all the advanced features in TF 2.

If you work primarily with deep learning models (CNNs, RNNs, LSTMs, and so forth), you'll probably use some of the classes in the `tf.keras` namespace, which is the implementation of `Keras` in TF 2. Moreover, `tf.keras.layers` provides several standard layers for neural networks. As you'll see later, there are several ways to define `Keras`-based models, via the `tf.keras.Sequential` class, a functional style definition, and via a subclassing technique. Alternatively, you can still use lower-level operations and automatic differentiation if you wish to do so.

Furthermore, TF 2 removes duplicate functionality, provides a more intuitive syntax across APIs, and also compatibility throughout the TF 2 ecosystem. TF 2 even provides a backward compatibility module called `tf.compat.v1` (which does not include `tf.contrib`), and a conversion script `tf_upgrade_v2` to help users migrate from TF 1.x to TF 2.

Another significant change in TF 2 is eager execution as the default mode (not deferred execution), with new features such as the `@tf.function` decorator and TF 2 privacy-related features. Here is a condensed list of some TF 2 features and related technologies:

- Support for `tf.keras`: a specification for high-level code for ML and DL
- TensorFlow.js v1.0: TF in modern browsers
- TensorFlow Federated: an open source framework for ML and decentralized data
- Ragged tensors: nested variable-length (“uneven”) lists
- TensorFlow probability: probabilistic models combined with deep learning
- Tensor2Tensor: a library of DL models and datasets

TF 2 also supports a variety of programming languages and hardware platforms, including:

- 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 TF 2 home page, where you will find links to many resources for TF 2: *https://www.tensorflow.org*

C.1.1 TF 2 Use Cases

TF 2 is designed to solve tasks that arise in a plethora of 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

The preceding list of use cases can be solved in TF 1.x as well as TF 2, and in the latter case, the code tends to be simpler and cleaner compared to their TF 1.x counterpart.

C.1.2 TF 2 Architecture: The Short Version

TF 2 is written in C++ and supports operations involving primitive values and tensors (discussed later). 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.

TF 2 supports arithmetic operations on tensors (i.e., multi-dimensional arrays with enhancements) as well as conditional logic, "for" loops, and "while" loops. Although it's possible to switch between eager execution mode and deferred mode in TF 2, all the code samples in this book use eager execution mode.

Data visualization is handled via TensorBoard (discussed in Chapter 2) that is included as part of TF 2. As you will see in the code samples in this book, TF 2 APIs are available in Python and can therefore be embedded in Python scripts.

So, enough already with the high-level introduction: let's learn how to install TF 2, which is the topic of the next section.

C.1.3 TF 2 Installation

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

```
pip install tensorflow==2.0.0
```

When a production release of TF 2 is available, you can issue the following command from the command line (which will be the most current version of TF 2):

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

If you want to install a specific version (let's say version 1.13.1) of TensorFlow, type the following command:

```
pip install --upgrade tensorflow==1.13.1
```

You can also downgrade the installed version of TensorFlow. For example, if you have installed version 1.13.1 and you want to install version 1.10, specify the value 1.10 in the preceding code snippet. TensorFlow will uninstall your current version and install the version that you specified (i.e., 1.10).

As a sanity check, create a Python script with the following three line of code to determine the version number of TF that is installed on your machine:

```
import tensorflow as tf
print("TF Version:",tf.__version__)
print("eager execution:",tf.executing_eagerly())
```

Launch the preceding code and you ought to see something similar to the following output:

```
TF version: 2.0.0
eager execution: True
```

As a simple example of TF 2 code, place this code snippet in a text file:

```
import tensorflow as tf
print("1 + 2 + 3 + 4 =", tf.reduce_sum([1, 2, 3, 4]))
```

Launch the preceding code from the command line and you should see the following output:

```
1 + 2 + 3 + 4 = tf.Tensor(10, shape=(), dtype=int32)
```

C.1.4 TF 2 and the Python REPL

In case you aren't already familiar with the Python `REPL` (read-eval-print-loop), it's accessible by opening a command shell and then typing the following command:

```
python
```

As a simple illustration, access TF 2-related functionality in the `REPL` by importing the TF 2 library as follows:

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

Now check the version of TF 2 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 TF 2 that you installed):

```
TF version: 2.0.0
```

Although the `REPL` is useful for short code blocks, the TF 2 code samples in this book are Python scripts that you can launch with the Python executable.

C.2 Other TF 2-based Toolkits

In addition to providing support for TF 2-based code on multiple devices, TF 2 provides the following toolkits:

- TensorBoard for visualization (included as part of TensorFlow)
- TensorFlow Serving (hosting on a server)
- TensorFlow Hub
- TensorFlow Lite (for mobile applications)
- Tensorflow.js (for Web pages and NodeJS)

TensorBoard is a graph visualization tool that runs in a browser. Launch TensorBoard from the command line as follows: open a command shell and type the following command to access a saved TF graph in the subdirectory `/tmp/abc` (or a directory of your choice):

```
tensorboard –logdir /tmp/abc
```

Note that there are two consecutive dashes ("-") that precede the `log-dir` parameter in the preceding command. Now launch a browser session and navigate to this URL: `localhost:6006`

After a few moments you will see a visualization of the TF 2 graph that was created in your code and then saved in the directory `/tmp/abc`.

TensorFlow Serving is a cloud-based flexible, high-performance serving system for ML models that is 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.TF 2.org/serving/*

TensorFlow Lite was specifically created for mobile development (both Android and iOS). Please keep in mind that TensorFlow Lite supersedes TF 2 Mobile, which was an earlier SDK for developing mobile applications. TensorFlow Lite (which also exists for TF 1.x) supports on-device ML inference with low latency and a small binary size. Moreover, TensorFlow Lite supports hardware acceleration with the Android Neural Networks API. More information about TensorFlow Lite can be found at:

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`. You can also use `tensorflow.js` with NodeJS. More information about `tensorflow.js` can be found at: *https://js.tensorflow.org*.

C.3 TF 2 Eager Execution

TF 2 eager execution mode makes TF 2 code much easier to write compared to TF 1.x code (which used deferred execution mode). You might be surprised to discover that TF introduced "eager execution" as an alternative to deferred execution in version 1.4.1, but this feature was vastly underutilized. With TF 1.x code, TensorFlow creates a dataflow graph that consists of (1) a set of `tf.Operation` objects that represent units of computation, and (2) `tf.Tensor` objects that represent the units of data that flow between operations.

Variously, TF 2 evaluates operations immediately without instantiating a session object or a creating a graph. Operations return concrete values instead of creating a computational graph. TF 2 eager execution is based on Python control flow instead of graph control flow. Arithmetic operations are simpler and intuitive, as you will see in code samples later in this chapter. Moreover, TF 2 eager execution mode simplifies the debugging process. However, keep in mind that there isn't a 1:1 relationship between a graph and eager execution.

