Python for Programmers

O. Campesato

PYTHON
FOR
PROGRAMMERS

LICENSE, DISCLAIMER OF LIABILITY, AND LIMITED WARRANTY

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

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

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

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

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

PYTHON FOR PROGRAMMERS

OSWALD CAMPESATO

MERCURY LEARNING AND INFORMATION
Dulles, Virginia
Boston, Massachusetts
New Delhi

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

O. Campesato. *Python for Programmers.*
ISBN: 978-1-68392-817-1

Library of Congress Control Number: 2022940434

222324321 This book is printed on acid-free paper in the United States of America.

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

CONTENTS

PREFACE

WHAT IS THE PRIMARY VALUE PROPOSITION FOR THIS BOOK?

This book contains a fast-paced introduction to Python and Python-based solutions to various tasks. Some topics are presented in a high-level manner for two main reasons. First, it's important that you be exposed to these concepts. In some cases you will find topics that might pique your interest, and hence motivate you to learn more about them through self-study; in other cases you will probably be satisfied with a brief introduction. In other words, you will decide whether or not to delve into more detail regarding the topics in this book.

Second, a full treatment of all the topics that are covered in this book would significantly increase the size of this book, and few people are interested in reading technical tomes.

THE TARGET AUDIENCE

This book is intended primarily for developers who have little or no experience with Python and are interested in learning about Python as well as an introduction to Pandas.

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

WHAT WILL I LEARN FROM THIS BOOK?

The first chapter contains a quick tour of basic Python3, followed by a chapter that shows you how to work with loops and conditional logic in Python. The third chapter discusses data structures in Python, followed by a chapter that contains code samples for tasks that involve strings and arrays in Python.

The fifth chapter contains fundamental concepts in OOP (object-oriented programming), along with code samples that illustrate how they are implemented in Python. The sixth chapter introduces you to recursion and some fundamental topics in combinatorics. Finally, the Appendix contains an introduction to Pandas.

GETTING THE MOST FROM THIS BOOK

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

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

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

WHAT DO I NEED TO KNOW FOR THIS BOOK?

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

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

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

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

DOES THIS BOOK CONTAIN PRODUCTION-LEVEL CODE SAMPLES?

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

WHAT ARE THE NON-TECHNICAL PREREQUISITES FOR THIS BOOK?

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

HOW DO I SET UP A COMMAND SHELL?

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

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

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

If you are a PC user, you can install Cygwin (open source *https://cygwin.com/*) that simulates bash commands, or use another toolkit such as MKS (a commercial product). Please read the online documentation that describes the download and installation process. Note that custom aliases are not automatically set if they are defined in a file other than the main start-up file (such as .bash_login).

COMPANION FILES

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

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

The answer to this question varies widely, mainly because the answer depends heavily on your objectives. If you are primarily interested in machine learning, there are some subfields of machine learning, such as deep learning and reinforcement learning (and deep reinforcement learning) that might appeal to you. Fortunately, there are many resources available, and you can perform an Internet search for those resources. One other point: the aspects of machine learning for you to learn depend on who you are: the needs of a machine learning engineer, data scientist, manager, student or software developer are all different.

CHAPTER 1

INTRODUCTION TO PYTHON

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

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

NOTE

The Python scripts in this book are for Python 3.

TOOLS FOR PYTHON

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

http://continuum.io/downloads

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

easy_install and pip

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

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

NOTE

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

virtualenv

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

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

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

IPython

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

http://ipython.org/install.html

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

```
ipython
```

The preceding command displays the following output:

```
Python 3.9.13 (main, May 24 2022, 21:28:12)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.22.0 -- An enhanced Interactive Python. Type '?' for help.


In [1]:
```

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

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

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

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

GETTING HELP
------------

Within IPython you have various way to access help:

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

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

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

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

PYTHON INSTALLATION

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

```
python -V
```

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

```
Python 3.9.1
```

NOTE

Install Python 3.9.13 (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.

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 good guide to setting up your environment so that the `Python` executable is always available in every command shell is to follow the instructions here:

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

LAUNCHING PYTHON ON YOUR MACHINE

There are three different ways to launch Python:

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

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

NOTE

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

The Python Interactive Interpreter

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

```
python
```

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

```
Python 3.9.1 (v3.9.1:1e5d33e9b9, Dec 7 2020, 12:44:01)
[Clang 12.0.0 (clang-1200.0.32.27)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Type the expression `2 + 7` at the prompt:

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

Python displays the following result:

```
9
>>>
```

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

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

```
python myscript.py
```

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

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

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

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

PYTHON IDENTIFIERS

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

- It starts with a letter A to Z, a to z, or an underscore (_)
- It includes 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 are used for strongly private identifiers

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

LINES, INDENTATION, AND 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.

QUOTATION AND COMMENTS IN PYTHON

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

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

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

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

The output of the preceding code block is here:

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

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

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

The output of the preceding code block is here:

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

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

```
#!/usr/bin/python
# First comment
```

```
print("Hello, Python!")  # second comment
```

This will produce following result:

```
Hello, Python!
```

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

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

You can comment multiple lines as follows:

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

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

SAVING YOUR CODE IN A MODULE

Earlier you saw how to launch the Python interpreter from the command line and then enter Python commands. However, 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)
```

Type the following command:

```
python first.py
```

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

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

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

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

SOME STANDARD MODULES IN PYTHON

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

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

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

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

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

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

THE HELP() AND DIR() FUNCTIONS

An internet search for Python-related topics usually returns a number of links with useful

information. Alternatively, you can check the official Python documentation site: docs.python.org

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

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

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

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

The preceding command generates the following output:

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

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

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

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

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

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

The output from the preceding command is here:

```
Help on method_descriptor:


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

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

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

COMPILE TIME AND RUNTIME CODE CHECKING

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

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

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

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

You should now have an understanding of some basic concepts, such as how to use the Python interpreter and how to launch your custom Python modules. The next section discusses primitive data types in Python.

SIMPLE DATA TYPES IN PYTHON

Python supports primitive data types, such as numbers (integers, floating point numbers, and exponential numbers), strings, and dates. Python also supports more complex data types,

such as lists (or arrays), tuples, and dictionaries. The next several sections discuss some of the Python primitive data types, along with code snippets that show you how to perform various operations on those data types.

WORKING WITH NUMBERS

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

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

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

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

The following examples demonstrate arithmetic operations involving integers:

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

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

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

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

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

The output from the preceding code block is here:

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

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

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

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

Working With Other Bases

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

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

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

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

Negative integers are displayed with a negative sign:

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

The `chr()` Function

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

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

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

However, the following equivalent code snippet is more intuitive:

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

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

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

The round() Function in Python

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

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

Formatting Numbers in Python

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

```
>>> x = 1.23456
>>> format(x, '0.2f')
'1.23'
>>> format(x, '0.3f')
'1.235'
>>> 'value is {:0.3f}'.format(x) 'value is 1.235'
>>> from decimal import Decimal
>>> a = Decimal('4.2')
>>> b = Decimal('2.1')
>>> a + b
Decimal('6.3')
>>> print(a + b)
6.3
>>> (a + b) == Decimal('6.3')
True
>>> x = 1234.56789
>>> # Two decimal places of accuracy
>>> format(x, '0.2f')
'1234.57'
>>> # Right justified in 10 chars, one-digit accuracy
>>> format(x, '>10.1f')
' 1234.6'
>>> # Left justified
>>> format(x, '<10.1f') '1234.6 '
>>> # Centered
>>> format(x, '^10.1f') ' 1234.6 '
>>> # Inclusion of thousands separator
>>> format(x, ',')
'1,234.56789'
>>> format(x, '0,.1f')
'1,234.6'
```

WORKING WITH FRACTIONS

Python supports the Fraction() function (defined in the fractions module) that accepts two integers that represent the numerator and the denominator (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.

UNICODE AND UTF-8

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

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

WORKING WITH UNICODE

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

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

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

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

Listing 1.1 displays the contents of Unicode1.py that illustrates how to display a string of characters in Japanese (Hiragana) 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 1.1 is here:

```
Chinese: 將探討 HTML5 及其他
Hiragana: D3 は かっこいい です!
```

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

WORKING WITH STRINGS

A string in Python3 is based on Unicode, whereas a string in Python2 is a sequence of ASCII-encoded bytes. You can concatenate two strings using the '+' operator. The following example prints a string and then concatenates two single-letter strings:

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

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

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

You can assign strings to variables and print them using the `print` command:

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

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

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

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

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

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

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

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

Comparing Strings

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

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

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

Listing 1.2: Compare.py

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

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

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

```
x and y: different
```

Formatting Strings in Python

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

```
import string

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

The output is shown here:

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

UNINITIALIZED VARIABLES AND THE VALUE NONE IN PYTHON

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

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

SLICING AND SPLICING STRINGS

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

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

The output from the preceding code block is here:

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

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

Testing for Digits and Alphabetic Characters

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

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

Listing 1.3: CharTypes.py

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

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

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

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

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

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

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

Listing 1.3 initializes some variables, followed by 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
```

SEARCH AND REPLACE A STRING IN OTHER STRINGS

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

Listing 1.4: FindPos1.py

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

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

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

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

The output from launching Listing 1.4 is here:

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

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

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

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

Listing 1.5: Replace1.py

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

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

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

REMOVE LEADING AND TRAILING CHARACTERS

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

Listing 1.6: Remove1.py

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

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

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

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

The output from launching Listing 1.6 is here:

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

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

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

print(a)
print(b)
```

The result is here:

```
ab
a b
```

PRINTING TEXT WITHOUT 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, execute the following statements in `Python`:

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

The output is shown here:

```
x:  9255-3.1
```

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

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

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

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

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

The output is here:

```
1233
1234567899
```

TEXT ALIGNMENT

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

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

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

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

WORKING WITH DATES

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

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 executing 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
```

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: 2905 days, 11:12:28.678439
```

EXCEPTION HANDLING IN PYTHON

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

An example of a try/except block is here:

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

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

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

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

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

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

The output from the preceding code block is here:

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

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

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

Listing 1.10: Exception1.py

```
import sys

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

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

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

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

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

HANDLING USER INPUT

Python enables you to read user input from the command line via the input() function or the raw_input() function. Typically, you assign user input to a variable, which will contain all characters that users enter from the keyboard. User input terminates when users press the <return> key (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 run-time.

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.

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))
```

Run above script as follows:

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

This will produce following result:

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

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

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

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

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

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

Listing 1.14: Hello.py

```
import sys

def main():
  if len(sys.argv) >= 2:
    name = sys.argv[1]
  else:
    name = 'World'
  print('Hello', name)

# Standard boilerplate to invoke the main() function
if __name__ == '__main__':
  main()
```

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

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

SUMMARY

This chapter showed you how to execute Python programs, as well as how to work with numbers and perform arithmetic operations on numbers in Python. Next, you learned how to work with strings and use string operations.

In addition, you learned about the difference between Unicode and ASCII in `Python` 3 and `Python` 2, respectively. Then you saw how to slice and splice strings, how to replace a string with another string, and also how to remove leading and trailing characters in a string.

Finally, you learned how to work with dates in `Python`, and then how to handle exceptions that can arise from user input.

CHAPTER 2

CONDITIONAL LOGIC IN PYTHON

This chapter is a continuation of the previous chapter, which discusses conditional logic, for loops, and `while` loops. Various code samples are included to illustrate each of these topics in `Python`.

The first part of this chapter briefly discusses precedence of operators as well as reserved words in `Python`. Next are `Python`-based code samples of various types of loops, including nested loops.

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

NOTE

The scripts in this book are for `Python` 3.x.

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)
```

The following website contains precedence rules for operators in `Python`:

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

PYTHON RESERVED WORDS

Every programming language has a set of reserved words that cannot be used as identifiers, and `Python` is no exception. Python's 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 script `test1.py` with the following code:

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

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

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

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

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.

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 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.

A For Loop with try/except in Python

Listing 2.1 shows the content 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 are 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
```

Numeric Exponents in Python

Listing 2.2 shows the content 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 as follows:

```
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
-----------
```

NESTED LOOPS

Listing 2.3 shows the content 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 2.3 is as follows:

```
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
```

THE SPLIT() FUNCTION WITH F0R 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 by invoking the `join()` function, thereby creating a line of text with one white space between any two words.

USING THE SPLIT() FUNCTION TO COMPARE WORDS

Listing 2.4 shows the content of `Compare2.py`, which 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 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 2.4 is here:

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

USING THE SPLIT() FUNCTION TO PRINT JUSTIFIED TEXT

Listing 2.5 shows the content of `FixedColumnCount.py`, which 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. This occurs whenever the variable `wordCount` is even. The output

from Listing 2.5 is as follows:

```
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
```

USING THE SPLIT() FUNCTION TO PRINT FIXED WIDTH TEXT

Listing 2.6 shows the content 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
```

USING THE SPLIT() FUNCTION TO COMPARE TEXT STRINGS

Listing 2.7 shows the content of `CompareStrings1.py`, which illustrates how to determine whether 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 `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 as follows:

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

USING THE SPLIT() FUNCTION TO DISPLAY CHARACTERS IN A STRING

Listing 2.8 shows the content of `StringChars1.py`, which 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)
```

THE JOIN() FUNCTION

Another way to remove extraneous spaces is to use the `join()` function:

```
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 as follows:

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

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 indentations instead of the curly braces that are used in other languages such as `JavaScript` and `Java`. Although the `list` data structure is not discussed until later in this chapter, the following simple code block 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 to process words and characters in a text string. The next part of this chapter shows examples of using conditional logic.

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:

```
>>> 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.

THE BREAK/CONTINUE/PASS STATEMENTS

The break statement 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 shows the content 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 as follows:

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

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.

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 two objects are the same object, which is important 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.

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 nonnull value is string2, the value of str4 is equal to string2.

LOCAL AND GLOBAL VARIABLES

Python variables can be local or global. A 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 nonlocal variable. You can specify a variable as nonlocal 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
```

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 a 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 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.py` 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 that 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 you reach the global scope (i.e., the module level).
4. If x still hasn't been found, Python checks `builtins_`

As a simple illustration, launch the Python interpreter and type the text shown in bold:

```
Python 3.9.1 (v3.9.1:1e5d33e9b9, Dec  7 2020, 12:44:01)
[Clang 12.0.0 (clang-1200.0.32.27)] 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
```

NOTE

You can access the `dict`*s that* `Python` *uses to track local and global scope by invoking* `locals()` *and* `globals()`*, respectively.*

PASS BY REFERENCE VERSUS VALUE

All parameters in the `Python` language are passed by reference. Therefore, if you change the value of a parameter inside 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.

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
```

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 as follows:

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

Listing 2.11 shows the content of `Divisors2.py`, which 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):
  divList= ""
  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 2.11 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 as follows:

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

Using a `while` Loop to Find Prime Numbers

Listing 2.12 shows the content of `Divisors3.py`, which contains a `while` loop, conditional logic, and the `%` (modulus) operator 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')
```

Launch the code in Listing 2.12 and you will see the following output:

```
12 is not prime
```

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:

- 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.

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. 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
```

SPECIFYING DEFAULT VALUES IN A FUNCTION

Listing 2.13 shows the content of DefaultValues.py, which 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
```

Returning Multiple Values From a Function

This task is accomplished by the code in Listing 2.14, which shows the content 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 as follows:

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

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 as follows:

```
s1 =  3
```

However, the `sum()` function in the preceding code block can only be used for two numeric values. Listing 2.15 shows the content of `VariableSum1.py`, which 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 the variable `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
```

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 how to use string operations. (The next chapter shows you how to work with conditional statements, loops, and user-defined functions.)

Next, you learned about condition logic, such as `if/elif` statements. You also learned how to work with loops, including `for` loops and `while` loops. In addition, you saw how to compute various values, such as the factorial value of a positive integer and a set of Fibonacci numbers.

CHAPTER 3

DATA STRUCTURES IN PYTHON

This chapter introduces an assortment of Python data structures; including lists, vectors, matrices, queues, tuples, dictionaries, and functional programming.

The first part of this chapter discusses lists and operations such as splicing and updating lists. The second portion of the chapter shows you how to work vectors, matrices, and queues.

The third portion of the chapter discusses tuples, sets, and dictionaries. The final part of this chapter discusses functional programming in Python. You will see Python code samples that illustrate how to define lambda expressions and how to use the `map()` function and `filter()` function in Python.

NOTE

The scripts in this book are for Python 3.x.