C.4 TF 2 Tensors, Data Types, and Primitive Types

In simplified terms, a TF 2 tensor is an n-dimensional array that is similar to a `NumPy ndarray`. A TF 2 tensor is defined by its dimensionality, as illustrated here:

```
scalar number:        a zeroth-order tensor
vector:               a first-order tensor
matrix:               a second-order tensor
3-dimensional array: a 3rd order tensor
```

The next section discusses some of the data types that are available in TF 2, followed by a section that discusses TF 2 primitive types.

C4.1 TF 2 Data Types

TF 2 supports the following data types (similar to the supported data types in TensorFlow 1.x):

- `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.

C.4.2 TF 2 Primitive Types

TF 2 supports `tf.constant()` and `tf.Variable()` as primitive types. Notice the capital "V" in `tf.Variable()`: this indicates a TF 2 class (which is not the case for lowercase initial letter such as `tf.constant()`).

A TF 2 *constant* is an immutable value, and a simple example is shown here:

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

A TF 2 *variable* is a "trainable value" in a TF 2 graph. For example, the slope `m` and y-intercept `b` of a best-fitting line for a dataset consisting of points in the Euclidean 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 `z` are defined as TF variables. In addition, `b` and `x` are initialized with numeric values, whereas the value of the variable `z` is an expression that epends on the value of `x` (which equals 2).

C.5 Constants in TF 2

Here is a short list of some properties of TF 2 constants:

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

Listing C.1 displays the contents of `tf2_constants1.py` that illustrates how to assign and print the values of some TF 2 constants.

Listing C.1: tf2_constants1.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.consta
nt([[[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 C.1 contains four `tf.constant()` statements that define TF 2 tensors of dimension 0, 1, 2, and 3, respectively. The second part of Listing C.1 contains four `print()` statements that display the shape of the four TF 2 constants that are defined in the first section of Listing C.1. The output from Listing C.1 is here:

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

Listing C.2 displays the contents of `tf2_constants2.py` that illustrates how to assign values to TF 2 constants and then print those values.

Listing C.2: tf2_constants2.py

```
import tensorflow as tf

x = tf.constant(5,name="x")
y = tf.constant(8,name="y")

@tf.function
def calc_prod(x, y):
  z = 2*x + 3*y
  return z

result = calc_prod(x, y)
print('result =',result)
```

Listing C.2 defines a "decorated" (shown in bold) Python function `calc_prod()` with TF 2 code that would otherwise be included in a TF 1.x `tf.Session()` code block. Specifically, `z` would be included in a `sess.run()` statement, along with a `feed_dict` that provides values for `x` and `y`. Fortunately, a decorated Python function in TF 2 makes the code look like "normal" Python code.

C.6 Variables in TF 2

TF 2.0 eliminates global collections and their associated APIs, such as `tf.get_variable`, `tf.variable_scope`, and `tf.initializers.global_variables`. Whenever you need a `tf.Variable` in TF 2, construct and initialize it directly, as shown here:

```
tf.Variable(tf.random.normal([2, 4])
```

Listing C.3 displays the contents of `tf2_variables.py` that illustrates how to compute values involving TF constants and variables in a `with` code block.

Listing C.3: tf2_variables.py

```
import tensorflow as tf

v = tf.Variable([[1., 2., 3.], [4., 5., 6.]])
print("v.value():", v.value())
print("")
print("v.numpy():", v.numpy())
print("")

v.assign(2 * v)
v[0, 1].assign(42)
v[1].assign([7., 8., 9.])
print("v:",v)
print("")

try:
  v= [7., 8., 9.]
except TypeError as ex:
  print(ex)
```

Listing C.3 defines a TF 2 variable `v` and prints its value. The next portion of Listing C.3 updates the value of `v` and prints its new value. The last portion of Listing C.3 contains a `try/except` block that attempts to update the value of `v[1]`. The output from Listing C.3 is here:

```
v.value(): tf.Tensor(
[[1. 2. 3.]
 [4. 5. 6.]], shape=(2, 3), dtype=float32)

v.numpy(): [[1. 2. 3.]
 [4. 5. 6.]]

v: <tf.Variable 'Variable:0' shape=(2, 3) dtype=float32,
numpy=
array([[ 2., 42.,  6.],
       [ 7.,  8.,  9.]], dtype=float32)>

'ResourceVariable' object does not support item
assignment
```

This concludes the quick tour involving TF 2 code that contains various combinations of TF constants and TF variables. The next few sections delve into more details regarding the TF primitive types that you saw in the preceding sections.

C.7 The `tf.rank()` API

The *rank* of a TF 2 tensor is the dimensionality of the tensor, whereas the *shape* of a tensor is the number of elements in each dimension. Listing C.4 displays the contents of `tf2_rank.py` that illustrates how to find the rank of TF 2 tensors.

Listing C.4: tf2_rank.py

```
import tensorflow as tf # tf2_rank.py

A = tf.constant(3.0)
B = tf.fill([2,3], 5.0)
C = tf.constant([3.0, 4.0])

@tf.function
def show_rank(x):
  return tf.rank(x)

print('A:',show_rank(A))
print('B:',show_rank(B))
print('C:',show_rank(C))
```

Listing C.4 contains familiar code for defining the TF constant `A`, followed by the TF tensor `B, which` is a 2x3 tensor in which every element has the value 5. The TF tensor `C` is a 1x2 tensor with the values 3.0 and 4.0.

The next code block defines the decorated Python function `show_rank()` that returns the rank of its input variable. The final section invokes `show_rank()` with `A` and then with `B`. The output from Listing C.4 is here:

```
A: tf.Tensor(0, shape=(), dtype=int32)
B: tf.Tensor(2, shape=(), dtype=int32)
C: tf.Tensor(1, shape=(), dtype=int32)
```

C.8 The `tf.shape()` API

The *shape* of a TF 2 tensor is the number of elements in each dimension of a given tensor.

Listing C.5 displays the contents of `tf2_getshape.py` that illustrates how to find the shape of TF 2 tensors.