With the preceding points in mind, let's take a look at the list data type in Python, which is discussed in the next section.

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.

Lists and Basic Operations

A list consists of comma-separated values enclosed in a pair of square brackets. The following examples illustrate the syntax for defining a list, and also how to perform various operations on a 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, (2023,01,31)]
>>> x1,x2,x3,date1 = line
>>> x1
'a'
>>> x2
10
>>> x3
20
>>> date1
(2023, 1, 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, (2023,01,31)]
>>> x1,x2,x3,(year,month,day) = line
>>> x1
'a'
>>> x2
10
>>> x3
20
>>> year
2023
>>> month
1
>>> 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, (2023,01,31)]
>>> x1,x2 = line
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: too many values to unpack
```

Reversing and Sorting a List

The `reverse()` method reverses the contents of a list:

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

The `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:

```
>>> 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 code 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 as follows:

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

Lists and Arithmetic Operations

The minimum value of a list of numbers is the first number 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 as follows:

```
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.

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 follows:

```
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]
```

Calculating Squares and Cubes 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]
```

SORTING LISTS OF NUMBERS AND STRINGS

Listing 3.2 shows the content of the script `Sorted1.py` that determines whether 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 as follows:

```
[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 letters (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']
```

CONCATENATING A LIST OF WORDS

`Python` provides the `join()` method for concatenating text strings, and some examples are 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. Either of the following is preferred:

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

The following is the most inefficient way to do so:

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

THE PYTHON RANGE() FUNCTION

In this section, we discuss the `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.

Counting Digits, Uppercase, and Lowercase Letters

Listing 3.3 shows the content of the `Python` script `CountCharTypes.py` that counts the occurrences of digits and letters in a string.

Listing 3.3: 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
  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 3.3 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 the character is uppercase or lowercase. In all cases, the values of the appropriate counter-related variables

are incremented. The output of Listing 3.3 is here:

```
Original String:    abc4234AFde
Number of digits:   4
Total alphanumeric: 7
Upper Case Count:   2
Lower Case Count:   5
```

LISTS 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 lists:

```
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 `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.

WORKING WITH LISTS AND THE SPLIT() FUNCTION

You can use the `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 as follows:

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

You can also search for a word in a string:

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

COUNTING WORDS IN A LIST

Python provides the `Counter` class that enables you to count the words in a list. Listing 3.4 shows the content of `CountWord2.py` that displays the top three words with the greatest frequency.

Listing 3.4: 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 3.4 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 3.4 is here:

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

ITERATING THROUGH PAIRS OF LISTS

`Python` supports operations on pairs of lists, which means that you can perform vector-like operations. Let's first look at the following snippet that 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]
```

LIST SLICES

`Python` enables you to extract a "slice" of a list and also update portions of a list. Listing 3.5 displays the content of `list_slices1.py` that illustrates how to extract a substring from a `Python` list.

Listing 3.5: list_slices1.py

```
list1 = list(range(0,8))
print("list1:",list1)

list1[:4] = [20,20,20,20]
print("list1:",list1)

list1[:4] = [400,300,200,100]
print("list1:",list1)

list1[-1] = [5000]
print("list1:",list1)
```

Listing 3.5 initializes the variable `list1` with the integers from 0 to 7 and displays its contents. The next code snippet inserts four occurrences of the value 20 at the beginning of `list1`, followed by a code snippet that appends the values 400, 300, 200, and 100 to `list1`. The final code snippet appends the item [5000] to `list1`. Launch Listing 3.5 and you will see the following output:

```
list1: [0, 1, 2, 3, 4, 5, 6, 7]
List1: [20, 20, 20, 20, 4, 5, 6, 7]
list1: [400, 300, 200, 100, 4, 5, 6, 7]
list1: [400, 300, 200, 100, 4, 5, 6, [5000]]
```

Listing 3.6 displays the content of `substrings1.py` that illustrates how to split a text string into substrings and recombine the substrings with a new character.

Listing 3.6: substrings1.py

```
my_str = "I love Chicago deep dish pizza"
str1 = my_str[:5]
str2 = my_str[6:20]
str3 = my_str[-5:]

print("my_str:",my_str)
print("str1:  ",str1)
print("str2:  ",str2)
print("str3:  ",str3)

idx1 = my_str.find("love")
idx2 = idx1+15
str4 = my_str[idx1:idx2]
print("str4:  ",str4)
```

Listing 3.6 initializes the variable `my_str` as a text string, followed by the variables `str1`, `str2`, and `str3`. Launch Listing 3.6 and you will see the following output:

```
my_str: I love Chicago deep dish pizza
str1:   I lov
str2:    Chicago deep
str3:   pizza
str4:   love Chicago de
```

Listing 3.7 displays the content of `substrings2.py` that illustrates how to split a text string into substrings and recombine the substrings with a new character.

Listing 3.7: substrings2.py

```
my_str = "I love Chicago deep dish pizza"
idx1 = my_str.find("d")
str1 = my_str[0:idx1]
str2 = my_str[idx1+1:]
char = "l"
str3 = str1+char+str2

print("my_str:",my_str)
print("str1:  ",str1)
print("str2:  ",str2)
print("str3:  ",str3)
```

Listing 3.7 initializes the variable `my_str` as a text string, followed by the variables `str1`, `str2`, and `str3`. Launch Listing 3.7 and you will see the following output:

```
my_str: I love Chicago deep dish pizza
str1:   I love Chicago
str2:   eep dish pizza
str3:   I love Chicago leep dish pizza
```

OTHER LIST-RELATED FUNCTIONS

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

Define a 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 work with vectors.

WORKING WITH VECTORS

A vector is a one-dimensional array of values, and you can perform vector-based operations, such as addition, subtraction, and the inner product. Listing 3.8 shows the content of `MyVectors.py` that illustrates how to perform vector-based operations.

Listing 3.8: 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 3.8 starts with the definition of three lists, 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 3.8 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
```

WORKING WITH MATRICES

A two-dimensional matrix is a two-dimensional array of values. 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 as follows:

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

Listing 3.9 shows the content of `My2DMatrix.py` that illustrates how to create and populate 2 two-dimensional matrices.

Listing 3.9: 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 3.9 initializes the variables rows and columns and then uses them to create the `rows x cols` matrix `my2DMatrix` whose values are initially 0. The next part of Listing 3.9 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 3.9 prints the contents of `my2DArray`. The output from Listing 3.9 is here:

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

QUEUES

A queue is a FIFO ("First In, First Out") data structure. 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 list to emulate a queue. However, there is also a queue object in `Python`. The following code snippets illustrate how to use a queue.

```
>>> 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)
```

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 tuples can be found online:

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.

Define a 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:

```
>>> 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)`.

SETS

A *set* 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 snippets illustrate how to work with a 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'])
```

DICTIONARIES

`Python` has a key/value structure called a dictionary (`dict`) that is a hash table. A 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 `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 `{}`.

Creating a Dictionary

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

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

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

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

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

Dictionaries also use the `get` method to retrieve key values:

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

The `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}
```

Checking for Keys in a Dictionary

You can check for the presence of a key in a dictionary:

```
>>> '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 dictionary. 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.

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
```

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 as follows:

```
y 7
x 5
z 13
```

Interpolating Data From a Dictionary

The `%` operator substitutes values from a dictionary into a string by name. Listing 3.10 contains an example of doing so.

Listing 3.10: 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 Listing 3.10 is here:

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

DICTIONARY FUNCTIONS AND METHODS

`Python` provides various functions and methods, 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 dictionary using the functions `clear()` to remove all elements, `copy()` to return a copy, `get()` to retrieve the value of a key, `items()` to display the `(key,value)` pairs of a dictionary, `keys()` to display the keys of a dictionary, and `values()` to return the list of values of a dictionary.

ORDERED DICTIONARIES

Regular dictionaries iterate over key/value pairs in arbitrary order. `Python` 2.7 introduced a new `OrderedDict` class in the collections module. The `OrderedDict` 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)]
```

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.

Dictionary Formatting

The % operator works conveniently to substitute values from a dictionary 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'
```

Python Multi Dictionaries

You can define entries in a dictionary so that they reference lists or other types of structures. Listing 3.11 shows the content of MultiDictionary1.py that illustrates how to define more complex dictionaries.

Listing 3.11: 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 3.11 starts by defining the dictionary d and printing its contents. The next portion of Listing 3.11 specifies a list-oriented dictionary, and then modifies the values for the keys a and b. The final portion of Listing 3.11 specifies a set-oriented dictionary, and then modifies the values for the keys a and b as well.

The output from Listing 3.11 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])})
```

OTHER SEQUENCE TYPES IN PYTHON

Python supports seven 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 types that you will encounter in the remaining chapters of this book. In addition, it makes sense to discuss mutable and immutable types, which is the topic of the next section.

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.

There is a 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 3.12.

Listing 3.12: 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 3.12 is here:

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

A 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 type is mutable. The immutable objects are of type bytes, complex, float, int, str, or tuple. 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 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 as follows:

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

PACKING/UNPACKING SEQUENCES

Python supports useful functionality regarding sequence types that simplify the task of assigning variables to values, which can be assigned directly or as the return values of a function. One type is called direct assignment and another type pertains to assigning variables to the return values of a function, both of which are discussed in the following subsections.

Automatic Packing (Direct Assignment)

The following code snippet illustrates direct assignment and is called *automatic packing* of a tuple:

```
tens = 10,20,30,40,50
```

The variable tens in the preceding code snippet is a tuple. Another example of direct assignment is shown here:

```
x,y,z = range(0,3)
```

The variables x, y, and z in the preceding code snippet are assigned the values 0, 1, and 2, respectively.

Unpacking Return Values of Functions

The following code block illustrates how to assign variables to the return values of a function:

```
def myfunc():
  # do various things
  return 1,2
x,y = myfunc()
```

As you can see, the variables x and y are initialized with the values 1 and 2, respectively. A more interesting example is the following code block that is a variation of the preceding code block:

```
def myfunc2():
  # do various things
  return [1,2,3], 4, 5
x,y,z = myfunc2()
```

In the preceding code sample the variables `x`, `y`, and `z` are initialized with the values [1,2,3], 4, and 5, respectively.

Swapping Pairs of Values

`Python` makes it very easy to swap the values of two variables, as shown in the following code block:

```
# x is 5 and y is 8:
x = 5
y = 8
# now x = 8 and y = 5:
y,x = x,y
```

Iterating Sequences in Loops

The following code snippet shows you how to iterate over a list whose elements are pairs of values:

```
for x, y in [ (1, 2), (30, 60), (40, 80) ]:
  print("x:",x,"y:",y)
```

The following code snippet shows you how to extract values from a list in which each element is a pair of values:

```
xlist = list()
ylist = list()
for x, y in [ (1, 2), (30, 60), (40, 80) ]:
  xlist.append(x)
  ylist.append(y)

print("xlist:",xlist)
print("ylist:",ylist)
```

Launch the preceding code block and you will see the following output:

```
xlist: [1, 30, 40]
ylist: [2, 60, 80]
```

LAMBDA EXPRESSIONS

Listing 3.13 shows the content of `Lambda1.py`, which illustrates how to create a simple lambda function.

Listing 3.13: Lambda1.py

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


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

print(x1)
print(x2)
```

Listing 3.13 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 3.13 is as follows:

```
12
HelloPython
```

FUNCTIONAL PROGRAMMING IN PYTHON: THE MAP() FUNCTION

This section contains code samples that show you how to use the `map()` function, some of which involve lambda expressions that were discussed in the previous section. A subsequent section illustrates how to use the `filter()` function and also how to combine the `filter()` and `map()` functions in `Python`.

Listing 3.14 displays the contents of `map1.py` that illustrates how to use the `Python map()` function.

Listing 3.14: map1.py

```
def remainder(num):
  return num % 4

numbers = [-10, 11, -20, 55, 100, 201]
print("numbers:",numbers)
print()

iterator = map(remainder, numbers)
# option #1:
print("Mapped results: #1:")
print(list(iterator))
print()

# option #2:
iterator = map(remainder, numbers)
print("Mapped results: #2:")
for iter in iterator:
  print("value:",iter)
```

Listing 3.14 starts with the `Python` function `remainder()` that returns the integer remainder of dividing an integer by 4. The next code snippet initializes the variable `numbers` with a list of six numbers and displays their values.

Next, the variable `iterator` is initialized as the result of invoking the built-in `map()` function with parameters `remainder` and `numbers`. The result of doing so initializes `iterator` is a list of the integer-based remainder of division by 4 of all the values in `numbers`.

The next code block displays the contents of `iterator` as a list, followed by a loop that iterates through the values in `iterator` and prints them. The output from Listing 3.14 is as follows:

```
numbers: [-10, 11, -20, 55, 100, 201]

Mapped results: #1:
[2, 3, 0, 3, 0, 1]

Mapped results: #2:
value: 2
value: 3
value: 0
value: 3
value: 0
value: 1
```

Listing 3.15 displays the contents of `map2.py` that illustrates how to use the `Python map()` function.

Listing 3.15: map2.py

```
numbers = [-10, 11, -20, 55, 100, 201]
print("numbers:",numbers)
print()

iterator = map(lambda num: num % 4, numbers)
print("Mapped results:")
print(list(iterator))
print()
```

Listing 3.15 initializes the variable numbers as a list of six numbers and displays its contents. The next block is a variation of the code in Listing 3.14: The variable `iterator` is initialized as the result of invoking the built-in `map()` function with a lambda expression that divides its input by 4, followed by the variable `numbers`. The result of doing so initializes `iterator` is a list of the integer-based remainder of division by 4 of all the values in `numbers`. The output from Listing 3.15 is as follows:

```
numbers: [-10, 11, -20, 55, 100, 201]
Mapped results:
[2, 3, 0, 3, 0, 1]
```

Listing 3.16 displays the contents of `map3.py` that illustrates how to define a lambda

expression with the Python `map()` function to display the uppercase version of a list of strings.

Listing 3.16: map3.py

```
my_str = ["I", "love", "Chicago", "deep", "dish", "pizza"]
print("my_str:",my_str)
print()

iterator = map(lambda str: str.upper(), my_str)
print("Mapped results:")
print(list(iterator))
print()


iterator = map(lambda str: str.lower(), my_str)
print("Mapped results:")
print(list(iterator))
```

Listing 3.16 involves strings instead of integers, and initializes the variable `iterator` in a slightly different manner than what you saw in Listing 3.14 and Listing 3.15. Specifically, notice the snippets shown in bold: you must pass the function name *with* parentheses as the first parameter to map because they are the "target" of a lambda expression, which in turn requires a function that will be executed with the data. The output from Listing 3.16 is as follows:

```
my_str: ['I', 'love', 'Chicago', 'deep', 'dish', 'pizza']

Mapped results:
['I', 'LOVE', 'CHICAGO', 'DEEP', 'DISH', 'PIZZA']

Mapped results:
['i', 'love', 'chicago', 'deep', 'dish', 'pizza']
```

Listing 3.17 displays the contents of `map4.py` that illustrates how to invoke the Python `map()` function with the `upper()` function to display the lowercase version of a list of strings.

Listing 3.17: map4.py

```
my_str = ["I", "love", "Chicago", "deep", "dish", "pizza"]
print("my_str:",my_str)
print()

print("Mapped results:")
words_lower = list(map(str.lower, my_str))
print("lower: ",words_lower)
print()

print("Mapped results:")
words_lower2 = tuple(map(str.lower, my_str))
print("lower: ",words_lower2)
```

Listing 3.17 is a variant of Listing 3.16: The latter contains lambda functions in the definition of the variable operator, whereas the former specifies the map() function without lambda expressions. Again, notice the code snippets shown in bold: pass the function name *without* parentheses (so it's *not* a function invocation) as the first parameter to the `map()` function. The output from Listing 3.17 is as follows:

```
my_str: ['I', 'love', 'Chicago', 'deep', 'dish', 'pizza']

Mapped results:
lower:  ['i', 'love', 'chicago', 'deep', 'dish', 'pizza']

Mapped results:
lower:  ('i', 'love', 'chicago', 'deep', 'dish', 'pizza')
```

Listing 3.18 displays the contents of `map5.py` that illustrates how to calculate the mean of a 2D array of random values.

Listing 3.18: map5.py

```
import numpy as np

def find_mean(x):
  mean = sum(x)/len(x)
  return mean

# generate some random numbers:
rand_vals = np.random.randint(1,50, size=(3,4))
print("Random values:")
print(rand_vals)

mean_vals = list(map(find_mean, rand_vals))
print("Row-based mean values:")
print(mean_vals)
```

Listing 3.18 defines the function `find_mean()` that calculates the mean of a list of values, followed by the variable `rand_vals` that is initialized as a 3x4 array of random integers. The variable `mean_vals` is initialized as a list of values that is returned from invoking the `find_mean()` function with the contents of `rand_vals` (via the `map()` function). The output from Listing 3.18 is as follows:

```
Random values:
[[32  7 36  7]
[ 8  3 36 19]
[29 27 19 35]]
Row-based mean values:
[20.5, 16.5, 27.5]
```

If you want the column-based mean values, simply specify the array `rand_vals.T`, which is the transpose of the array `rand_vals`, as shown in the following code snippet:

```
mean_vals = list(map(find_mean, rand_vals.T))
```

Listing 3.19 displays the contents of `map6.py` that illustrates how to invoke the `Python map()` function with an array arrays of string values.

Listing 3.19: map6.py

```
import numpy as np

def single_line(data):
  return " ".join(data)

friends = [["Sara", "Smith", "San Francisco", "CA"],
           ["John", "Stone", "Chicago",       "IL"],
           ["Dave", "Aster", "Los Angeles",   "CA"],
           ["Jane", "Davis", "Seattle",       "WA"]]

print("=> Friends:")
for friend in friends:
  print(friend)
print()

print("=> Friend details:")
friend_details = list(map(single_line, friends))
for friend in friend_details:
  print(friend)
print()
```

Listing 3.19 follows a similar pattern that you have seen in the previous five code samples. The variation in this code sample involves the function `single_line()` that returns a single string, via the `join()` function, based on the strings in the variable `friends` that is initialized as an array of strings.

Next, the contents of `friends` are displayed via a `print()` statement, followed by the contents of the variable `friend_details` that is initialized with the result of invoking the built-in `map()` function with the parameters `single_line` and friends. The output from Listing 3.19 is as follows:

```
=> Friends:
['Sara', 'Smith', 'San Francisco', 'CA']
['John', 'Stone', 'Chicago', 'IL']
['Dave', 'Aster', 'Los Angeles', 'CA']
['Jane', 'Davis', 'Seattle', 'WA']

=> Friend details:
Sara Smith San Francisco CA
John Stone Chicago IL
Dave Aster Los Angeles CA
Jane Davis Seattle WA
```

Listing 3.20 displays the contents of `map_lambda_cond.py` that illustrates how to invoke the `Python map()` function and a lambda function with conditional logic.

Listing 3.20: map_lambda_cond.py

```
import numpy as np

arr = [13, 60, 0, 2, 17, 19]
print("arr:",arr)

mult_5 = list(map(lambda x: True if x % 5 == 0 else False, arr))
print("Multiples of 5:",mult_5)
```

Listing 3.20 is similar to Listing 3.15 in that both of them involve a lambda function: in this code sample the lambda function contains conditional logic of the form if/else. The output from Listing 3.20 is as follows:

```
arr: [13, 60, 0, 2, 17, 19]
Multiples of 5: [False, True, True, False, False, False]
```

Listing 3.21 displays the contents of simple_comprehension.py that illustrates how to achieve the same results as the Listing 3.20 without using a map() function or a lambda expression.

Listing 3.21: simple_comprehension.py

```
arr = [13, 60, 0, 300, 17, 19]
print("array:",arr)

mult5 = [True if x % 5 == 0 else False for x in arr]
print("Multiples of 5:",mult5)
```

Listing 3.21 initializes arr as a list of six numbers and then displays its contents. The next code snippet initializes the variable mult5 as the numbers in arr that are multiples of 5. The output from Listing 3.21 is as follows:

```
array: [13, 60, 0, 300, 17, 19]
Multiples of 5: [False, True, True, True, False, False]
```

FUNCTIONAL PROGRAMMING IN PYTHON: THE FILTER() FUNCTION

This section illustrates how to use the filter() function and also how to combine the filter() and map() functions in Python.

Listing 3.22 displays the contents of filter1.py that illustrates how to use the Python filter() function.

Listing 3.22: filter1.py

```
numbers = [-10, 11, -20, 55, 100, 201]
even_vals = list(filter(lambda x: x % 2 == 0, numbers))

print("numbers:",numbers)
print("even:   ",even_vals)
```

Listing 3.22 initializes the variable numbers with a list of integers and then initializes the variable even_vals as a list of values that is returned by the filter() function that uses a lambda expression to return only even integers from the integers in the variable numbers. The output from Listing 3.22 is as follows:

```
numbers: [-10, 11, -20, 55, 100, 201]
even:    [-10, -20, 100]
```

Combining the filter() and map() Functions

Listing 3.23 displays the contents of filter_map1.py that illustrates how to combine the Python filter() and map() functions.

Listing 3.23: filter_map1.py

```
square_even_nums = map(lambda num: num ** 2,
filter(lambda num: num % 2 == 0, range(1,10)))
print("square of even numbers:")
print(list(square_even_nums))
print()

even_squared_nums = filter(lambda num: num % 4 == 0,
map(lambda num: num ** 2, range(1,10)))
print("even squared numbers:")
print(list(even_squared_nums))
```

Listing 3.23 defines the variable square_even_nums that returns the square of the even numbers in the range of integers from 1 to 10. Next, the variable even_squarer_n ms is initialized as the list of numbers between 1 and 10 whose squared value is a multiple of 4. The output from Listing 3.23 is as follows:

```
square of even numbers:
[4, 16, 36, 64]

even squared numbers:
[4, 16, 36, 64]
```

Listing 3.24 displays the contents of filter_map2.py that illustrates how to combine the Python filter() and map() functions.

Listing 3.24: filter_map2.py

```
square_div4_nums = map(lambda num: num ** 2,
filter(lambda num: num % 4 == 0, range(1,10)))
print("square of multiples of 4:")
print(list(square_div4_nums))
print()
```

```
even_div4_nums = filter(lambda num: num % 4 == 0,
map(lambda num: num ** 2, range(1,10)))
print("multiple of 4 of squared numbers:")
print(list(even_div4_nums))
```

Listing 3.35 is similar to Listing 3.23, with the new criterion that specifies numbers that are multiples of 4 instead of multiples of 2. The output from Listing 3.35 is as follows:

```
square of multiples of 4:
[16, 64]

multiple of 4 of squared numbers:
[4, 16, 36, 64]
```

SUMMARY

This chapter started with an explanation of lists and operations such as splicing and updating lists. Then you learned how to work with vectors, matrices, and queues.

Next you learned about tuples, sets, and dictionaries. You also saw `Python` code samples that illustrate how to define lambda expressions and how to use the `map()` function and `filter()` function in `Python`.

CHAPTER 4

STRINGS AND ARRAYS

This chapter shows you how to use Python to perform various tasks involving strings and arrays. The code samples in this chapter consist of the following sequence: examples that involve scalars and strings, followed by examples involving vectors (explained further at the end of this introduction), and then examples involving 2D matrices.

The first part of this chapter contains a quick review of the time complexity of algorithms, followed by various Python code samples for solving well-known tasks, such as finding palindromes, reversing strings, and determining if the characters in a string are unique.

The second part of this chapter discusses 2D arrays, along with NumPy-based code samples that illustrate various operations that can be performed on 2D matrices. This section also discusses 2D matrices, which are 2D arrays, along with some tasks that you can perform on them. This section also discusses multidimensional arrays, which have properties that are analogous to lower-dimensional arrays.

The third part of this chapter introduces search algorithms, such as linear search and binary search, followed by some well-known sorting algorithms, such as the bubble sort, the merge sort, and the quick sort.

One other detail to keep in mind pertains to the terms *vectors* and *arrays*. In mathematics, a *vector* is a one-dimensional construct, whereas an *array* has at least two dimensions. In software development, an array can refer to a one-dimensional array or a higher-dimensional array (depending on the speaker). In this book a vector is always a one-dimensional construct. However, the term *array* always refers to a one-dimensional array; higher dimensional arrays will be referenced as "2D array," "3D array," and so forth. Therefore, the tasks involving 2D arrays start from the section titled "Working With 2D Arrays."

There are several points to keep in mind before you read this chapter. First, the Python code samples typically also contain simple NumPy functionality, such as NumPy arrays. If need be, you can find many free online tutorials that provide an introduction to NumPy.

Second, the Python code samples involving recursion assume that you have an understanding of recursion (possibly in a different programming language). However, if you are unfamiliar (or uncomfortable) with recursion, consider reading the first half of Chapter 6 that contains a gentler introduction to this topic, along with simpler code samples.

Third, the code samples in this chapter are intended to solve an interesting yet diverse set of tasks; however, feel free to focus on the code samples in this chapter that are of immediate interest, and you can always read the other code samples at some point in the future.

TIME AND SPACE COMPLEXITY

Algorithms are assessed in terms of the amount of space (based on input size) and the amount of time required for the algorithms to complete their execution, which is represented by "big O" notation. There are three types of time complexity: best case, average case, and worst case. Keep in mind that an algorithm with very good best case performance can have a relatively poor worst case performance.

Recall that O(n) means that an algorithm executes in linear time because its complexity is bounded above and below by a linear function. For example, if three algorithms require 2*n, 5*n, or n/2 operations, respectively, then all of them have O(n) complexity.

Moreover, if the best, average, and worst time performance for a linear search is 1, n/2, and n operations, respectively, which in turn equals O(1), O(n), and O(n), respectively.

The *time-space trade-off* refers to reducing either the amount of time or the amount of memory that is required for executing an algorithm, which involves choosing one of the following:

- execute in less time and more memory

- execute in more time and less memory

Although reducing both time and memory is desirable, it's also a more challenging task. For example, the calculation of Fibonacci numbers is much more efficient via an iterative algorithm than a recursive solution, but the former also requires an array to store intermediate values. The iterative solution has a higher memory requirement than a recursive solution.

Another point to keep in mind is the following inequalities (logarithms can be in any base that is greater than or equal to 2) for any positive integer `n` > 1:

```
O(log n) < O(n) < O(n*log n)
```

In addition, the following inequalities with powers of `n`, powers of 2, and factorial values are also valid:

```
O(n**2) < O(n**3) < O(2**n) < O(n!)
```

If you are unsure about any of the preceding inequalities, perform an online search for tutorials that provide the necessary details.

TASK: MAXIMUM AND MINIMUM POWERS OF AN INTEGER

Listing 4.1 displays the contents of `max_min_power_k2.py` that illustrate how to calculate the largest (smallest) power of a number whose base is `k` that is less than (greater than) a given number. So, if `num` and `k` are positive integers, the task is two-fold:

- find the *largest* number such that k**powk <= num
- find the *smallest* number such that k**powk >= num

For example, 16 is the *largest* power of two that is less than 24 and 32 is the *smallest* power of two that is greater than 24.

As another example, 625 is the largest power of five that is less than 1000 and 3125 is the smallest power of five that is greater than 1000.

Listing 4.1: max_min_power_k2.py

```
def max_min_powerk(num,k):
  powk = 1
  while(powk <= num):
    powk *= k
  if(powk > num):
    powk /= k
  return int(powk), int(powk*k)

nums = [24,17,1000]
powers = [2,3,4,5]

for num in nums:
  for k in powers:
    lowerk,upperk = max_min_powerk(num, k)
    print("num:",num,"power:",k,"lower",lowerk,"upper:",upperk)
  print()
```

Listing 4.1 starts with the function `max_min_powerk()` that contains a loop that repeatedly multiplies the local variable `powk` (initialized with the value 1) by `k`. When the value of `powk` is greater than the current value of `num`, `powk` is divided by `k` so that we have the lower bound solution.

Note that this function returns `powk` and `powk*k` that are the lower bound and higher bound solutions for this task. Launch the code in Listing 4.1 and you will see the following output:

```
num: 24 power: 2 lower 16 upper: 32
num: 24 power: 3 lower 9 upper: 27
num: 24 power: 4 lower 16 upper: 64
num: 24 power: 5 lower 5 upper: 25

num: 17 power: 2 lower 16 upper: 32
num: 17 power: 3 lower 9 upper: 27
num: 17 power: 4 lower 16 upper: 64
num: 17 power: 5 lower 5 upper: 25

num: 1000 power: 2 lower 512 upper: 1024
num: 1000 power: 3 lower 729 upper: 2187
num: 1000 power: 4 lower 256 upper: 1024
num: 1000 power: 5 lower 625 upper: 3125
```

TASK: BINARY SUBSTRINGS OF A NUMBER

Listing 4.2 displays the contents of the `binary_numbers.py` that illustrates how to display all binary substrings whose length is less than or equal to a given number.

Listing 4.2: binary_numbers.py

```
import numpy as np
def binary_values(width):
  print("=> binary values for width=",width,":")
  for i in range(0,2**width):
    bin_value = bin(i)
    str_value = str(bin_value)
    print(str_value[2:])
  print()

max_width = 4
for ndx in range(1,max_width):
  binary_values(ndx)
```

Listing 4.2 starts with the function `binary_values()` whose loop iterates from 0 to `2**width`, where width is the parameter for this function. The loop variable is `i` and during each iteration, `bin_value` is initialized with the binary value of `i`.

Next, the variable `str_value` is the string-based value of `bin_value`, which is stripped of the two leading characters `0b`. Launch the code in Listing 4.2 and you will see the following output:

```
=> binary values for width= 1 :
0
1

=> binary values for width= 2 :
0
1
10
11

=> binary values for width= 3 :
0
1
10
11
100
101
110
111
```

TASK: COMMON SUBSTRING OF TWO BINARY NUMBERS

Listing 4.3 displays the contents of `common_bits.py` that illustrates how to find the longest common substring of two binary strings.

Listing 4.3: common_bits.py

```
def common_bits(num1, num2):
    bin_num1 = bin(num1)
    bin_num2 = bin(num2)
    bin_num1 = bin_num1[2:]
    bin_num2 = bin_num2[2:]

    if(len(bin_num2) < len(bin_num1)):
      while(len(bin_num2) < len(bin_num1)):
        bin_num2 = "0" + bin_num2

    print(num1,"=",bin_num1)
    print(num2,"=",bin_num2)

    common_bits2 = 0
    for i in range(0,len(bin_num1)):
      if((bin_num1[i] == bin_num2[i]) and (bin_num1[i] =='1')):
        common_bits2 += 1
    return common_bits2

nums1 = [61,28, 7,100,189]
nums2 = [51,14,28,110, 14]

for idx in range(0,len(nums1)):
  num1 = nums1[idx]
  num2 = nums2[idx]
  common = common_bits(num1, num2)

  print(num1,"and",num2,"have",common,"bits in common")
  print()
```

Listing 4.3 starts with the function `common_bits()` that initializes the binary numbers `bin_num1` and `bin_num2` with the binary values of the two input parameters, after which the initial string "0b" is removed from both numbers.

Next, a loop iterates from 0 to the length of the string `bin_num1` in order to check each digit to see whether or not it equals 1. Each time that the digit 1 is found, the value of

common_bits2 (initialized with the value 0) is incremented. When the loop terminates, the variable common_bits2 equals the number of times that bin_num1 and bin_num2 have a 1 in the same position.

The final portion of Listing 4.3 iterates through a pair of arrays with positive integers values and invokes common_bits() during each iteration of the loop. Now launch the code in Listing 4.3 and you will see the following output:

```
61 = 111101
51 = 110011
61 and 51 have 3 bits in common

28 = 11100
14 = 01110
28 and 14 have 2 bits in common

7 = 111
28 = 11100
7 and 28 have 3 bits in common

100 = 1100100
110 = 1101110
100 and 110 have 3 bits in common

189 = 10111101
14 = 00001110
189 and 14 have 2 bits in common
```

TASK: MULTIPLY AND DIVIDE VIA RECURSION

Listing 4.4 displays the contents of the recursive_multiply.py that illustrates how to compute the product of two positive integers via recursion.