Listing C.5: tf2_getshape.py

```
import tensorflow as tf

a = tf.constant(3.0)
print("a shape:",a.get_shape())

b = tf.fill([2,3], 5.0)
print("b shape:",b.get_shape())

c = tf.constant([[1.0,2.0,3.0], [4.0,5.0,6.0]])
print("c shape:",c.get_shape())
```

Listing C.5 contains the definition of the TF constant `a` whose value is 3.0. Next, the TF variable `b` is initialized as a 1x2 vector with the value `[[2,3], 5.0]`, followed by the constant `c` whose value is `[[1.0,2.0,3.0],[4.0,5.0,6.0]]`. The three `print()` statements display the values of `a`, `b`, and `c`. The output from Listing C.5 is here:

```
a shape: ()
b shape: (2, 3)
c shape: (2, 3)
```

Shapes that specify a 0-D Tensor (scalar) are numbers (9, -5, 2.34, and so forth), [], and (). As another example, Listing C.6 displays the contents of `tf2_shapes.py` that contains an assortment of tensors and their shapes.

Listing C.6: tf2_shapes.py

```
import tensorflow as tf
```

```
list_0 = []
tuple_0 = ()
print("list_0:",list_0)
print("tuple_0:",tuple_0)

list_1 = [3]
tuple_1 = (3)
print("list_1:",list_1)
print("tuple_1:",tuple_1)

list_2 = [3, 7]
tuple_2 = (3, 7)
print("list_2:",list_2)
print("tuple_2:",tuple_2)

any_list1  = [None]
any_tuple1 = (None)
print("any_list1:",any_list1)
print("any_tuple1:",any_tuple1)

any_list2 = [7,None]
any_list3 = [7,None,None]
print("any_list2:",any_list2)
print("any_list3:",any_list3)
```

Listing C.6 contains simple lists and tuples of various dimensions in order to illustrate the difference between these two types. The output from Listing C.6 is probably what you would expect, and it's shown here:

```
list_0: []
tuple_0: ()
list_1: [3]
tuple_1: 3
list_2: [3, 7]
tuple_2: (3, 7)
any_list1: [None]
any_tuple1: None
any_list2: [7, None]
any_list3: [7, None, None]
```

C.9 Variables in TF 2 (Revisited)

TF 2 variables can be updated during backward error propagation. TF 2 variables can also be saved and then restored at a later point in time. The following list contains some properties of TF 2 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 simple examples of TF 2 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` ("linear model") defines a line in the Euclidean plane. Other properties of TF 2 variables are listed here:

- a tensor that's updateable via operations
- exist outside the context of `session.run`
- like a "regular" variable
- holds the learned model parameters
- variables can be shared (or nontrainable)
- used for storing/maintaining state
- internally stores a persistent tensor
- 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 2 also provides the method `tf.assign()` in order to modify values of TF 2 variables; be sure to read the relevant code sample later in this chapter so that you learn how to use this API correctly.

C.9.1 TF 2 Variables versus Tensors

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

TF *variables* represent your model's trainable parameters (e.g., 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.

In the next section you will learn about the `@tf.function` “decorator” for Python functions and how it can improve performance.

C.10 What is `@tf.function` in TF 2?

TF 2 introduced the `@tf.function` “decorator” for Python functions that defines a graph and performs session execution: it’s sort of a “successor” to `tf.Session()` in TF 1.x. Since graphs can still be useful, `@tf.function` transparently converts Python functions into functions that are "backed" by graphs. This decorator also converts tensor-dependent Python control flow into TF control flow, and also adds control dependencies to order read and write operations to TF 2 state. Remember that `@tf.function` works best with TF 2 operations instead of `NumPy` operations or Python primitives.

In general, you won't need to decorate functions with @tf.function; use it to decorate high-level computations, such as one step of training, or the forward pass of a model.

Although TF 2 eager execution mode facilitates a more intuitive user interface, this ease-of-use can be at the expense of decreased performance. Fortunately, the `@tf.function` decorator is a technique for generating graphs in TF 2 code that execute more quickly than eager execution mode.

The performance benefit depends on the type of operations that are performed: matrix multiplication does not benefit from the use of `@tf.function`, whereas optimizing a deep neural network can benefit from `@tf.function`.

C.10.1 How Does `@tf.function` Work?

Whenever you decorate a Python function with `@tf.function`, TF 2 automatically builds the function in graph mode. If a Python function that is decorated with `@tf.function` invokes other Python functions that are not decorated with `@tf.function`, then the code in those "nondecorated" Python functions will also be included in the generated graph.

Another point to keep in mind is that a `tf.Variable` in eager mode is actually a "plain" Python object: this object is destroyed when it's out of scope. However, a `tf.Variable` object defines a persistent object if the function is decorated via `@tf.function`. In this scenario, eager mode is disabled and the `tf.Variable` object defines a node in a persistent TF 2 graph. Consequently, a function that works in eager mode without annotation can fail when it is decorated with `@tf.function`.

C.10.2 A Caveat about `@tf.function` in TF 2

If constants are defined *before* the definition of a decorated Python function, you can print their values inside the function using the Python `print()` function. But if constants are defined *inside* the definition of a decorated Python function, you can print their values inside the function using the TF 2 `tf.print()` function. Consider this code block:

```
import tensorflow as tf

a = tf.add(4, 2)

@tf.function
def compute_values():
  print(a) # 6

compute_values()

# output:

# tf.Tensor(6, shape=(), dtype=int32)
```

As you can see, the correct result is displayed (shown in bold). However, if you define constants *inside* a decorated Python function, the output contains types and attributes but *not* the execution of the addition operation. Consider the following code block:

```
import tensorflow as tf

@tf.function
def compute_values():
  a = tf.add(4, 2)
  print(a)

compute_values()
```

```
# output:
# Tensor("Add:0", shape=(), dtype=int32)
```

The zero in the preceding output is part of the tensor name and not an outputted value. Specifically, `Add:0` is output zero of the `tf.add()` operation. Any additional invocation of `compute_values()` prints nothing. If you want actual results, one solution is to return a value from the function, as shown here:

```
import tensorflow as tf

@tf.function
def compute_values():
  a = tf.add(4, 2)
  return a

result = compute_values()
print("result:", result)
```

The output from the preceding code block is here:

```
result: tf.Tensor(6, shape=(), dtype=int32)
```

A second solution involves the TF `tf.print()` function instead of the Python `print()` function, as shown in bold in this code block:

```
@tf.function
def compute_values():
  a = tf.add(4, 2)
     tf.print(a)
```

A third solution is to cast the numeric values to tensors if they do not affect the shape of the generated graph, as shown here:

```
import tensorflow as tf

@tf.function
def compute_values():
  a = tf.add(tf.constant(4), tf.constant(2))
  return a

result = compute_values()
print("result:", result)
```

C.10.3 The `tf.print()` Function and Standard Error

There is one more detail to remember: the Python `print()` function "sends" output to something called "standard output" that is associated with a file descriptor whose value is 1; however, `tf.print()` sends output to "standard error" that is associated with a file descriptor whose value is 2. In programming languages such as C, only errors are sent to standard error, so keep in mind that the behavior of `tf.print()` differs from the convention regarding standard out and standard error. The following code snippets illustrate this difference:

```
python3 file_with_print.py    1>print_output
python3 file_with_tf.print.py 2>tf.print_output
```

If your Python file contains both `print()` and `tf.print()` you can capture the output as follows:

```
python3 both_prints.py 1>print_output 2>tf.print_output
```

However, keep in mind that the preceding code snippet might also redirect *real* error messages to the file `tf.print_output`.

C.11 Working with `@tf.function` in TF 2

The preceding section explained how the output will differ depending on whether you use the Python `print()` function versus the `tf.print()` function in TF 2 code, where the latter function also sends output to standard error instead of standard output.

This section contains several examples of the `@tf.function` decorator in TF 2 to show you some nuances in behavior that depend on where you define constants and whether you use the `tf.print()` function or the Python `print()` function. Also keep in mind the comments in the previous section regarding `@tf.function`, as well as the fact that you don't need to use `@tf.function` in all your Python functions.

C.11.1 An Example without `@tf.function`

Listing C.7 displays the contents of `tf2_simple_function.py` that illustrates how to define a Python function with TF 2 code.

Listing C.7: tf2_simple_function.py

```
import tensorflow as tf

def func():
  a = tf.constant([[10,10],[11.,1.]])
  b = tf.constant([[1.,0.],[0.,1.]])
  c = tf.matmul(a, b)
  return c
```

```
print(func().numpy())
```

The code in Listing C.7 is straightforward: a Python function `func()` defines two TF 2 constants, computes their product, and returns that value.

Since TF 2 works in eager mode by default, the Python function `func()` is treated as a "normal" function. Launch the code and you will see the following output:

```
[[20. 30.]
 [22. 3.]]
```

C.11.2 An Example with `@tf.function`

Listing C.8 displays the contents of `tf2_at_function.py` that illustrates how to define a decorated Python function with TF code.

Listing C.8: tf2_at_function.py

```
import tensorflow as tf

@tf.function
def func():
  a = tf.constant([[10,10],[11.,1.]])
  b = tf.constant([[1.,0.],[0.,1.]])
  c = tf.matmul(a, b)
  return c

print(func().numpy())
```

Listing C.8 defines a decorated Python function: the rest of the code is identical to Listing C.7. However, because of the `@tf.function` annotation, the Python `func()` function is "wrapped" in a `tensorflow.python.eager.def_function.Function` object. The Python function is assigned to the `.python_function` property of the object.

When `func()` is invoked, the graph construction begins. Only the Python code is executed, and the behavior of the function is traced so that TF 2 can collect the required data to construct the graph. The output is shown here:

```
[[20. 30.]
 [22.  3.]]
```

C.11.3 Overloading Functions with `@tf.function`

If you have worked with programming languages such as Java and C++, you are already familiar with the concept of “overloading” a function. If this term is new to you, the idea is simple: an overloaded function is a function that can be invoked with different data types. For example, you can define an overloaded "add" function that can add two numbers as well as "add" (i.e., concatenate) two strings.

If you're curious, overloaded functions in various programming languages are implemented via "name mangling," which means that the signature (the parameters and their data types for the function) are appended to the function name in order to generate a unique function name. This happens "under the hood," which means that you don't need to worry about the implementation details.

Listing C.9 displays the contents of `tf2_overload.py` that illustrates how to define a decorated Python function that can be invoked with different data types.

Listing C.9: tf2_overload.py

```
import tensorflow as tf

@tf.function
def add(a):
  return a + a

print("Add 1:             ", add(1))
print("Add 2.3:           ", add(2.3))
print("Add string tensor:", add(tf.constant("abc")))

c = add.get_concrete_function(tf.TensorSpec(shape=None,
dtype=tf.string))
c(a=tf.constant("a"))
```

Listing C.9 defines a decorated Python function `add()` is preceded by a `@tf.function` decorator. This function can be invoked by passing an integer, a decimal value, or a TF 2 tensor and the correct result is calculated. Launch the code and you will see the following output:

```
Add 1:             tf.Tensor(2, shape=(), dtype=int32)
Add 2.3:           tf.Tensor(4.6, shape=(),
dtype=float32)
Add string tensor: tf.Tensor(b'abcabc', shape=(),
dtype=string)
c: <tensorflow.python.eager.function.ConcreteFunction
object at 0x1209576a0>
```

C.11.4 What is AutoGraph in TF 2?

`AutoGraph` refers to the conversion from Python code to its graph representation, which is a significant new feature in TF 2. In fact, `AutoGraph` is automatically applied to functions that are decorated with `@tf.function`; this decorator creates callable graphs from Python functions.

`AutoGraph` transforms a subset of Python syntax into its portable, high-performance and language agnostic graph representation, thereby bridging the gap between TF 1.x and TF 2.0. In fact, autograph allows you to inspect its auto-generated code with this code snippet. For example, if you define a Python function called `my_product()`, you can inspect its auto-generated code with this snippet:

```
print(tf.autograph.to_code(my_product))
```

In particular, the Python `for/while` construct in implemented in TF 2 via `tf.while_loop` (break and continue are also supported). The Python `if` construct is implemented in TF 2 via `tf.cond`. The "`for _ in dataset`" is implemented in TF 2 via `dataset.reduce`.

AutoGraph also has some rules for converting loops. A `for` loop is converted if the iterable in the loop is a tensor, and a `while` loop is converted if the `while` condition depends on a tensor. If a loop is converted, it will be dynamically "unrolled" with `tf.while_loop`, as well as the special case of a `for x in tf.data.Dataset` (the latter is transformed into `tf.data.Dataset.reduce`). If a loop is not converted, it will be statically unrolled.

AutoGraph supports control flow that is nested arbitrarily deep, so you can implement many types of ML programs. Check the online documentation for more information regarding AutoGraph.

C.12 Arithmetic Operations in TF 2

Listing C.10 displays the contents of `tf2_arithmetic.py` that illustrates how to perform arithmetic operations in a TF 2.

Listing C.10: tf2_arithmetic.py

```
import tensorflow as tf

@tf.function # repłace print() with tf.print()
def compute_values():
  a = tf.add(4, 2)
```

```
    b = tf.subtract(8, 6)
    c = tf.multiply(a, 3)
    d = tf.math.divide(a, 6)

    print(a) # 6
    print(b) # 2
    print(c) # 18
    print(d) # 1

compute_values()
```

Listing C.10 defines the decorated Python function `compute_values()` with simple code for computing the sum, difference, product, and quotient of two numbers via the `tf.add()`, `tf.subtract()`, `tf.multiply()`, and the `tf.math.divide()` APIs, respectively. The four `print()` statements display the values of `a`, `b`, `c`, and `d`. The output from Listing C.10 is here:

```
tf.Tensor(6,   shape=(), dtype=int32)
tf.Tensor(2,   shape=(), dtype=int32)
tf.Tensor(18,  shape=(), dtype=int32)
tf.Tensor(1.0, shape=(), dtype=float64)
```

C.13 Caveats for Arithmetic Operations in TF 2

As you can probably surmise, you can also perform arithmetic operations involves TF 2 constants and variables. Listing C.11 displays the contents of `tf2_const_var.py` that illustrates how to perform arithmetic operations involving a TF 2 constant and a variable.