Listing 4.4: recursive_multiply.py

```
import numpy as np

def add_repeat(num, times, sum):
  if(times == 0):
    return sum
  else:
    return add_repeat(num, times-1, num+sum)

arr1 = np.array([5,13,25,17,100])
arr2 = np.array([9,10,25,10,100])

for i in range(0,len(arr1)):
  num1 = arr1[i]
  num2 = arr2[i]
  prod = add_repeat(num1, num2, 0)
  print("product of",num1,"and",num2,"=",prod)
```

Listing 4.4 starts with the function add_repeat(num,times,sum) that performs repeated addition by recursively invokes itself. Note that this function uses tail recursion: Each invocation of the function replaces times with times-1 and also replaces sum with num+sum (the latter is the tail recursion). The terminating condition is when times equals 0, at which point the function returns the value of sum. Launch the code in Listing 4.4 and you will see the following output:

```
product of 5 and 9 = 45
product of 13 and 10 = 130
product of 25 and 25 = 625
product of 17 and 10 = 170
product of 100 and 100 = 10000
```

Listing 4.5 displays the contents of the recursive_divide.py that illustrates how to compute the quotient of two positive integers via recursion.

Listing 4.5: recursive_divide.py

```
import numpy as np
def sub_repeat(num1, num2, remainder):
  if(num1 < num2):
    return num1
  else:
    #print("num1-num2:",num1-num2,"num2:",num2)
    return sub_repeat(num1-num2, num2, remainder)

arr1 = np.array([9,13,25,17,100])
arr2 = np.array([5,10,25,10,100])

for i in range(0,len(arr1)):
  num1 = arr1[i]
  num2 = arr2[i]
  prod = sub_repeat(num1, num2, 0)
```

```
  print("remainder of",num1,"/",num2,"=",prod)
```

Listing 4.5 contains code that is very similar to Listing 4.4: The difference involves replacing addition with subtraction. Launch the code in Listing 4.5 and you will see the following output:

```
remainder of 9 / 5 = 4
remainder of 13 / 10 = 3
remainder of 25 / 25 = 0
remainder of 17 / 10 = 7
remainder of 100 / 100 = 0
```

TASK: SUM OF PRIME AND COMPOSITE NUMBERS

Listing 4.6 displays the contents of the `pair_sum_sorted.py` that illustrates how to determine whether or not a sorted array contains the sum of two specified numbers.

Listing 4.6: pair_sum_sorted.py

```
import numpy as np

PRIME_NUM = 1
COMPOSITE = 0
prime_sum = 0
comp_sum  = 0
prime_list = np.array([])
comp_list  = np.array([])
arr1 = np.array([5,10,17,23,30,47,50])

def is_prime(num):
  div = 2

  while(div < num):
   if( num % div != 0):
      div += 1
   else:
      return COMPOSITE
  return PRIME_NUM

for ndx in range(0,len(arr1)):
  num = arr1[ndx]

  if(is_prime(num) == PRIME_NUM):
    prime_list = np.append(prime_list, num)
    prime_sum += num
  else:
    comp_list = np.append(comp_list, num)
    comp_sum += num

print("prime list:",prime_list)
print("comp  list:",comp_list)
print("prime sum: ",prime_sum)
print("comp sum:  ",comp_sum)
```

Listing 4.6 starts with the function `is_prime()` that determines whether or not its input parameter is a prime number. The next portion of code in Listing 4.6 is a loop that ranges from 0 to the number of elements. During each iteration, the current number is added to the variable `prime_sum` if that number is a prime; otherwise, it is added to the variable `comp_sum`.

The final portion of Listing 4.6 displays the sum of the even numbers and the sum of the odd numbers in the input array `arr1`. Launch the code in Listing 4.6 and you will see the following output:

```
prime list: [ 5. 17. 23. 47.]
comp  list: [10. 30. 50.]
prime sum:  92
comp sum:   90
```

The next portion of this chapter contains various examples of string-related tasks. As needed, you can review the relevant portion of Chapter 1 regarding some of the `Python` built-in string functions, such as `int()` and `len()`.

TASK: COUNT WORD FREQUENCIES

Listing 4.7 displays the contents of the `word_frequency.py` that illustrates how to determine the frequency of each word in an array of words.

Listing 4.7: word_frequency.py

```
import numpy as np

def word_count(words,check_word):
  count = 0
```

```
  for word in words:
    if(word.lower() == check_word.lower()):
      count += 1
  return count

sents = np.array([["I", "love", "thick", "pizza"],
                  ["I", "love", "deep", "dish","pizza"],
                  ["Pepperoni","and","sausage","pizza"],
                  ["Pizza", "with", "mozzarella"]],dtype=object)

words = np.array([])
for sent in sents:
  for word in sent:
    words = np.append(words,word)

word_counts = {}
for word in words:
  count = word_count(words,word)
  word_counts[word] = count

print("word_counts:")
print(word_counts)
```

Listing 4.7 starts with the function `word_count()` that counts the number of occurrences of a given word in a sentence. The next portion of Listing 4.7 contains a loop that iterates through each sentence of an array of sentences. For each sentence, the code invokes the function `word_count()` with each word in the current sentence. Launch the code in Listing 4.7 and you will see the following output:

```
word_counts:
{'I': 2, 'love': 2, 'thick': 1, 'pizza': 4, 'deep': 1, 'dish': 1, 'Pepperoni': 1, 'and': 1,
'sausage': 1, 'Pizza': 4, 'with': 1, 'mozzarella': 1}
```

Listing 4.8 displays the contents of the `word_frequency2.py` that illustrates another way to determine the frequency of each word in an array of words.

Listing 4.8: word_frequency2.py

```
import numpy as np

sents = np.array([["I", "love", "thick", "pizza"],
                  ["I", "love", "deep", "dish","pizza"],
                  ["Pepperoni","and","sausage","pizza"],
                  ["Pizza", "with", "mozzarella"]],dtype=object)

word_counts = dict()
for sent in sents:
  for word in sent:
    word = word.lower()
    #print("word:",word)

    if(word not in word_counts.keys()):
       word_counts[word] = 0
    word_counts[word] += 1

print("word_counts:")
print(word_counts)
```

Listing 4.8 concatenates all the sentences and then populates a `Python` dictionary with word frequencies whereas Listing 4.7 directly populates a `Python` dictionary with word frequencies. Launch the code in Listing 4.8 and you will see the following output:

```
word_counts:
{'i': 2, 'love': 2, 'thick': 1, 'pizza': 4, 'deep': 1, 'dish': 1, 'pepperoni': 1, 'and': 1,
'sausage': 1, 'with': 1, 'mozzarella': 1}
```

TASK: CHECK IF A STRING CONTAINS UNIQUE CHARACTERS

Listing 4.9 displays the contents of the `unique_str.py` that illustrates how to determine whether or not a string contains unique letters: note that the solution is for `ASCII`-based characters.

Listing 4.9: unique_chars.py

```
import numpy as np

def unique_chars(str):
  if (len(str) > 128):
    return false

  str = str.lower()

  char_set = np.zeros([128])
```

```
for i in range (0,len(str)):
    char = str[i]
    val = ord('z') - ord(char)
    #print("val:",val)

    if (char_set[val] == 1):
      # found duplicate character
      return False
    else:
      char_set[val] = 1

  return True

arr1 = np.array(["a string", "second string", "hello world"])

for str in arr1:
  print("string:",str)
  result = unique_chars(str)
  print("unique:",result)
  print()
```

Listing 4.9 starts with the function `unique_chars()` that converts its parameter `str` to lowercase letters and then initializes the 1×128 integer array `char_set` whose values are all 0. The next portion of this function iterates through the characters of the string `str` and initializes the integer variable `val` with the offset position of each character from the character 'z'.

If this position in `char_set` equals 1, then a duplicate character has been found; otherwise, this position is initialized with the value 1. Note that the value `False` is returned if the string `str` contains duplicate letters, whereas the value `True` is returned if the string str contains unique characters. Now launch the code in Listing 4.9 and you will see the following output:

```
string: a string
unique: True

string: second string
unique: False

string: hello world
unique: False
```

TASK: INSERT CHARACTERS IN A STRING

Listing 4.10 displays the contents of the `insert_chars.py` that illustrates how to insert each character of one string in every position of another string.

Listing 4.10: insert_chars.py

```
def insert_char(str1, chr):
  result = str1

  result = chr + str1
  for i in range(0,len(str1)):
    left = str1[:i+1]
    right = str1[i+1:]
    #print("left:",left,"right:",right)
    inserted = left + chr + right

    result = result + " " + inserted
  return result

str1 = "abc"
str2 = "def"
print("str1:",str1)
print("str2:",str2)

insertions = ""
for i in range(0,len(str2)):
  new_str = insert_char(str1, str2[i])
  #print("new_str:",new_str)
  insertions = insertions+ " " + new_str

print("result:",insertions)
```

Listing 4.10 starts with the function `insert_char()` that has a string `str1` and a character `chr` as input parameters. The next portion of code is a loop whose loop variable is `i` and is used to split the string `str1` into two strings: the left substring from positions 0 to `i`, and the right substring from position `i+1`. A new string with three components is constructed: the left string, the character `chr`, and the right string.

The next portion of Listing 4.10 contains a loop that iterates through each character of `str2`; during each iteration, the code invokes `insert_char()` with string `str1` and the current character. The number of new strings generated bytes this code equals the following

product: `(len(str1)+1)*len(str2)`.

Launch the code in Listing 4.10 and you will see the following output:

```
str1: abc
str2: def
result:  dabc adbc abdc abcd eabc aebc abec abce fabc afbc abfc abcf
```

TASK: STRING PERMUTATIONS

There are several ways to determine whether or not two strings are permutations of each other. One way involves sorting the strings alphabetically: If the resulting strings are equal, then they are permutations of each other.

A second technique is to determine whether or not they have the same number of occurrences for each character. A third way is to add the numeric counterpart of each letter in the string: If the numbers are equal and the strings have the same length, then they are permutations of each other.

Listing 4.11 displays the contents of the `string_permute.py` that illustrates how to determine whether or not two strings are permutations of each other.

Listing 4.11: string_permute.py

```
import numpy as np

def permute(str1,str2):
  str1d = sorted(str1)
  str2d = sorted(str2)
  permute = (str1d == str2d)

  print("string1: ",str1)
  print("string2: ",str2)
  print("permuted:",permute)
  print()

strings1 = ["abcdef", "abcdef"]
strings2 = ["efabcf", "defabc"]

for idx in range(0,len(strings1)):
  str1 = strings1[idx]
  str2 = strings2[idx]
  permute(str1,str2)
```

Listing 4.11 starts with the function `permute()` that takes the two strings `str1` and `str2` as parameters. Next, the strings `str1d` and `str2d` are initialized with the result of sorting the characters in the strings `str1` and `str2`, respectively. At this point, we can determine whether or not `str1` and `str2` are permutations of each other by checking whether or not the strings `str1d` and `str2d` are equal. Launch the code in Listing 4.11 and you will see the following output:

```
string1:  abcdef
string2:  efabcf
permuted: False

string1:  abcdef
string2:  defabc
permuted: True
```

TASK: FIND ALL SUBSETS OF A SET

Listing 4.12 displays the contents of the `powerset.py` that illustrates how to list all the subsets of a given set.

Listing 4.12: powerset.py

```
import numpy as np

# strings of the form:
# [a0, a1, ..., an]
def create_array(width):
  arr1 = np.array([])
  for i in range(0,width):
    str1 = "a"+str(i)
    arr1 = np.append(arr1,str1)
  return arr1

def binary_values(arr1,width):
  print("=> binary values for width=",width,":")
  for num in range(0,2**width):
    bin_value = bin(num)
    str_value = bin_value[2:]

    # left-pad with "0" characters:
```

```
    for i in range(0,width-len(str_value)):
      str_value = "0" + str_value

    subset = ""
    # check for '1' in a right-to-left loop:
    for ndx in range(len(str_value)-1,-1,-1):
      chr = str_value[ndx]
      if(chr == '1'):
        subset = subset + " " +arr1[ndx]

    if(subset == ""):
      print("{}")
    else:
      if(subset[0] == " "):
        subset = subset[1:]
      print(subset)

width = 4
arr1 = create_array(width)
print("arr1:",arr1)
binary_values(arr1,width)
```

Listing 4.12 starts with the function `create_array` that creates (and returns) the array `arr1` of strings of the form `a1, a2, . . . , an`, where n is the value of the input parameter. The next portion of Listing 4.12 defines the function `binary_values` that constructs all the subsets of `arr1` that is constructed in the function `create_array()`.

The function `binary_values` contains a loop whose loop variable `num` iterates through the numbers 1 through `2**width`, where `width` is the value of the input parameter. For each value of `num`, the variable `bin_value` is its binary equivalent, and the variable `str_value` skips the first two character positions (i.e., "0b") of `bin_value`.

The next portion of Listing 4.12 contains a loop that pads `str_value` with the string "0" until its length equals `width`, followed by another loop that creates a string called `subset` that consists of the `ith` character of `arr1` and whose length equals the number of occurrences of "1" in the variable `str_value`. If this logic is not clear to you, perform a "desk check" that iterates through the code in `create_array()`.

The final portion of Listing 4.12 invokes the function `create_array()` with the value 4 for the parameter `width`, after which all the subsets of a set of 4 elements is displayed. Launch the code in Listing 4.12 and you will see the following output:

```
arr1: ['a0' 'a1' 'a2' 'a3']
=> binary values for width= 4 :
{}
a3
a2
a3 a2
a1
a3 a1
a2 a1
a3 a2 a1
a0
a3 a0
a2 a0
a3 a2 a0
a1 a0
a3 a1 a0
a2 a1 a0
a3 a2 a1 a0
```

TASK: CHECK FOR PALINDROMES

One way to determine whether or not a given string is a palindrome is to compare the string with the reverse of the string: If the two strings are equal, then the string is a palindrome. Moreover, there are two ways to reverse a string: One way involves the `Python reverse()` function, and another way is to process the characters in the given string in a right-to-left fashion, and to append each character to a new string.

Another technique involves iterating through the characters in a left-to-right fashion and comparing each character with its corresponding character that is based on iterating through the string in a right-to-left fashion.

Listing 4.13 displays the contents of the `palindrome1.py` that illustrates how to determine whether or not a string or a positive integer is a palindrome.

Listing 4.13: palindrome1.py

```
import numpy as np

def palindrome1(str):
  full_len = int(len(str))
  half_len = int(len(str)/2)

  for i in range (0,half_len):
    lchar = str[i]
```

```
    rchar = str[full_len-1-i]
    if(lchar != rchar):
      return False
  return True

arr1 = np.array(["rotor", "tomato", "radar","maam"])
arr2 = list([123, 12321, 555])

# CHECK FOR STRING PALINDROMES:
for str in arr1:
  print("check string:",str)
  result = palindrome1(str)
  print("palindrome:  ",result)
  print()

# CHECK FOR NUMERIC PALINDROMES:
for num in arr2:
  print("check number:",num)
  str1 = np.str(num)
  str2 = ""
  for digit in str1:
    str2 += digit

  result = palindrome1(str2)
  print("palindrome:  ",result)
  print()
```

Listing 4.13 defines the function `palindrome1()` that contains a loop that compares the elements from opposite directions of the string `str`. Specifically, the character in position 0 is compared with the right-most character of `str`: if they are different, then str is not a palindrome, and the value `False` is returned.

However, if the two characters are the same, then the process is repeated, this time with the character in position 1 and the character that is to the left of the right-most character of `str`. Repeat this process using the same conditional logic: If we reach the end of this loop, then `str` is a palindrome and the value `True` is returned.

The next portion of Listing 4.13 initializes the variables `arr1` and `arr2` with strings and numbers, respectively, followed by two loops. The first loop invokes the function `palindrome1()` with each element in `arr1`, and the second loop initializes the string-based counterpart to each number in `arr2`, and then invokes the function `palindrome1()` with that string. Launch the code in Listing 4.13 and you will see the following output:

```
check string: rotor
palindrome:   True

check string: tomato
palindrome:   False

check string: radar
palindrome:   True

check string: maam
palindrome:   True

check number: 123
palindrome:   False

check number: 12321
palindrome:   True

check number: 555
palindrome:   True
```

TASK: CHECK FOR LONGEST PALINDROME

This section extends the code in the previous section by examining substrings of a given string. Listing 4.14 displays the contents of the `longest_palindrome.py` that illustrates how to determine the longest palindrome in a given string. Note that a single character is always a palindrome, which means that every string has a substring that is a palindrome (in fact, any single character in any string is a palindrome).