Listing C.11: tf2_const_var.py

```
import tensorflow as tf

v1 = tf.Variable([4.0, 4.0])
c1 = tf.constant([1.0, 2.0])

diff = tf.subtract(v1,c1)
print("diff:",diff)
```

Listing C.11 computes the difference of the TF variable `v1` and the TF constant `c1`, and the output is shown here:

```
diff: tf.Tensor([3. 2.], shape=(2,), dtype=float32)
```

However, if you update the value of `v1` and then print the value of `diff`, it will *not* change. You must reset the value of `diff`, just as you would in other imperative programming languages.

Listing C.12 displays the contents of `tf2_const_var2.py` that illustrates how to perform arithmetic operations involving a TF 2 constant and a variable.

Listing C.12: tf2_const_var2.py

```
import tensorflow as tf

v1 = tf.Variable([4.0, 4.0])
c1 = tf.constant([1.0, 2.0])

diff = tf.subtract(v1,c1)
print("diff1:",diff.numpy())

# diff is NOT updated:
v1.assign([10.0, 20.0])
print("diff2:",diff.numpy())

# diff is updated correctly:
diff = tf.subtract(v1,c1)
print("diff3:",diff.numpy())
```

Listing C.12 recomputes the value of `diff` in the final portion of Listing C.11, after which it has the correct value. The output is shown here:

```
diff1: [3. 2.]
diff2: [3. 2.]
diff3: [9. 18.]
```

C.13.1 TF 2 and Built-in Functions

Listing C.13 displays the contents of `tf2_math_ops.py` that illustrates how to perform additional arithmetic operations in a TF graph.

Listing C.13: tf2_math_ops.py

```
import tensorflow as tf

PI = 3.141592

@tf.function # replace print() with tf.print()
def math_values():
  print(tf.math.divide(12,8))
  print(tf.math.floordiv(20.0,8.0))
  print(tf.sin(PI))
  print(tf.cos(PI))
  print(tf.math.divide(tf.sin(PI/4.), tf.cos(PI/4.)))

math_values()
```

Listing C.13 contains a hard-coded approximation for `PI`, followed by the decorated Python function `math_values()` with five `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 C.13 is here:

```
1.5
tf.Tensor(2.0,            shape=(), dtype=float32)
tf.Tensor(6.2783295e-07,  shape=(), dtype=float32)
tf.Tensor(-1.0,           shape=(), dtype=float32)
tf.Tensor(0.99999964,     shape=(), dtype=float32)
```

Listing C.14 displays the contents of `tf2_math-ops_pi.py` that illustrates how to perform arithmetic operations in TF 2.

Listing C.14: tf2_math_ops_pi.py

```
import tensorflow as tf
import math as m

PI = tf.constant(m.pi)

@tf.function # repłace print() with tf.print()
def math_values():
  print(tf.math.divide(12,8))
  print(tf.math.floordiv(20.0,8.0))
  print(tf.sin(PI))
  print(tf.cos(PI))
  print(tf.math.divide(tf.sin(PI/4.), tf.cos(PI/4.)))

math_values()
```

Listing C.14 is almost identical to the code in Listing C.13: the only difference is that Listing C.14 specifies a hard-coded value for PI, whereas Listing C.14 assigns `m.pi` to the value of PI. As a result, the approximated value is one decimal place closer to the correct value of zero. The output from Listing C.14 is here, and notice how the output format differs from Listing C.13 due to the Python `print()` function:

```
1.5
tf.Tensor(2.0,           shape=(), dtype=float32)
tf.Tensor(-8.742278e-08, shape=(), dtype=float32)
tf.Tensor(-1.0,          shape=(), dtype=float32)
tf.Tensor(1.0,           shape=(), dtype=float32)
```

C.14 Calculating Trigonometric Values in TF 2

Listing C.15 displays the contents of `tf2_trig_values.py` that illustrates how to compute values involving trigonometric functions in TF 2.

Listing C.15: tf2_trig_values.py

```
import tensorflow as tf
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)

@tf.function # this decorator is okay
def math_values():
  print("a:",a)
  print("b:",b)
  print("c:",c)
  print("d:",d)

math_values()
```

Listing C.14 is straightforward: there are several of the same TF 2 APIs that you saw in Listing C.13. In addition, Listing C.14 contains the `tf.tan()` API, which computes the tangent of a number (in radians). The output from Listing C.14 is here:

```
a: tf.Tensor(0.49999997, shape=(), dtype=float32)
b: tf.Tensor(0.86602545, shape=(), dtype=float32)
c: tf.Tensor(2.0000002,  shape=(), dtype=float32)
d: tf.Tensor(0.57735026, shape=(), dtype=float32)
```

C.15 Calculating Exponential Values in TF 2

Listing C.15 displays the contents of `tf2_exp_values.py` that illustrates how to compute values involving additional trigonometric functions in TF 2.

Listing C.15: tf2_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)

@tf.function # this decorator is okay
def math_values():
  print('a: ', a)
  print('b: ', b)
  print('s1:', s1)
  print('s2:', s2)
  print('t2:', t2)

math_values()
```

Listing C.15 starts with the TF 2 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 output from Listing C.15 is here:

```
a:  tf.Tensor(2.7182817,  shape=(), dtype=float32)
b:  tf.Tensor(0.13533528, shape=(), dtype=float32)
s1: tf.Tensor(0.880797,   shape=(), dtype=float32)
s2: tf.Tensor(0.880797,   shape=(), dtype=float32)
t2: tf.Tensor(0.9640276,  shape=(), dtype=float32)
```

C.16 Working with Strings in TF 2

Listing C.16 displays the contents of `tf2_strings.py` that illustrates how to work with strings in TF 2.

Listing C.16: tf2_strings.py