Listing 4.14: longest_palindrome.py

```
import numpy as np

def check_string(str):
  result = 0
  str_len  = len(str)
  str_len2 = int(len(str)/2)

  for i in range(0,str_len2):
    if(str[i] != str[str_len-i-1]):
```

```
        result = 1
        break

    if(result == 0):
      #print(str, "is a palindrome")
      return str
    else:
      #print(str, "is not a palindrome")
      return None

my_strings = ["abc","abb","abccba","azaaza","abcdefgabccbax"]
max_pal_str = ""
max_pal_len = 0

for my_str in my_strings:
  max_pal_str = ""
  max_pal_len = 0
  for i in range(0,len(my_str)-1):
    for j in range(0,len(my_str)-i+1):
      sub_str = my_str[i:i+j]
      #print("checking:",sub_str,"in =>",my_str)
      a_str = check_string(sub_str)

      if(a_str != None):
        if(max_pal_len < len(a_str)):
          max_pal_len = len(a_str)
          max_pal_str = a_str

  print("string:",my_str)
  print("maxpal:",max_pal_str)
  print()
```

Listing 4.14 starts with the function `check_string()` that performs the same functionality as the function `palindrome1()` in Listing 4.13.

The next portion of Listing 4.14 initializes the variable `my_strings` as a list of strings, each of which will be processed to determine which strings (if any) are palindromes. Next, a loop iterates through each element of `my_strings`, followed by a nested loop via the loop variables `i` and `j`. The loop variable `i` iterates from left to right, starting from the left-most character in the string `my_str`. However, the loop variable `j` iterates from right to left, starting from the right-most character in `my_str`.

During each iteration through the nested loop, the substring `sub_str` is constructed, which consists of the characters in position `i` through `i+j` of `my_str`, after which the function `check_string()` is invoked to determine whether or not `sub_str` is a palindrome.

If the result is true, then the length of `a_str` (which is returned from `check_string()`) is compared with the length of the longest palindrome that has been found thus far. If necessary, update the values of the variables `max_pal_len` and `max_pal_str` whose values are displayed when the nested loop has completed execution. Launch the code in Listing 4.14 and you will see the following output:

```
string: abc
maxpal: a

string: abb
maxpal: bb

string: abccba
maxpal: abccba

string: azaaza
maxpal: azaaza

string: abcdefgabccbax
maxpal: abccba
```

WORKING WITH SEQUENCES OF STRINGS

This section contains `Python` code samples that search strings to determine the following:

- the maximum length of a sequence of consecutive 1s in a string
- find a given sequence of characters in a string
- the maximum length of a sequence of unique characters

After you complete this section you can explore variations of these tasks that you can solve using the code samples in this section.

The Maximum Length of a Repeated Character in a String

Listing 4.15 displays the contents of `max_char_sequence.py` that illustrates how to rotate the elements in an array.

Listing 4.15: max_char_sequence.py

```
import numpy as np

def max_seq(my_str,char):
  max = 0
  left = 0
  right = 0
  counter = 0
  for i in range(0,len(my_str)):
    curr_char = my_str[i]
    if(curr_char == char):
      counter += 1
      right = i
      if( max < counter):
        max = counter
        #print("new max:",max)
    else:
      counter = 0
      left = i
      right = i

  print("my_str:",my_str)
  print("max sequence of",char,":",max)
  print()

str_list = np.array(["abcdef","aaaxyz","abcdeeefghij"])
char_list = np.array(["a","a","e"])

for idx in range(0,len(str_list)):
  my_str = str_list[idx]
  char = char_list[idx]
  max_seq(my_str,char)
```

Listing 4.15 defines the function `max_seq()` that initializes several scalar variables to keep track of the `left` index and `right` index positions of the parameter `my_str`. The next block of code is a loop that iterates from 0 to the length of `my_str`, and updates `counter` to keep track of the number of consecutive occurrences of the current character, where the latter starts from index 0.

When a different character is encountered, the values of `left` and `right` are updated so that they start from the index position of the unequal character that was encountered. In addition, the value of `counter` is compared with `max` in order to ensure that the latter is the length of the longest sequence of equal characters. Launch the code in Listing 4.15 and you will see the following output:

```
my_str: abcdef
max sequence of a : 1

my_str: aaaxyz
max sequence of a : 3

my_str: abcdeeefghij
max sequence of e : 3
```

Find a Given Sequence of Characters in a String

Listing 4.16 displays the contents of `max_substr_sequence.py` that illustrates how to find the longest right-most sequence of characters that matches a given string.

Listing 4.16: max_substr_sequence.py

```
import numpy as np

def rightmost_substr(my_str,substr):
  left = -1
  len_substr = len(substr)

  # check for substr from right to left:
  for i in range(len(my_str)-len_substr,-1,-1):
    curr_str = my_str[i:i+len_substr]
    #print("checking curr_str:",curr_str)

    if(substr == curr_str):
      left = i
      break

  if(left >= 0):
    print(substr,"is in index",left,"of:",my_str)
  else:
    print(substr,"does not appear in",my_str)
  print()

str_list = np.array(["abcdef","aaaxyz","abcdeeefghij"])
```

```
substr_list = np.array(["bcd","aaa","cde"])

for idx in range(0,len(str_list)):
  my_str = str_list[idx]
  substr = substr_list[idx]
  print("checking:",substr,"in:",my_str)
  rightmost_substr(my_str,substr)
```

Listing 4.16 defines the function `rightmost_substr()` that contains a loop that performs a right-to-left comparison of the parameter `substr` with the parameter `my_str`. If a match occurs, then the index `i` of `substr` in which the match occurred is printed, otherwise a message to the contrary is printed.

The next portion of Listing 4.16 initializes the `NumPy` array `str_list` with multiple strings, along with the variable `substr_list` as a `NumPy` array of strings. The final portion of Listing 4.16 contains a loop that iterates through the elements of `str_list`, along with the corresponding elements of `substr_list`, and then invokes the function `rightmost_substr()` with this pair of strings. Launch the code in Listing 4.16 and you will see the following output:

```
checking: bcd in: abcdef
bcd is in index 1 of: abcdef

checking: aaa in: aaaxyz
aaa is in index 0 of: aaaxyz

checking: cde in: abcdeeefghij
cde is in index 2 of: abcdeeefghij
```

TASK: LONGEST SEQUENCES OF SUBSTRINGS

This section contains `Python` code samples that search strings to determine the following:

- the maximum length of a sequence of consecutive 1s in a string
- find a given sequence of characters in a string
- the maximum length of a sequence of unique characters

After you complete this section, you can explore variations of these tasks that you can solve using the code samples in this section.

The Longest Sequence of Unique Characters

Listing 4.17 displays the contents of `longest_unique.py` that illustrates how to find the longest sequence of unique characters in a string.

Listing 4.17: longest_unique.py

```
import numpy as np

def rightmost_substr(my_str):
  left = 0
  right = 0
  sub_str = ""
  longest = ""
  my_dict = dict()

  for pos in range(0,len(my_str)):
    char = my_str[pos]
    if(char not in my_dict.keys()):
      my_dict[char] = 1
      unique = my_str[left:pos+1]
      #print("new unique:", unique)

      if(len(longest) < len(unique)):
        longest = unique
        right = pos
    else:
      my_dict = dict()
      left = pos+1
      right = pos+1

  print("longest unique:",longest)
  print()

str_list = np.array(["abcdef","aaaxyz","abcdeeefghij"])

for idx in range(0,len(str_list)):
  my_str = str_list[idx]
  print("checking:",my_str)
  rightmost_substr(my_str)
```

Listing 4.17 defines the function `rightmost_substr()` that contains a loop that stores each

character of my_str in a Python dictionary my_dict. If a character is encountered already exists in my_dict, the length of the longest current sequence is compared with the length of the string unique (which is always the longest unique sequence found thus far).

If the current sequence is longer, then longest is updated; otherwise, a new instance of the Python dictionary my_dict is created and the process begins anew until the end of the string my_str is reached. Launch the code in Listing 4.17 and you will see the following output:

```
checking: abcdef
longest unique: abcdef

checking: aaaxyz
longest unique: axyz

checking: abcdeeefghij
longest unique: efghij
```

The Longest Repeated Substring

Listing 4.18 displays the contents of max_repeated_substr.py that illustrates how to find the longest sequence of unique characters in a string.

Listing 4.18: max_repeated_substr.py

```
def check_string(my_str, sub_str, pos):
  str_len = len(my_str)
  sub_len = len(sub_str)
  #print("my_str:",my_str,"sub_str:",sub_str)
  match = None
  part_str = ""
  left = 0
  right = 0

  for i in range(0,str_len-sub_len-pos):
    left  = pos+sub_len+i
    right = left+sub_len
    #print("left:",left,"right:",right)
    part_str = my_str[left:right]

    if(part_str == sub_str):
      match = part_str
      break

  return match,left

print("==> Check for repeating substrings of length at least 2")
my_strings = ["abc","abb","abccba","azaaza","abcdefgabccbaxyz"]

for my_str in my_strings:
  half_len = int(len(my_str)/2)
  max_len = 0
  max_str = ""
  for i in range(0,half_len+1):
    for j in range(2,half_len+1):
      sub_str = my_str[i:i+j]
      a_str,left = check_string(my_str, sub_str,i)

      if(a_str != None):
        print(a_str,"appears in pos",i,"and pos",left,"in =>",my_str)
        if(max_len < len(a_str)):
          max_len = len(a_str)
          max_str = a_str

  if(max_str != ""):
    print("=> Maximum repeating substring:",max_str)
  else:
    print("No maximum repeating substring:",my_str)
  print()
```

Listing 4.18 defines the function check_string() that checks whether or not the parameter substr appears in the string mystr, that is offset by the value of the parameter pos. If a match is successful, the function check_string() returns the matching string and the position where the match occurred.

The next portion of Listing 4.18 initializes the variable my_strings as a list of strings, followed by a loop that iterates through the elements of my_strings. During each iteration of the loop, a nested loop is executed that involves loop variables i for the outer loop and j for the inner loop.

The outer (nested) loop iterates from index 0 to the midpoint of the current element of my_strings, whereas the inner (nested) loop iterates from index 2 to the midpoint of the current element of my_strings. During each iteration of the nested loop, the string sub_str is initialized with the sequence of characters from index position i through index position i+j.

Next, the function check_string() is invoked with my_str, sub_str, and index ii, and the result

is used to determine whether or not to update the values of `max_len` and `max_str`. Launch the code in Listing 4.18 and you will see the following output:

```
==> Check for repeating substrings of length at least 2
No maximum repeating substring: abc

No maximum repeating substring: abb

No maximum repeating substring: abccba

az appears in pos 0 and pos 3 in => azaaza
aza appears in pos 0 and pos 3 in => azaaza
za appears in pos 1 and pos 4 in => azaaza
=> Maximum repeating substring: aza

ab appears in pos 0 and pos 7 in => abcdefgabccbaxyz
abc appears in pos 0 and pos 7 in => abcdefgabccbaxyz
bc appears in pos 1 and pos 8 in => abcdefgabccbaxyz
=> Maximum repeating substring: abc
```

This concludes the portion of the chapter regarding strings-related code samples. The next section introduces you to arrays and various code samples.

WORKING WITH 1D ARRAYS

A *one-dimensional array* in `Python` is a one-dimensional construct whose elements are homogeneous (i.e., mixed data types are not permitted). Given two arrays A and B, you can add or subtract them, provided that they have the same number of elements. You can also compute the inner product of two vectors by calculating the sum of their component-wise products.

Now that you understand some of the rudimentary operations with one-dimensional matrices, the following subsections illustrate how to perform various tasks on matrices in `Python`.

Rotate an Array

Listing 4.19 displays the contents of the `Python` script `rotate_list.py` that illustrates how to rotate the elements in a list.

Listing 4.19: rotate_list.py

```
import numpy as np
list = [5,10,17,23,30,47,50]
print("original:",list)

shift_count = 2
for ndx in range(0,shift_count):
  item = list.pop(0)
  arr1 = list.append(item)

print("rotated: ",list)
```

Listing 4.19 initializes the variable `list` with a list of integers and prints its contents. The next portion of Listing 4.19 contains a loop that iterates from 0 to `shift_count` (initialized with the value 2) that is the number of times to shift the contents of the variable `list`. The shift is performed by invoking the `pop()` method to remove the left-most element of `list` and then immediately appending the "popped" element to the variable `list`. Launch the code in Listing 4.19 and you will see the following output:

```
original: [5, 10, 17, 23, 30, 47, 50]
rotated:  [17, 23, 30, 47, 50, 5, 10]
```

TASK: INVERT ADJACENT ARRAY ELEMENTS

Listing 4.20 displays the contents of the `Python` script `invert_items.py` that illustrates how to invert ("swap") adjacent pairs of elements in an array.

Listing 4.20: invert_items.py

```
import numpy as np
arr1 = np.array([5,10,17,23,30,47,50])
print("original:",arr1)

mid_point = int(len(arr1)/2)

for ndx in range(0,mid_point+2,2):
  temp = arr1[ndx]
  arr1[ndx] = arr1[ndx+1]
```

```
    arr1[ndx+1] = temp

print("inverted:",arr1)
```

Listing 4.20 initializes the Numpy array arr1 and displays its contents. The next portion of Listing 4.20 contains a loop that uses the variable ndx to iterate from 0 to the midpoint of the array arr1.

During each iteration, the variable temp is initialized as the contents of the current element of arr1 in order to switch the values of the elements in arr1 in index positions ndx and ndx+1. Notice that the loop variable ndx is incremented by *two* instead of *one* because we are processing pairs of adjacent elements in arr1. Launch the code in Listing 4.20 and you will see the following output:

```
original: [ 5 10 17 23 30 47 50]
inverted: [10  5 23 17 47 30 50]
```

Listing 4.21 displays the contents of the Python script swap.py that illustrates how to invert adjacent values in an array *without* using an intermediate temporary variable.

Listing 4.21: swap.py

```
import numpy as np
def swap(num1,num2):
  delta = num2 - num1
  #print("num1:",num1,"num2:",num2)

  num2 = delta
  num1 = num1+delta
  num2 = num1-delta
  #print("num1:",num1,"num2:",num2)
  return num1,num2

arr1 = np.array([15,4,23,35,80,50])
print("BEFORE arr1:",arr1)

for idx in range(0,len(arr1),2):
  num1, num2 = swap(arr1[idx],arr1[idx+1])
  arr1[idx]   = num1
  arr1[idx+1] = num2
  #print("arr1:",arr1)

print("AFTER  arr1:",arr1)
```

Listing 4.21 defines the function swap() that swaps the values of two integer variables *without* using a loop. Perform a manual check with a pair of integers to confirm that swap() does indeed swap the values of its two parameters.

The next portion of Listing 4.21 initializes the Numpy array arr1, followed by a loop that iterates through adjacent pairs of elements of arr1, During each iteration, the function swap() is invoked and the returned values are used to swap adjacent value sin the array arr1. Launch the code in Listing 4.21 and you will see the following output:

```
BEFORE arr1: [15  4 23 35 80 50]
AFTER  arr1: [ 4 15 35 23 50 80]
```

WORKING WITH 2D ARRAYS

A *two-dimensional array* in Python is a two-dimensional construct whose elements are homogeneous (i.e., mixed data types are not permitted). Given two arrays A and B, you can add or subtract them, provided that they have the same number of rows and columns.

Multiplication of matrices works differently from addition or subtraction: if A is an mxn matrix that you want to multiply (on the right of A) by B, then B must be an nxp matrix. The rule for matrix multiplication is as follows: the number of columns of A must equal the number of rows of B.

In addition, the *transpose* of matrix A is another matrix At such that the rows and columns are interchanged. Thus, if A is an mxn matrix then At is an nxm matrix. The matrix A is *symmetric* if A = At. The matrix A is the *identity matrix* I if the values in the main diagonal (upper left to lower right) are 1 and the other values are 0.

The matrix A is *invertible* if there is a matrix B such that A*B = B*A = I. Based on the earlier discussion regarding the product of two matrices, both A and B must be square matrices with the same number of rows and columns.

Now that you understand some of the rudimentary operations with matrices, the following subsections illustrate how to perform various tasks on matrices in Python.

THE TRANSPOSE OF A MATRIX

As a reminder, the transpose of matrix A is matrix At, where the rows and columns of A are

the columns and rows, respectively, of matrix At.

Listing 4.22 displays the contents of the mat_transpose.py that illustrates how to find the transpose of an mxn matrix.

Listing 4.22: mat_transpose.py

```
import numpy as np

# the transpose of a matrix is a 90 degree rotation
def transpose(A,rows,cols):
   for i in range(0,rows):
     for j in range(i,cols):
       #print("switching",A[i,j],"and",A[j,i])
       temp = A[i,j]
       A[i,j] = A[j,i]
       A[j,i] = temp
   return A

A = np.array([[100,3],[500,7]])
print("=> original:")
print(A)
At = transpose(A, 2, 2)
print("=> transpose:")
print(At)
print()

# example 2:
A = np.array([[100,3,-1],[30,500,7],[123,456,789]])
print("=> original:")
print(A)
At = transpose(A, 3, 3)
print("=> transpose:")
print(At)
```

Listing 4.22 defines the function transpose() that takes three parameters: an array A, the number of rows of A, and the number of number of columns of A. This function contains a nested loop that swaps the rows and columns of A, using a temporary variable temp whenever a swap is performed, in order to find (and return) the transpose of A.

The next portion of Listing 4.22 contains two examples of an array whose transpose is calculated via the function transpose(). Launch the code in Listing 4.22 and you will see the following output:

```
=> original:
[[100   3]
[500   7]]
=> transpose:
[[100 500]
[  3   7]]

=> original:
[[100   3  -1]
[ 30 500   7]
[123 456 789]]
=> transpose:
[[100  30 123]
[  3 500 456]
[ -1   7 789]]
```

In case you didn't notice, the transpose At of a matrix A is actually a 90 degree rotation of matrix A. Hence, if A is a square matrix of pixels values for a PNG, then At is a 90 degree rotation of the PNG. However, if you take the transpose of At, the result is the original matrix A.

SEARCH ALGORITHMS

A linear search algorithm checks each element in an array in a linear fashion, which means the the search starts from the first element in the array and proceeds through the remaining elements until either 1) the element is found or 2) the end of the array is reached. Here is an example of a linear search in an array of numbers.

Linear Search

Listing 4.23 displays the contents of the linear_search.py that illustrates how to perform a linear search with an array of numbers.

Listing 4.23: linear_search.py

```
import numpy as np

found = -1
item = 123
arr1 = np.array([1,3,5,123,400])
```

```
for i in range(0,len(arr1)):
  if (item == arr1[i]):
     found = i
     break

if (found >= 0):
  print("found",item,"in position",found)
else:
  print(item,"not found")
```

Listing 4.23 is straightforward: A loop iterates through the elements of the `Numpy` array `arr1` and if it contains any element that equals the value of `item`, the code sets the value of `found` equal to `i` (the loop variable) and then exits the loop. A message is printed based on whether or not the value of `found` is greater than 0. Launch the code in Listing 4.23 and you will see the following output:

```
found 123 in position 3
```

Binary Search Walkthrough

A binary search requires a sorted array and also involves a recursion-based algorithm. The intuition is as follows:

- Step 1: Given a number and the middle value of a non-empty array, one of the following can happen:
  - Case 1—The number equals the middle value of the sorted array: we are finished.
  - Case 2—The number is greater than the middle value of the sorted array: Go to Step 1 with the left half of the array.
  - Case 3—The number is less than the middle value of the sorted array: Go to Step 1 with the right half of the array.

If the array in question is empty, then the item does not appear in the array.

Here are some examples that illustrate each of the possibilities described in the preceding list.

Example #1 (Case 1):
Item = 25

```
arr1 = np.array([10, 20, 25, 40, 100]);
```

We found the item in the middle of the array

Example #2 (Case 2):
Item = 25

```
arr1 = np.array([1,5,10, 15, 20, 25, 40]);
```

First iteration: 25 and [10,20,25]
Second iteration: 25 and [25]
Third iteration: we found item 25

Example #3 (Case 3):
Item = 25

```
arr1 = np.array([10, 20, 25, 40, 100,150,400]);
```

First iteration: 25 and [10,20,25,40]
Second iteration: 25 and [25,40]
Third iteration: 25 and [25]
Fourth iteration: we found item 25

Example #4 (no match):
Item = 25

```
arr1 = np.array([1,5,10, 15, 20, 30, 40]);
```

First iteration: 25 and [20,30,40]
Second iteration: 25 and [20]
Third iteration: 25 and []
Item not found

If necessary, you can create your own test examples and then perform a manual iteration through each test case.

Binary Search (Iterative Solution)

Listing 4.24 displays the contents of the `binary_search.py` that illustrates how to perform a binary search with an array of numbers.

Listing 4.24: binary_search.py

```
import numpy as np

arr1 = np.array([1,3,5,123,400])
left = 0
right = len(arr1)-1
found = -1
item = 123

while(left <= right):
  mid = int(left + (right-left)/2)

  if(arr1[mid] == item):
    found = mid
    break
  elif (arr1[mid] < item):
    left = mid+1
  else:
    right = mid-1

print("array:",arr1)

if( found >= 0):
  print("found",item,"in position",found)
else:
print(item,"not found")
```

Listing 4.24 initializes the `Numpy` array `arr1` with five integer values, along with several scalar variables, including `left` and `right` that keep track of the left index and right index, respectively, of `arr1`.

The next portion of Listing 4.24 is a loop that executes as long as the value of `left` (initially 0) is less than or equal to the value of `right` (whose initial value is `len(arr1)-1`). During each iteration, the variable mid is set equal to the integer-based average of the sum of `left` and `right`.

If the element in `arr1` at position `mid` equals `item`, the variable `found` is initialized to the value of `mid` and the loop exits. Although, if the value of `arr1[mid]` is *less* than item, then the value of left is set equal to `mid+1`, which means the next iteration through the loop will check the right half of the elements between `arr1[left]` and `arr1[right]`.

However, if the value of `arr1[mid]` is *greater* than `item`, then the value of `right` is set equal to `mid-1`, which means the next iteration through the loop will check the *left* half of the elements between `arr1[left]` and `arr1[right]`. Launch the code in Listing 4.24 and you will see the following output:

```
array: [  1   3   5 123 400]
found 123 in position 3
```

Binary Search (Recursive Solution)

Listing 4.25 displays the contents of the `binary_search_recursive.py` that illustrates how to perform a binary search recursively with an array of numbers.

Listing 4.25: binary_search_recursive.py

```
import numpy as np

def binary_search(data, item, left, right):
  if left > right:
    return False
  else:
    # incorrect (can result in overflow):
    # mid = (left + right) / 2
    mid = int(left + (right-left)/2)

    if item == data[mid]:
      return True
    elif item < data[mid]:
      # recursively search the left half
      return binary_search(data, item, left, mid-1)
    else:
      # recursively search the right half
      return binary_search(data, item, mid+1, right)

arr1  = np.array([1,3,5,123,400])
item  = 123
left  = 0
right = len(arr1)-1
result = binary_search(arr1, item, left, right)

print("array: ",arr1)
print("item:  ",item)
print("found",item,":",result)
```

Listing 4.25 defines the function `binary_search()` that is invoked recursively in order to determine whether or not a given number is an element of an array. Notice that this function takes four parameters: the array `data` of numbers, the `item` to check, the `left` position, and the `right` position.

The conditional logic in this function is the same as the conditional logic in Listing 4.24: the difference is that the function is invoked recursively with the updated value of either left or right instead of executing the code in a `while` loop. The recursion continues until either (1) the value of `left` is greater than the value of `right`, or (2) `item` is found in `arr1`,

The next portion of Listing 4.25 initializes the `NumPy` array `arr1` with five numbers, along with values for `item`, `left`, and `right`, and then invokes the function `binary_search()` to determine whether or not item appears in the array `arr1`. Launch the code in Listing 4.25 and you will see the following output:

```
array: [  1   3   5 123 400]
found 123 in position 3
```

WELL-KNOWN SORTING ALGORITHMS

There are numerous sorting algorithms, each of which have a best case, average case, and worst case in terms of performance. Interestingly, some algorithms can perform the worst when a given array is already sorted.

The following subsections contain code samples for the following well-known sort algorithms:

- bubble sort
- merge sort
- quick sort

If you want to explore sorting algorithms in more depth, perform an internet search for additional sorting algorithms.

Bubble Sort

A *bubble sort* involves a nested loop whereby each element of an array is compared with the elements to the right of the given element. If an array element is less than the current element, the values are "swapped," which means that the contents of the array will eventually be sorted from smallest to largest value.

Here is an example:

```
arr1 = np.array([40, 10, 30, 20]);
Item = 40;
Step 1: 40 > 10 so switch these elements:
arr1 = np.array([10, 40, 30, 20]);
Item = 40;
Step 2: 40 > 30 so switch these elements:
arr1 = np.array([10, 30, 40, 20]);
Item = 40;
Step 3: 40 > 20 so switch these elements:
arr1 = np.array([10, 30, 20, 40]);
```

As you can see, the smallest element is in the left-most position of the array `arr1`. Now repeat this process by comparing the second position (which is index 1) with the right-side elements.

```
arr1 = np.array([10, 30, 20, 40]);
Item = 30;
Step 4: 30 > 20 so switch these elements:
arr1 = np.array([10, 20, 30, 40]);
Item = 30;
Step 4: 30 < 40 so do nothing
```

As you can see, the smallest elements two elements occupy the first two positions in the array `arr1`. Now repeat this process by comparing the third position (which is index 2) with the right-side elements.

```
arr1 = np.array([10, 20, 30, 40]);
Item = 30;
Step 4: 30 < 40 so do nothing
```

The array `arr1` is now sorted in increasing order (in a left-to-right fashion). If you want to reverse the order so that the array is sorted in decreasing order (in a left-to-right fashion), simply replace the ">" operator with the "<" operator in the preceding steps.

Listing 4.26 displays the contents of the `bubble_sort.py` that illustrates how to perform a bubble sort on an array of numbers.

Listing 4.26: bubble_sort.py

```
import numpy as np
arr1 = np.array([40, 10, 30, 20]);

for i in range(1,arr1.length-1):
  for j in range(i+1,arr1.length):
    if(arr1[i] > arr1[j]):
      temp = arr1[i];
      arr1[i] = arr1[j];
      arr1[j] = temp;
```

You can manually perform the code execution in Listing 4.26 to convince yourself that the code is correct. (Hint: It's the same sequence of steps that you saw earlier in this section.) Launch the code in Listing 4.26 and you will see the following output:

```
initial: [40 10 30 20]
sorted:  [10 20 30 40]
```

Find Anagrams in a List of Words

Recall that the word `word1` is an anagram of `word2` if the letters in `word2` are a permutation of the letters in `word1`. Listing 4.27 displays the contents of the `anagrams.py` that illustrates how to check if two words are anagrams of each other.

Listing 4.27: anagrams.py

```
import numpy as np

words = np.array([["abc"],["evil"],["Z"],["cab"],["live"],["Z"],["xyz"],["zyx"],["bac"],["Z"]])
print("=> Initial words:")
print(words)
print()

marked = np.zeros(len(words))
marked = marked.astype(int)

switched = np.array([])
for i in range(0,len(words)-1):
  if(marked[i] == 0):
    sorti = sorted(words[i][0])
    switched = np.append(switched,words[i])
    #print("2switched:",switched)

    for j in range(i+1,len(words)):
      if(marked[j] == 0):
        sortj = sorted(words[j][0])
        #print("wordi:",words[i],"sorti:",sorti,"sortj:",sortj)
        if(sorti == sortj):
          #print(sorti,"and",sortj,"are anagrams")
          switched = np.append(switched,words[j])
          #print("3switched:",switched)
          marked[i] = 1
          marked[j] = 1
          #print("3marked:", marked)

print("=> Adjacent anagrams:")
print(switched)
```

Listing 4.27 initializes the variable `words` as a `NumPy` array of one-dimensional strings, and then displays its contents. Next, the variable `marked` is initialized as a `NumPy` array of length `len(words)`, where each element contains the value 0, after which the elements of marked are treated as integer values. In addition, the variable `switched` is initialized as an empty `NumPy` array.

The next portion of Listing 4.27 contains a loop that ranges from 0 to `len(words)-1` with the loop variable ii. The main block of this loop is controlled by conditional logic that checks whether or not `marked[i]` equals 0: if so, then the variables sort and switched are initialized, followed by another loop is executed from `i+1` to `len(words)` with the loop variable `j`.

The preceding loop contains similar condition logic as the outer loop: If `marked[j]` equals 0, then the variable sorta is initialized and compared with `sorti` that was previously initialized. If the two variables are equal, then `marked[i]` and `marked[j]` are set equal to 0.

The final code snippet in Listing 4.27 displays the contents of `switched`, which contains a list of adjacent anagrams. Launch the code in Listing 4.27 and you will see the following output:

```
=> Initial words:
[['abc']
['evil']
['Z']
['cab']
['live']
['Z']
['xyz']
['zyx']
```

```
['bac']
['Z']]


=> Adjacent anagrams:
['abc' 'cab' 'bac' 'evil' 'live' 'Z' 'Z' 'Z' 'xyz' 'zyx']
```

MERGE SORT

A *merge sort* is a divide-and-conquer algorithm that merges two arrays of sorted values. In the following subsections, you will see three different ways to perform a merge sort. The first code sample involves a third array, whereas the second and third code samples do not require a third array. Moreover, the third code sample involves one `while` loop whereas the second code sample involves a pair of nested loops, which means that the third code sample is simpler and also more memory efficient.

Merge Sort With a Third Array

The simplest way to merge two arrays involves copying elements from those two arrays to a third array, as shown here:

```
   A          B          C
+-----+    +-----+    +-----+
|  20 |    |  50 |    |  20 |   A
|  80 |    |  70 |    |  50 |   B
| 200 |  + | 100 |  = |  70 |   B
| 300 |    +-----+    |  80 |   A
| 500 |               | 100 |   B
+-----+               | 200 |   A
                      | 300 |   A
                      | 500 |   A
                      +-----+
```

The right-most column in the preceding diagram lists the array (either `A` or `B`) that contains each number. As you can see, the order `ABBABAAA` switches between array `A` and array `B`. However, the final three elements are from array `A` because all the elements of array `B` have been processed. Two other possibilities exist: Array `A` is processed and `B` still has some elements, or both `A` and `B` have the same size. Of course, even if `A` and `B` have the same size, it's still possible that the final sequence of elements are from a single array.

For example, array `B` is longer than array `A` in the example below, which means that the final values in array `C` are from `B`:

```
A = [20,80,200,300,500]
B = [50,70,100]
```

The following example involves array `A` and array `B` with the same length:

```
A = [20,80,200]
B = [50,70,300]
```

The next example also involves list `A` and list `B` with the same length, but all the elements of `A` are copied to `B` and then all the elements of `B` are copied to `C`:

```
A = [20,30,40]
B = [50,70,300]
```

Listing 4.28 displays the contents of the `merge_sort1.py` that illustrates how to perform a merge sort on two arrays of numbers.

Listing 4.28: merge_sort1.py

```
import numpy as np

items1 = np.array([20, 30, 50, 300])
items2 = np.array([80, 100, 200])

def merge_items():
  items3 = np.array([])
  ndx1 = 0
  ndx2 = 0

  # 1) always add the smaller element first:
  while(ndx1 < len(items1) and ndx2 < len(items2)):
    #print("items1 data:",items1[ndx1],"items2 data:",items2[ndx2])

    data1 = items1[ndx1]
    data2 = items2[ndx2]
    if(data1 < data2):
      #print("adding data1:",data1)
      items3 = np.append(items3,data1)
      ndx1 += 1
    else:
```

```
        #print("adding data2:",data2)
        items3 = np.append(items3,data2)
        ndx2 += 1

    # 2) append any remaining elements of items1:
    while(ndx1 < len(items1)):
        #print("MORE items1:",items1[ndx1])
        items3 = np.append(items3,data1)
        ndx1 += 1

    # 3) append any remaining elements of items2:
    while(ndx2 < len(items2)):
        #print("MORE items2:",items2[ndx2])
        items3 = np.append(items3,data2)
        ndx2 += 1
    return items3

# display the merged list:
items3 = merge_items()
print("items1:",items1)
print("items2:",items2)
print("items3:",items3)
```

Listing 4.28 initializes the `NumPy` arrays `items1` and `items2`, followed by the function `merge_items()` that creates an empty `NumPy` array `items3` and the scalar variables `ndx1` and `ndx2` that keep track of the current index position in `items1` and `items2`, respectively.

The key idea is to compare the value of `items1[ndx1]` with the value of `items2[ndx2]`. If the smaller value is `items1[ndx1]`, then this value is appended to `items3` and `ndx1` is incremented. Otherwise, `items2[ndx2]` is appended to `items3` and `ndx2` is incremented.