```
import tensorflow as tf

x1 = tf.constant("café")
print("x1:",x1)
tf.strings.length(x1)
print("")

len1 = tf.strings.length(x1, unit="UTF8_CHAR")
len2 = tf.strings.unicode_decode(x1, "UTF8")

print("len1:",len1.numpy())
print("len2:",len2.numpy())
print("")
```

```
# String arrays
x2 = tf.constant(["Café", "Coffee", "caffè", "咖啡"])
print("x2:",x2)
print("")

len3 = tf.strings.length(x2, unit="UTF8_CHAR")
print("len2:",len3.numpy())
print("")

r = tf.strings.unicode_decode(x2, "UTF8")
print("r:",r)
```

Listing C.16 defines the TF 2 constant `x1` as a string that contains an accent mark. The first `print()` statement displays the first three characters of `x1`, followed by a pair of hexadecimal values that represent the accented "e" character. The second and third `print()` statements display the number of characters in `x1`, followed by the UTF8 sequence for the string `x1`.

The next portion of Listing C.16 defines the TF 2 constant `x2` as a 1st order TF 2 tensor that contains four strings. The next `print()` statement displays the contents of `x2`, using UTF8 values for characters that contain accent marks.

The final portion of Listing C.16 defines `r` as the Unicode values for the characters in the string `x2`. The output from Listing C.14 is here:

```
x1: tf.Tensor(b'caf\xc3\xa9', shape=(), dtype=string)

len1: 4
len2: [ 99  97 102 233]

x2: tf.Tensor([b'Caf\xc3\xa9' b'Coffee' b'caff\xc3\xa8'
b'\xe5\x92\x96\xe5\x95\xa1'], shape=(4,), dtype=string)

len2: [4 6 5 2]

r: <tf.RaggedTensor [[67, 97, 102, 233], [67, 111,
102, 102, 101, 101], [99, 97, 102, 102, 232], [21654,
21857]]>
```

Chapter 2 contains a complete code sample with more examples of a `RaggedTensor` in TF 2.

C.17 Working with Tensors and Operations in TF 2

Listing C.17 displays the contents of `tf2_tensors_operations.py` that illustrates how to use various operators with tensors in TF 2.

Listing C.17: tf2_tensors_operations.py

```
import tensorflow as tf

x = tf.constant([[1., 2., 3.], [4., 5., 6.]])

print("x:", x)
print("")
print("x.shape:", x.shape)
print("")
print("x.dtype:", x.dtype)
print("")
print("x[:, 1:]:", x[:, 1:])
print("")
print("x[..., 1, tf.newaxis]:", x[..., 1, tf.newaxis])
print("")
print("x + 10:", x + 10)
print("")
print("tf.square(x):", tf.square(x))
print("")
print("x @ tf.transpose(x):", x @ tf.transpose(x))

m1 = tf.constant([[1., 2., 4.], [3., 6., 12.]])
print("m1:              ", m1 + 50)
print("m1 + 50:         ", m1 + 50)
print("m1 * 2:          ", m1 * 2)
print("tf.square(m1):   ", tf.square(m1))
```

Listing C.17 defines the TF tensor `x` that contains a 2x3 array of real numbers. The bulk of the code in Listing C.17 illustrates how to display properties of x by invoking `x.shape` and `x.dtype`, as well as the TF function `tf.square(x)`. The output from Listing C.17 is here:

```
x: tf.Tensor(
[[1. 2. 3.]
 [4. 5. 6.]], shape=(2, 3), dtype=float32)

x.shape: (2, 3)

x.dtype: <dtype: 'float32'>

x[:, 1:]: tf.Tensor(
[[2. 3.]
 [5. 6.]], shape=(2, 2), dtype=float32)

x[..., 1, tf.newaxis]: tf.Tensor(
[[2.]
```

```
 [5.]], shape=(2, 1), dtype=float32)

x + 10: tf.Tensor(
[[11. 12. 13.]
 [14. 15. 16.]], shape=(2, 3), dtype=float32)

tf.square(x): tf.Tensor(
[[ 1.  4.  9.]
 [16. 25. 36.]], shape=(2, 3), dtype=float32)

x @ tf.transpose(x): tf.Tensor(
[[14. 32.]
 [32. 77.]], shape=(2, 2), dtype=float32)

m1:            tf.Tensor(
[[51. 52. 54.]
 [53. 56. 62.]], shape=(2, 3), dtype=float32)

m1 + 50:       tf.Tensor(
[[51. 52. 54.]
 [53. 56. 62.]], shape=(2, 3), dtype=float32)

m1 * 2:        tf.Tensor(
[[ 2.  4.  8.]
 [ 6. 12. 24.]], shape=(2, 3), dtype=float32)

tf.square(m1):    tf.Tensor(
[[  1.   4.  16.]
 [  9.  36. 144.]], shape=(2, 3), dtype=float32)
```

C.18 2nd Order Tensors in TF 2 (1)

Listing C.18 displays the contents of `tf2_elem2.py` that illustrates how to define a 2nd order TF tensor and access elements in that tensor.

Listing C.18: tf2_elem2.py

```
import tensorflow as tf

arr2 = tf.constant([[1,2],[2,3]])

@tf.function
def compute_values():
  print('arr2: ',arr2)
  print('[0]:  ',arr2[0])
  print('[1]:  ',arr2[1])

compute_values()
```

Listing C.18 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 C.18 is here:

```
arr2:   tf.Tensor(
[[1 2]
 [2 3]], shape=(2, 2), dtype=int32)
[0]:   tf.Tensor([1 2], shape=(2,), dtype=int32)
[1]:   tf.Tensor([2 3], shape=(2,), dtype=int32)
```

C.19 2nd Order Tensors in TF 2 (2)

Listing C.19 displays the contents of `tf2_elem3.py` that illustrates how to define a 2nd order TF 2 tensor and access elements in that tensor.

Listing C.19: tf2_elem3.py

```
import tensorflow as tf

arr3 = tf.constant([[[1,2],[2,3]],[[3,4],[5,6]]])

@tf.function # repłace print() with tf.print()

def compute_values():
  print('arr3:   ',arr3)
  print('[1]:    ',arr3[1])
  print('[1,1]:  ',arr3[1,1])
  print('[1,1,0]:',arr3[1,1,0])

compute_values()
```

Listing C.19 contains the TF constant `arr3` that is initialized with the value `[[[1,2],[2,3]],[[3,4],[5,6]]]`. The four `print()` statements display the value of `arr3`, 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 C.19 (adjusted slightly for display purposes) is here:

```
arr3:    tf.Tensor(
[[[1 2]
  [2 3]]

 [[3 4]
  [5 6]]], shape=(2, 2, 2), dtype=int32)
[1]:     tf.Tensor(
[[3 4]
 [5 6]], shape=(2, 2), dtype=int32)
[1,1]:   tf.Tensor([5 6], shape=(2,), dtype=int32)
[1,1,0]: tf.Tensor(5, shape=(), dtype=int32)
```

C.20 Multiplying Two 2nd Order Tensors in TF

Listing C.20 displays the contents of `tf2_mult.py` that illustrates how to multiply 2nd order tensors in TF 2.

Listing C.20: tf2_mult.py