The second part of the function `merge_items()` contains a loop that appends any remaining items in `items1` to `items3`, followed by another loop that appends any remaining items in `items2` to `items3`. The final portion of Listing 4.28 invokes the `merge_items()` function and then displays the contents of `items1`, `items2`, and `items3`.

There are several points to keep in mind regarding the code in Listing 4.28. First, the initial loop in `merge_items()` iterates through both `items1` and `items2` until the final element is reached in one of these two arrays: consequently, only one of these two arrays can be non-empty (and possibly both are empty), which means that only the second loop or the third loop is executed, but not both.

Second, this algorithm will only work if the elements in arrays `items1` and `items2` are sorted: To convince yourself that this is true, change the elements in `items1` (or in `items2`) so that they are no longer sorted and you will see that the output is incorrect. Third, this algorithm populates the array `items3` with the sorted list of values; later you will see an example of a merge sort that does not require the array `items3`. Now launch the code in Listing 4.28 and you will see the following output:

```
items1: [ 20  30  50 300]
items2: [ 80 100 200]
items3: [ 20  30  50  80 100 200 300]
```

Merge Sort Without a Third Array

Listing 4.29 displays the contents of the `merge_sort2.py` that illustrates how to perform a merge sort on two *sorted* arrays without using a third array.

Listing 4.29: merge_sort2.py

```
import numpy as np

items1 = np.array([20, 30, 50, 300, 0, 0, 0, 0])
items2 = np.array([80, 100, 200])

print("merge items2 into items1:")
print("INITIAL items1:",items1)
print("INITIAL items2:",items2)

def merge_arrays():
  ndx1 = 0
  ndx2 = 0
  last1 = 4 # do not count the 0 values

  # merge elements of items2 into items1:
  while(ndx2 < len(items2)):
    #print("items1 data:",items1[ndx1],"items2 data:",items2[ndx2])
    data1 = items1[ndx1]
    data2 = items2[ndx2]

    while(data1 < data2):
      prev1 = ndx1
      #print("incrementing ndx1:",ndx1)
      ndx1  += 1
      data1 = items1[ndx1]
```

```
      for idx3 in range(last1,ndx1,-1):
        #print("shift",items1[idx3],"to the right")
        items1[idx3] = items1[idx3-1]

      # insert data2 into items1:
      items1[ndx1] = data2
      ndx1   = 0
      ndx2  += 1
      last1 += 1
      #print("=> shifted items1:",items1)

merge_arrays()
print("UPDATED items1:",items1)
```

Although Listing 4.29 is an implementation of a merge sort algorithm, it differs from Listing 4.28 because a third array (such as `items3` in Listing 4.28) is not required. As you can see, Listing 4.28 starts by initializing two `Numpy` arrays `items1` and `items2` with integer values, and then displays their contents.

However, there is a key difference: The right-most four elements of `items1` are `0`: these values will be replaced by the elements in `items2`, whose length is 3 (i.e., smaller than the number of available "zero" slots).

The next portion of Listing 4.29 defines the function `merge_arrays()` that starts by defining the scalar variables `ndx1` and `ndx2` that keep track of the current index position in `items1` and `items2`, respectively. The variable `last1` is initialized as 4, which is the right nonzero element in `items1`.

Listing 4.29 then defines a loop that iterates through the elements of `items2` in order to determine where to insert each of its elements in `items1`. Specifically, for each element `items2[ndx2]`, another loop determines determines the index position `ndx1` to insert `items2[ndx2]`. Note that before the insertion can be performed, the code shifts non-zero values to the right by one index position. As a result, an "open slot" becomes available for inserting `items2[ndx2]`. The final portion of Listing 4.29 invokes the function and then prints the contents of `items1`.

Keep in mind the following point: This code sample relies on the assumption that the right-most four values are 0 and that none of these values is a "legal" value in the array `items1`. However, the code sample in the next section removes this assumption. Now launch the code in Listing 4.29 and you will see the following output:

```
merge items2 into items1:
INITIAL items1: [ 20  30  50 300   0   0   0   0]
INITIAL items2: [ 80 100 200]
UPDATED items1: [ 20  30  50  80 100 200 300   0]
```

Merge Sort: Shift Elements From End of Lists

In this scenario we assume that matrix `A` has enough uninitialized elements at the end of the matrix in order to accommodate all the values of matrix `B`, as shown here:

```
   A          B          A
+-----+    +-----+    +-----+
|  20 |    |  50 |    |  20 |   A
|  80 |    |  70 |    |  50 |   B
| 200 |  + | 100 |  = |  70 |   B
| 300 |    +-----+    |  80 |   A
| 500 |               | 100 |   B
| xxx |               | 200 |   A
| xxx |               | 300 |   A
| xxx |               | 500 |   A
+-----+               +-----+
```

Listing 4.30 displays the contents of the `merge_sort3.py` that illustrates how to perform a merge sort on two *sorted* arrays without using a third array.

Listing 4.30: merge_sort3.py

```
import numpy as np

items1 = np.array([20, 30, 50, 300])
items2 = np.array([80, 100, 200])
last1 = len(items1)
last2 = len(items1)

print("=> merge items2 into items1 <=")
print("INITIAL items1:",items1)
print("INITIAL items2:",items2)

# append None to items1 for "empty slots":
for i in range(0,len(items2)):
  items1 = np.append(items1,None)
#print("AFTER  items1:",items1)

len1 = len(items1)
```

```
len2 = len(items2)

# start from the end of items1 and items2
# and shift items to the end of items1
def merge_arrays(items1,items2,len1,len2):
  ndx1 = len1-1
  ndx2 = len2-1
  last1 = len(items1)-1
  last2 = len(items2)-1

  # merge elements of items2 into items1:
  while(ndx1 >=0 and ndx2 >=0):
    #print("ndx1:",ndx1, "ndx2:",ndx2)
    data1 = items1[ndx1]
    data2 = items2[ndx2]
    #print("Bitems1 data:",data1,"ndx1:",ndx1)
    #print("Bitems2 data:",data2,"ndx2:",ndx2)

    if(data1 > data2):
      items1[last1] = data1
      ndx1 -= 1
      last1 -= 1
    else:
      items1[last1] = data2
      ndx2 -= 1
      last1 -= 1
    #print("Citems1:",items1)
    #print("Citems2:",items2)

merge_arrays(items1,items2,last1,len2)
print("MERGED items1:",items1)
```

The code for Listing 4.30 does not require an inner loop, and therefore Listing 4.30 is simpler than Listing 4.29. The code starts with two sorted arrays `items1` and `items2`, after which `items1` is padded with the value `None` so that it can accommodate the integers from `items2`. Launch the code in Listing 4.30 and you will see the following output:

```
merge items2 into items1:

=> merge items2 into items1 <=
INITIAL items1: [ 20  30  50 300]
INITIAL items2: [ 80 100 200]
MERGED items1: [20 30 50 80 100 200 300]
```

Quick Sort

Quick sort is a divide-and-conquer sorting algorithm that selects a so-called “pivot” element that splits an array into two parts (i.e., the elements on the left side and the elements on the right side of the pivot element). The pivot element can be selected in various ways, such as:

- the first element
- the last element
- a random element

The key idea regarding the quick sort algorithm is to select a pivot element x and then perform the following in linear time:

- Determine the correct location for x in the array.
- Place smaller elements on the left side of x.
- Place larger elements on the right side of x.

Listing 4.31 displays the contents of the `Python` file `quick_sort.py` that illustrates how to perform a quick sort on an array of numbers.

Listing 4.31: quick_sort.py

```
def partition(left_idx, right_idx, array):
  # initialize pivot's index to left_idx
  pivot_idx = left_idx
  pivot_val = array[pivot_idx]

  # when left_idx pointer crosses right_idx pointer
  # then swap the pivot with right_idx pointer
  while left_idx < right_idx:
    # increment left_idx pointer till it exceeds pivot
    while left_idx < len(array) and array[left_idx] <= pivot_val:
      left_idx += 1

    # decrement right_idx pointer till it's less than pivot
    while array[right_idx] > pivot_val:
      right_idx -= 1

    # if left_idx and right_idx have not crossed
    # swap the numbers on left_idx and right_idx
```

```
      if(left_idx < right_idx):
        array[left_idx],array[right_idx]=array[right_idx],array[left_idx]

    # swap pivot with right_idx pointer:
    array[right_idx], array[pivot_idx]=array[pivot_idx],array[right_idx]

    # right_idx pointer divides the array into 2 subarrays
    return right_idx

def quick_sort(left_idx, right_idx, array):
  if (left_idx < right_idx):
    # part_idx is partitioning index
    # array[part_idx] is at right_idx
    part_idx = partition(left_idx, right_idx, array)

    # sort both sides of partition:
    quick_sort(left_idx, part_idx - 1, array)
    quick_sort(part_idx + 1, right_idx, array)


array = [ 10, 7, 8, 9, 1, 5 ]
print(f'Initial array: {array}')

quick_sort(0, len(array) - 1, array)
print(f'Sorted array:  {array}')
```

Listing 4.31 starts with the definition of the `partition()` function that takes three parameters `left_idx`, `right_idx`, and `array`, which represent the left index, the right index, and an array variable, respectively.

The next portion of Listing 4.31 is an outer loop that executes while the value of the `left_idx` variable is less than the right_idx variable. During each iteration of the outer loop, another `while` loop executes and increment `left_idx` by 1 as long as the value of left_idx is less than the length of array *and* the value of `array[left_idx]` is less than or equal to the pivot value.

Next, another loop decrements the value of the variable `eight_idx` as long as the value of `array[right_idx]` is greater than the value of the value of `right_idx` value. Thus, `left_idx` moves in a left-to-right fashion through the elements of array, whereas `right_idx` moves in a right-to-left fashion through the elements of array.

The next snippet of conditional logic checks if left_idx is less than `right_idx`, and if so, then array "swaps" the values in index `left_idx` and index `right_idx`, as shown here:

```
array[left_idx], array[right_idx] = array[right_idx], array[left_idx]
```

After the outer loop has completed execution, the next portion of the `partition()` function swaps the values of the index position `right_idx` and the index position `pivot_idx`, as shown here:

```
array[right_idx],array[pivot_idx] = array[pivot_idx],array[right_idx]
```

The final code snippet in the `partition()` function returns the value of the `right_idx` variable.

The next portion of Listing 4.31 is the `quick_sort()` function that has the same parameters as the `partition()` function, along with conditional logic checks if `left_idx` is less than `right_idx`. If the latter is true, then the variable `part_idx` is initialized with the result of invoking the `partition()` function, after which the `quick_sort()` function is recursively invoked twice, as shown here:

```
quick_sort(start, part_idx - 1, array)
quick_sort(part_idx + 1, end, array)
```

The final portion of Listing 4.31 initializes the variable array with a list of integers, invokes the `quick_sort()` function, and then displays the sorted array. Now launch the code in Listing 4.31 and you will see the following output:

```
Initial array: [10, 7, 8, 9, 1, 5]
Sorted array:  [1, 5, 7, 8, 9, 10]
```

SUMMARY

This chapter started with an introduction to one-dimensional vectors and how to calculate their length or magnitude, as well as the inner product of pairs of vectors. Then you saw how to perform various tasks involving numbers, such as multiplying and dividing two positive integers via recursive addition and subtraction, respectively.

In addition, you learned about working with strings, and how to check a string for unique characters, how to insert characters in a string, and how to find permutations of a string. Next, you learned about determining whether or not a string is a palindrome.

Moreover, you learned how to calculate the transpose of a matrix, which is the equivalent of rotating a bitmap of an image by 90 degrees.

Then you learned about search algorithms such as linear search and binary search (iterative and recursive). Finally, you learned about well-known sort algorithms, such as the bubble sort, the merge sort (with three variations), and the quick sort.

CHAPTER 5

BUILT-IN FUNCTIONS AND CUSTOM CLASSES

This chapter introduces you to some Python built-in functions, how to create custom classes in Python, and object-oriented concepts such as inheritance and polymorphism.

The first part of this chapter discusses the Python functions such as the filter() function, the map() function, and the reduce() function. You will also learn something about lambda functions, which are often used in conjunction with these same Python functions. The second part of this chapter shows you how to define custom classes, and how to manage lists of objects that are instances of your custom Python classes.

The final portion of this chapter contains a light introduction to encapsulation, single and multiple inheritance, and polymorphism in Python. There are many subtle points involving inheritance and object-oriented programming, and after you have read this chapter you can perform a "deep dive" into this topics in order to write object-oriented Python code.

A PYTHON MODULE VERSUS PACKAGE

A module generally contains definitions, functions, and Python code, and it "lives" in a file with a .py extension. A module can also import other modules, and by convention, the import statements are placed at the top of the file (but this is not a strict requirement). Note that zip files and DLL files can also be modules.

However, when a module imports a directory, the presence of the file __init__.py is significant because Python will then treat the directory as a package. The file __init__.py can contain some initialization code for the package (in fact, it can even be empty), and such a file appears in each subdirectory that must be treated as a package by Python.

As you have seen in previous chapters, a module uses the import statement to import a module, and this can be accomplished in various ways. You can import a single module from a package, as shown here:

```
import myutils.xmlparse
```

The preceding code imports the xml submodule myutils.xml, and it must be fully-qualified in your code:

```
myutils.xmlparse(xmlDoc)
```

Another way to import the submodule xmlparse is here:

```
from myutils import xmlparse
```

Although the preceding code imports the xml submodule xmlparse, the latter is available without the package prefix, as shown here:

```
xmlparse(xmlDoc)
```

You can even import a single function from a module, as shown here:

```
from myutils.xmlparse import parseDOM
```

PYTHON FUNCTIONS VERSUS METHODS

A method is a function, but a function is not necessarily a method. A method is a function that is "attached" to an object or Class. Thus, str.upper() is a method, whereas sorted() is a function.

In addition, functions can be available as methods, which happens to be the case for the functions in the re module: re.sub is a function, and if you make a regex object by compiling a pattern, many of the module functions are also available as methods on the resulting object. An example is here:

```
>>> import re
>>> regex = re.compile("\d+")
>>> regex.sub # this is a method
```

```
>>> re.sub # this is a function
```

The distinction between function and method is whether the function is defined in a class or not. Functions in a module are just functions, functions in class are methods of the class or methods of the resulting objects.

In addition, function is a more general term (all methods are functions but not all functions are methods). Thus, the word *function* is used as a "generic term," whereas the word *method* is used specifically with regard to classes or objects (the method of the list type, the methods of the `str` type, and so forth).

FUNCTIONALLY ORIENTED PROGRAMMING IN PYTHON

`Python` supports methods (called *iterators* in `Python` 3), such as `filter()`, `map()`, and `reduce()` that are very useful when you need to iterate over the items in a list, create a dictionary, or extract a subset of a list. These iterators are discussed in the following subsections.

The Python `filter()` Function

The *filter* function enables you to extract a subset of values based on conditional logic. The following example returns a list of odd numbers between 0 and 15 inclusive that are multiples of 3:

```
>>> range(0,15)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
>>> def f(x): return x % 2 != 0 and x % 3 == 0
...
>>> filter(f, range(0, 15))
[3, 9]
>>>
```

The Python `map()` Function

The Python `map()` command is a built-in function that applies a function to each item in an iterable. The `map(func, seq)` calls `func(item)`, where `item` is an element in a sequence `seq`, and returns a list of the return values.

Listing 5.1 displays the contents of `Map1.py` that illustrates how to use the `map()` function to compute the cube and fourth power of a set of numbers.

Listing 5.1: Map1.py

```
def cube(x): return x*x*x
def fourth(x): return x*x*x*x

x1 = map(cube,  range(1, 5))
x2 = map(fourth, range(1, 5))

print (x1)
print (x2)
```

Listing 5.1 starts with the definition of two functions called `cube()` and `fourth()`, each of which takes a single numeric value. The `cube()` function returns the cube of its argument, and the `fourth()` function returns the fourth power of its argument.

The next portion of Listing 5.1 contains two invocations of the `map()` function. The first invocation specifies the `cube()` function as the first parameter, and the integers between 1 and 4 inclusive as the second parameter via the `range()` function. The second invocation specifies the `fourth()` function as the first parameter, along with the integers between 1 and 4 inclusive. The output from Listing 5.1 is here:

```
[1, 8, 27, 64]
[1, 16, 81, 256]
```

The Python lambda Operator

The lambda operator (or lambda function) enables you to define "anonymous" functions that are often used in combination with the functions `filter()`, `map()`, and `reduce()`. An example of a lambda function that adds two arguments is here:

```
>>> f = lambda x,y : x + y
>>> f(2,3)
5
```

The next section shows you how to use this lambda function with the `reduce()` function in `Python`.