```
import tensorflow as tf

m1 = tf.constant([[3., 3.]])   # 1x2
m2 = tf.constant([[2.],[2.]]) # 2x1
p1 = tf.matmul(m1, m2)        # 1x1

@tf.function
def compute_values():
  print('m1:',m1)
  print('m2:',m2)
  print('p1:',p1)

compute_values()
```

Listing C.20 contains two TF constant `m1` and `m2` that are initialized with the value `[[3., 3.]]` and `[[2.],[2.]]`. 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 C.20 is here:

```
m1: tf.Tensor([[3. 3.]], shape=(1, 2), dtype=float32)
m2: tf.Tensor(
[[2.]
 [2.]], shape=(2, 1), dtype=float32)
p1: tf.Tensor([[12.]], shape=(1, 1), dtype=float32)
```

C.21 Convert Python Arrays to TF Tensors

Listing C.21 displays the contents of `tf2_convert_tensors.py` that illustrates how to convert a Python array to a TF 2 tensor.

Listing C.21: tf2_convert_tensors.py

```
import tensorflow as tf
import numpy as np

x1 = np.array([[1.,2.],[3.,4.]])
x2 = tf.convert_to_tensor(value=x1, dtype=tf.float32)

print ('x1:',x1)
print ('x2:',x2)
```

Listing C.21 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 is the result of converting `x_data` to a TF tensor. The output from Listing C.21 is here:

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

C.21.1 Conflicting Types in TF 2

Listing C.22 displays the contents of `tf2_conflict_types.py` that illustrates what happens when you try to combine incompatible tensors in TF 2.

Listing C.22: tf2_conflict_types.py

```
import tensorflow as tf

try:
  tf.constant(1) + tf.constant(1.0)
except tf.errors.InvalidArgumentError as ex:
  print(ex)

try:
  tf.constant(1.0, dtype=tf.float64) + tf.constant(1.0)
except tf.errors.InvalidArgumentError as ex:
  print(ex)
```

Listing C.22 contains two `try/except` blocks. The first block adds two constants 1 and 1.0, which are compatible. The second block attempts to add the value 1.0 that's declared as a `tf.float64` with 1.0, which are not compatible tensors. The output from Listing C.22 is here:

```
cannot compute Add as input #1(zero-based) was expected
to be a int32 tensor but is a float tensor [Op:Add] name:
add/
cannot compute Add as input #1(zero-based) was expected
to be a double tensor but is a float tensor [Op:Add]
name: add/
```

C.22 Differentiation and `tf.GradientTape` in TF 2

Automatic differentiation (i.e., calculating derivatives) is useful for implementing ML algorithms such as back propagation for training various types of neural networks (NNs). During eager execution, the TF 2 context manager `tf.GradientTape` traces operations for computing gradients. This context manager provides a `watch()` method for specifying a tensor that will be differentiated (in the mathematical sense of the word).

The `tf.GradientTape` context manager records all forward-pass operations on a "tape." Next, it computes the gradient by "playing" the tape backward, and then discards the tape after a single gradient computation. Thus, a `tf.GradientTape` can only compute one gradient: subsequent invocations throw a runtime error. Keep in mind that the `tf.GradientTape` context manager only exists in eager mode.

Why do we need the `tf.GradientTape` context manager? Consider deferred execution mode, where we have a graph in which we know how nodes are connected. The gradient computation of a function is performed in two steps: (1) backtracking from the output to the input of the graph, and (2) computing the gradient to obtain the result.

By contrast, in eager execution the only way to compute the gradient of a function using automatic differentiation is to construct a graph. After constructing the graph of the operations executed within the `tf.GradientTape` context manager on some "watchable" element (such as a variable), we can instruct the tape to compute the required gradient. If you want a more detailed explanation, the `tf.GradientTape` documentation page contains an example that explains how and why tapes are needed.

The default behavior for `tf.GradientTape` is to "play once and then discard." However, it's possible to specify a persistent tape, which means that the values are persisted and therefore the tape can be "played" multiple times. The next section contains several examples of `tf.GradientTape`, including an example of a persistent tape.

C.23 Examples of `tf.GradientTape`

Listing C.23 displays the contents of `tf2_gradient_tape1.py` that illustrates how to invoke `tf.GradientTape` in TF 2. This example is one of the simplest examples of using `tf.GradientTape` in TF 2.

Listing C.23: tf2_gradient_tape1.py

```
import tensorflow as tf

w = tf.Variable([[1.0]])

with tf.GradientTape() as tape:
  loss = w * w

grad = tape.gradient(loss, w)
print("grad:",grad)
```

Listing C.23 defines the variable `w`, followed by a `with` statement that initializes the variable `loss` with expression `w*w`. Next, the variable `grad` is initialized with the derivative that is returned by the tape, and then evaluated with the current value of `w`.

As a reminder, if we define the function `z = w*w`, then the first derivative of `z` is the term `2*w` , and when this term is evaluated with the value of 1.0 for `w`, the result is 2.0. Launch the code in Listing C.23 and you will see the following output:

```
grad: tf.Tensor([[2.]], shape=(1, 1), dtype=float32)
```

C.23.1 Using the `watch()` Method of `tf.GradientTape`

Listing C.24 displays the contents of `tf2_gradient_tape2.py` that also illustrates the use of `tf.GradientTape` with the `watch()` method in TF 2.

Listing C.24: tf2_gradient_tape2.py

```
import tensorflow as tf

x = tf.constant(3.0)

with tf.GradientTape() as g:
  g.watch(x)
  y = 4 * x * x
dy_dx = g.gradient(y, x)
```

Listing C.24 contains a similar `with` statement as Listing C.23, but this time a `watch()` method is also invoked to "watch" the tensor `x`. As you saw in the previous section, if we define the function `y = 4*x*x`, then the first derivative of `y` is the term `8*x`; when the latter term is evaluated with the value `3.0`, the result is 24.0.

Launch the code in Listing C.24 and you will see the following output:

```
dy_dx: tf.Tensor(24.0, shape=(), dtype=float32)
```

C.23.2 Using Nested Loops with `tf.GradientTape`

Listing C.25 displays the contents of `tf2_gradient_tape3.py` that also illustrates how to define nested loops with `tf.GradientTape` in order to calculate the first and the second derivative of a tensor in TF 2.

Listing C.25: tf2_gradient_tape3.py

```
import tensorflow as tf

x = tf.constant(4.0)
with tf.GradientTape() as t1:
  with tf.GradientTape() as t2:
    t1.watch(x)
    t2.watch(x)
    z = x * x * x
  dz_dx = t2.gradient(z, x)
d2z_dx2 = t1.gradient(dz_dx, x)

print("First  dz_dx:  ",dz_dx)
print("Second d2z_dx2:",d2z_dx2)

x = tf.Variable(4.0)
with tf.GradientTape() as t1:
  with tf.GradientTape() as t2:
    z = x * x * x
  dz_dx = t2.gradient(z, x)
d2z_dx2 = t1.gradient(dz_dx, x)

print("First  dz_dx:  ",dz_dx)
print("Second d2z_dx2:",d2z_dx2)
```

The first portion of Listing C.25 contains a nested loop, where the outer loop calculates the first derivative and the inner loop calculates the second derivative of the term x*x*x when x equals 4. The second portion of Listing C.25 contains another nested loop that produces the same output with slightly different syntax.

In case you're a bit rusty regarding derivatives, the next code block shows you a function z, its first derivative z', and its second derivative z'':

```
z   = x*x*x
z'  = 3*x*x
z'' = 6*x
```

When we evaluate z, z', and z'' with the value 4.0 for x, the result is 64.0, 48.0, and 24.0, respectively. Launch the code in Listing C.25 and you will see the following output:

```
First  dz_dx:   tf.Tensor(48.0, shape=(), dtype=float32)
Second d2z_dx2: tf.Tensor(24.0, shape=(), dtype=float32)
First  dz_dx:   tf.Tensor(48.0, shape=(), dtype=float32)
Second d2z_dx2: tf.Tensor(24.0, shape=(), dtype=float32)
```

C.23.3 Other Tensors with `tf.GradientTape`

Listing C.26 displays the contents of tf2_gradient_tape4.py that illustrates how to use tf.GradientTape in order to calculate the first derivative of an expression that depends on a 2x2 tensor in TF 2.

Listing C.26: tf2_gradient_tape4.py

```
import tensorflow as tf

x = tf.ones((3, 3))

with tf.GradientTape() as t:
  t.watch(x)
  y = tf.reduce_sum(x)
  print("y:",y)
  z = tf.multiply(y, y)
  print("z:",z)
  z = tf.multiply(z, y)
  print("z:",z)

# the derivative of z with respect to y
dz_dy = t.gradient(z, y)
print("dz_dy:",dz_dy)
```

In Listing C.26, y equals the sum of the elements in the 3x3 tensor x, which is 9.

Next, z is assigned the term y*y and then multiplied again by y, so the final expression for z (and its derivative) is here:

```
z  = y*y*y
z' = 3*y*y
```

When z' is evaluated with the value 9 for y, the result is 3*9*9, which equals 243. Launch the code in Listing C.26 and you will see the following output (slightly reformatted for readability):

```
y: tf.Tensor(9.0,       shape=(), dtype=float32)
z: tf.Tensor(81.0,      shape=(), dtype=float32)
z: tf.Tensor(729.0,     shape=(), dtype=float32)
dz_dy: tf.Tensor(243.0, shape=(), dtype=float32)
```

C.23.4 A Persistent Gradient Tape

Listing C.27 displays the contents of tf2_gradient_tape5.py that illustrates how to define a persistent gradient tape in order to with tf.GradientTape in order to calculate the first derivative of a tensor in TF 2.

Listing C.27: tf2_gradient_tape5.py

```
import tensorflow as tf

x = tf.ones((3, 3))

with tf.GradientTape(persistent=True) as t:
  t.watch(x)
  y = tf.reduce_sum(x)
  print("y:",y)
  w = tf.multiply(y, y)
     print("w:",w)

  z = tf.multiply(y, y)
  print("z:",z)
  z = tf.multiply(z, y)
  print("z:",z)

# the derivative of z with respect to y
dz_dy = t.gradient(z, y)
print("dz_dy:",dz_dy)
dw_dy = t.gradient(w, y)
print("dw_dy:",dw_dy)
```

Listing C.27 is almost the same as Listing C.26: the new sections are displayed in bold. Note that w is the term y*y and therefore the first derivative `w'` is `2*y`. Hence, the values for `w` and `w'` are 81 and 18, respectively, when they are evaluated with the value 9.0. Launch the code in Listing C.27 and you will see the following output (slightly reformatted for readability), where the new output is shown in bold:

```
y: tf.Tensor(9.0,       shape=(), dtype=float32)
w: tf.Tensor(81.0,      shape=(), dtype=float32)
z: tf.Tensor(81.0,      shape=(), dtype=float32)
z: tf.Tensor(729.0,     shape=(), dtype=float32)
dz_dy: tf.Tensor(243.0, shape=(), dtype=float32)
dw_dy: tf.Tensor(18.0,  shape=(), dtype=float32)
```

C.24 Google Colaboratory

Depending on the hardware, GPU-based TF 2 code is typically at least 15 times faster than CPU-based TF 2 code. 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 TF 2).

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

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

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

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

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

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

As mentioned earlier, there is one other *very* nice feature of Google Colaboratory: you can execute code on a GPU for up to twelve hours per day for free. This free GPU support is extremely useful for people who don't have a suitable GPU on their local machine (which is probably the majority of users), and now they launch TF 2 code to train neural networks in less than 20 or 30 minutes that would otherwise require multiple hours of CPU-based execution time.

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

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

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

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

```
!pip install tflearn
```

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

Incidentally, you can also run TF 2 code and TensorBoard in Google Colaboratory, with support for CPUs and GPUs (and support for TPUs will be available later). Native to this link for more information:

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

C.25 Other Cloud Platforms

GCP (Google Cloud Platform) is a cloud-based service that enables you to train TF 2 code in the cloud. GCP provides 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 TF 2 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.

C.25.1 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 USD 300 dollars worth of credit (over one year) if you have never used Google cloud.

C.26 Summary

This chapter introduced you to TF 2, a very brief view of its architecture, and some of the tools that are part of the TF 2 "family." Then you learned how to write basic Python scripts containing TF 2 code with TF constants and variables. You also learned how to perform arithmetic operations and also some built-in TF functions.

Next, you learned how to calculate trigonometric values, how to use for loops, and how to calculate exponential values. You also saw how to perform various operations on 2nd order TF 2 tensors. In addition, you saw code samples that illustrate how to use some of the new features of TF 2, such as the `@tf.function` decorator and `tf.GradientTape`.

Then you got an introduction to Google Colaboratory, which is a cloud-based environment for machine learning and deep learning. This environment is based on Jupyter notebooks, with support for Python and various other languages. Google Colaboratory also provides up to 12 hours of free GPU use on a daily basis, which is a very nice feature.

INDEX