The Python reduce() Function

The `reduce(func, seq)` function returns a single value constructed by calling the binary function `func` on the first two items of the sequence `seq` to compute a result, and then applies `func` on that result and the *next* item in `seq`, and so on until a single value is returned. Thus the `reduce()` function repeatedly performs a pair-wise reduction (and hence its name) on a sequence until a single value is computed.

As an illustration, the following example defines a lambda function that adds two numbers and a `range(1,6)` that calculates the sum of the numbers `1` through `6`:

```
>>> f = lambda x,y: x+y
>>> reduce(f, range(1,6))
15
```

In case the functionality of the `reduce()` function is new to you, there are other scenarios that have similar functionality. For example, recall that multiplication of two numbers is implemented as repeated addition (along with a shift operator). As a second example, `NoSQL` databases perform operations using map/reduce algorithms and the reduce portion has a similar implementation.

IMPORTING CUSTOM PYTHON MODULES

In addition to importing `Python` Standard Library modules, you can import your custom modules into other custom modules. Listing 5.2 and Listing 5.3 display the contents of `Double.py` and `CallDouble.py` that illustrate this functionality.

Listing 5.2: Double.py

```
def double(num):
    return 2*num

result = double(5)
print 'double 5 =', result
```

Listing 5.2 defines the function `double()` that returns 2 times its argument, followed by the variable `result` that is assigned the value of `double(5)`. If you invoke Listing 5.2 from the `Python` interpreter or launch the program from the command line, you will see the following output:

```
double 5 = 10
```

Listing 5.3: CallDouble.py

```
import Double
```

Listing 5.3 contains one line of code: an `import` statement that imports the `Double` module that is displayed in Listing 5.2. Launch Listing 5.3 from the command line and the output is shown here:

```
double 5 = 10
```

The combination of Listing 5.2 and Listing 5.3 demonstrates how easy it is to import a custom `Python` module. However, you obviously need the flexibility of invoking imported functions with different values.

Listing 5.4 and Listing 5.5 display the contents of `Triple.py` and `CallTriple.py` that illustrate how to achieve this flexibility.

Listing 5.4: Triple.py

```
def triple(num):
    return 3*num
```

Listing 5.4 contains only the definition of the function `triple()` that returns 3 times its argument, and there are no invocations of that function or any print statements.

Listing 5.5: CallTriple.py

```
from Triple import triple

print '3 times 4 is:', triple(4)
print '3 times 9 is:', triple(9)
```

Launch Listing 5.5 from the command line and you will see the following output:

```
3 times 4 is: 12
3 times 9 is: 27
```

Suppose that `Triple.py` also contained a function called `quadruple()` and you wanted to import that function. You can do so with the following variation of the `import` statement:

```
from Triple import double, quadruple
```

If you want to import *all* the functions that are defined in `Triple.py`, use this form of the

import statement:

```
from Triple import *
```

This concludes the brief overview regarding how to import custom modules. The next section in this chapter shows you how to define your own custom classes in Python.

HOW TO CREATE CUSTOM CLASSES

The Python language supports the creation of custom classes, which is available in other programming languages, such as Java and C++. However, there is the "Python way" of creating custom classes, which you will learn about in this chapter.

As a starting point, Listing 5.6 displays the contents of SimpleClass.py that illustrates how to define a simple custom class called BaseClass in Python.

Listing 5.6: SimpleClass.py

```
#!/usr/bin/env python

class BaseClass:
    def __init__(self):
        self.x = 3

    def main(self):
        print 'inside main'
        print 'x:',self.x

if __name__ == '__main__':
    baseClass = BaseClass()
    baseClass.main()
```

Listing 5.6 starts with the definition of the class BaseClass that contains two functions. The built-in "magic" function __init__ (explained in the next section) assigns the value 3 to the variable x. The main() function contains two print statements that display the string "inside main" and also the value of the variable x.

The next portion of Listing 5.6 uses conditional logic to instantiate the class BaseClass and assign that instance to the variable baseClass, and then invoke the main() method of baseClass. The output from launching SimpleClass.py is here:

```
inside main
x: 3
```

CONSTRUCTION AND INITIALIZATION OF OBJECTS

In the previous section, you saw an example of the __init__ function, which is one of the "magic" methods that exist in Python. Three magic methods are shown here:

```
__init__
__new__
__del__
```

In brief, the __new__ method is invoked in order to create an instance of a class, which occurs when you invoke the following type of statement:

```
myObj = MyClass()
```

Next, any arguments during creation time are passed to the "initializer" method __init__, which specifies the behavior of an object during initialization. You can think of the __new__ method and the __init__ methods as the "constructor" of an object.

For example, consider the following snippet:

```
myObj = MyClass('pizza', 25);
```

In the preceding code snippet, the values pizza and 25 are passed as arguments to the __init__ method.

Finally, when an object is deleted, the "destructor" method __del__ is invoked and it defines behavior during garbage collection. This method is useful when additional resources need to be deallocated. However, there is no guarantee that __del__ will be executed, so it's better to close resources (such as database connections and sockets) when they are no longer needed.

NOTE

If you are a beginner you rarely (if ever) need to use the __new__ and __del__ methods.

There are *many* other magic methods in Python (for comparisons, numeric functions, conversions, and so forth), and an extensive list of such methods is here:

http://docs.python.org/2/reference/datamodel.html#specialnames

COMPILED MODULES

The directory that contains the module `Triple.py` will also contain the compiled version called `Triple.pyc`, which is automatically generated by `Python` in order to improve performance. The contents of `Triple.pyc` are platform independent, and therefore machines of different architectures can share a `Python` module directory.

You can also invoke the `Python` interpreter with the `-O` flag, and `Python` will generate optimized code that is stored in `.pyo` files. In addition, `.pyc` files are ignored and `.py` files are compiled to optimized bytecode when the `–O` flag is used. Keep in mind that there is no difference in speed when a program is read from a `.pyc` or `.pyo` file versus from a `.py` file; the only difference is the load time.

CLASSES, FUNCTIONS, AND METHODS IN PYTHON

In high-level terms, a Python *function* is a block of code that:

- is called by name
- can be passed data to operate on (i.e., the parameters)
- can optionally return data (the return value)

All data that is passed to a function is explicitly passed. Contrastingly, a *method* is a block of code that

- is called by name
- is associated with an object

A method differs from a function in two ways:

- A method is implicitly passed the object for which it was called.
- A method is able to operate on data that is contained within the class.

Keep in mind that that an object is always an instance of a class. If you think of a class as the “definition,” then the object is an instance of that definition.

Instance variables in an object have values that are local to that object; in other words, two instances of the same class maintain distinct values for their variables.

However, the value of *class variables* is the same for all the objects that are instances of the same class. In the `Java` world, a variable that is declared as static is a class variable: if you change its value in one object, its new value is visible among all objects that are instances of the same class.

By way of comparison, methods in C++ are called member functions, and Java contains only methods (not functions). A method can manipulate private data that is defined in a class.

ACCESSORS AND MUTATORS VERSUS @PROPERTY

Object-oriented languages such as `Java` encourage the use of accessors and mutators (often called getters and setters) rather than direct access to a property. For example, if `x` is a property of a custom class, then the accessor method `getX()` returns the value of `x` and the mutator method `setX()` sets the value of `x` (you would also specify an argument in the case of the `setX()` method).

By contrast, `Python` has a `@property` decorator that lets you add getters and setters retroactively for attribute access. Consider the following example:

```
>>> class Foo(object):
...   @property
...   def foo(self):
...     return 4
>>> obj = Foo()
>>> obj.foo
4
```

The preceding code defines a class `Foo` with a method called `foo()`. The variable `obj` is an instance of the class `Foo`, and notice how it’s possible to write `obj.foo` in order to obtain the result (4). This functionality is possible because of the `@property` decorator in `Python`.

Consequently, you can define your custom `Python` classes by “allowing” attribute access, and if it becomes necessary to add get/set methods later on, you can do so without breaking any existing code.

NOTE

Accessors and mutators are common in languages such as Java, whereas direct access is preferred in Python.

CREATING AN EMPLOYEE CUSTOM CLASS

This section contains an example of defining a custom Python class to keep track of some employee-related information. In the OO world, this type of class is called a "value object" because its only purpose is to keep track of one or more properties (such as the properties of a mailing address or a customer).

The example in this section uses accessors for accessing property values as well as direct access so that you can see how to use both techniques.

Listing 5.7 displays the contents of the custom Python class Employee.py that keeps track of an employee's first name, last name, and title.

Listing 5.7: Employee.py

```
#!/usr/bin/env python

class Employee(object):
    def __init__(self,fname,lname,title):
        self.fname = fname
        self.lname = lname
        self.title = title

    def firstName(self):
        return self.fname

    def lastName(self):
        return self.lname

    def theTitle(self):
        return self.title

    def main(self):
        print 'fname:',self.fname
        print 'lname:',self.lname
        print 'title:',self.title

if __name__ == '__main__':
    emp1 = Employee('John','Smith','Director')
    emp1.main()
    print 'Last Name:',(emp1.lastName())
    print

    emp2 = Employee('Jane','Edwards','VP')
    emp2.main()
    print 'Last Name:',(emp2.lastName())
    print
```

Listing 5.7 contains the definition of the Employee class, which defines three functions firstName(), lastName(), and thetitle() that return the current employee's first name, last name, and title, respectively. In addition, the __init__ function contains initialization code that sets the values of the same three properties, and the main() function prints the values of these three properties.

The final portion of Listing 5.7 contains the standard Python idiom for distinguishing between direct execution (such as from the command line) versus the situation in which the module is simply imported into another Python module. In our case, this class will be launched directly, which means that the code block will instantiate emp1 and emp2, both of which are instances of the Employee class. In addition, the code initializes the properties for emp1 and emp2 and then prints the values of those properties by invoking the main() method.

In addition, you can retrieve the value of a property by invoking its associated method in much the same way that you would in other programming languages such as Java. An example of retrieving and then printing the last name of the first employee is here:

```
print 'Last Name:',emp1.lastName()
```

You can display the first name and title of the first employee by invoking emp1.firstName() and emp1.title(), respectively.

WORKING WITH A LIST OF EMPLOYEES

In the previous section, you learned how to create the custom Python class Employee that keeps track of three attributes of an employee. This section shows you how to create a custom Python class called Employees that creates a list of Employee objects, where each object contains information about a single employee. Of course, a real-world application would specify

numerous other attributes.

Listing 5.8 displays the contents of the Python module Employees.py that uses a Python list to keep track of multiple Employee objects, each of which represents information about a single employee.

Listing 5.8: Employees.py

```
#!/usr/bin/env python

from Employee import *

class Employees:
    def __init__(self):
        self.list = []

    def firstEmp(self):
        return self.list[0]

    def addEmp(self,emp):
        self.list.append(emp)

    def displayAll(self):
        for i in range(0,len(self.list)):
          emp = self.list[i]
          print 'First:',emp.firstName()
          print 'Last:', emp.lastName()
          print 'Title:',emp.theTitle()
          print '--------------'

if __name__ == '__main__':
    emp1 = Employee('John','Smith','Director')
    emp2 = Employee('Jane','Edwards','VP')
    emp3 = Employee('Dave','Jones','Manager')

    allEmps = Employees()

    allEmps.addEmp(emp1)
    allEmps.addEmp(emp2)
    allEmps.addEmp(emp3)

    allEmps.displayAll()
```

Listing 5.8 starts with an import statement that imports the definition of the Employee class that was defined in the previous section. Next, Listing 5.8 contains the definition of the Employee class that defines several methods.

The __init__ method simply initializes an empty list that will keep track of each employee object. The firstEmp() method returns the first employee object in the list, and the addEmp() method appends the current Employee instance to the list.

The displayAll() method iterates through the list of employees and prints the three properties of each Employee object. This functionality is possible because the Employee object was imported, and therefore its methods are accessible in Listing 5.8.

The output of Listing 5.8 is here:

```
First: John
Last: Smith
Title: Director
--------------
First: Jane
Last: Edwards
Title: VP
--------------
First: Dave
Last: Jones
Title: Manager
--------------
```

The code sample in this section (and the previous section) provides an example of how you can use a collection (in the English sense of the word) of Python classes to model a real-world scenario (i.e., tracking employees in a company). Although the syntax is different, other object-oriented languages (such as Java and C#) use a similar approach.

There are several ways in which you can enhance this code sample. First, you can use a database to persist employee-related information. A database can provide various benefits, such as enforcing transaction-related integrity and also enable you to deploy the application to different platforms.

Second, you can provide Web services that can perform similar functionality in a Web browser instead of the command line.

WORKING WITH LINKED LISTS IN PYTHON

You can use Python in order to create other data structures that are not a part of the Python

distribution. In this section you will learn how to create a *singly linked list* using custom Python classes.

Although they are not discussed in this chapter, you can create other related data structures, such as *doubly linked lists* and *circular lists*. Each node in a doubly linked list contains a reference to its predecessor and successor, whereas each node in a singly linked list contains only a reference to its successor.

A *circular list* can be a singly listed list or a doubly linked list; in addition, the "tail" or final node references the "head" or root node, thereby making the list circular.

The next section contains an example of a singly linked list in Python.

CUSTOM CLASSES AND LINKED LISTS

Listing 5.9 displays the contents of LLAndList.py that illustrates how to create a linked list where the nodes contain the values in a Python list.

Listing 5.9: LLAndList.py

```
class Node:
  def __init__(self):
    # contains the data
    self.data = None

    # reference to the next node
    self.next = None

# this creates a tail->head list
# instead of a head->tail list
class LinkedList:
  def __init__(self):
      self.curr_node = None

  # create and append a new node
  def add_node(self, data):
      new_node = Node()
      new_node.data = data

      # link new node to 'previous' node
      new_node.next = self.curr_node

      # current node equals the new node
      self.curr_node = new_node

  def print_items(self):
      node = self.curr_node
      while node:
          print node.data
          node = node.next

list1 = ['a', '12', 'b', '34', 'c', 'd']
myLL = LinkedList()

# add items to the linked list
for val in list1:
  myLL.add_node(val)

print 'List of Items:'
myLL.print_items()
```

Listing 5.9 contains the definition of the Node class, which creates a "value object" that will contain the value of each element in list1 via the data property. The Node class also defines the next property, whose value represents the next element in the list.

The next portion of Listing 5.9 defines the LinkedList class that performs some initialization in the __init__ method, and also defines the add_node() and print_items() methods.

The add_node() method adds a new node to the linked list by invoking the Node class and then updating the value of the next property appropriately. Finally, the print_items() method displays the data value of each node in the linked list. The output from Listing 5.10 is here:

```
List of Items:
d
c
34
b
12
a
```

CUSTOM CLASSES AND DICTIONARIES

Listing 5.10 displays the contents of LLAndDict.py that illustrates how to create a linked list where each node references a Python dictionary.

Listing 5.10: LLAndDict.py

```
class Node:
  def __init__(self):
    # contains the data
    self.data = None

    # reference to the next node
    self.next = None

# this creates a tail->head list
# instead of a head->tail list
class LinkedList:
  def __init__(self):
      self.curr_node = None

  # create and append a new node
  def add_node(self, data):
      new_node = Node()
      new_node.data = data

      # link new node to 'previous' node
      new_node.next = self.curr_node

      # current node equals the new node
      self.curr_node = new_node

  def print_items(self):
      node = self.curr_node
      while node:
          print node.data
          node = node.next

dict1 = {'a':'aaa', 'b':'bbb', 'c': 'ccc'}
myLL = LinkedList()

# add items to the linked list
for w in dict1:
  myLL.add_node(w+" "+dict1[w])

print 'List of Keys and Values:'
myLL.print_items()
```

Listing 5.10 contains code that is very similar to the previous section. The difference involves the following code block that uses a `Python dict` instead of a `Python` list:

```
dict1 = {'a':'aaa', 'b':'bbb', 'c': 'ccc'}
myLL = LinkedList()

# add items to the linked list
for w in dict1:
  myLL.add_node(w+" "+dict1[w])

print 'List of Keys and Values:'
myLL.print_items()
```

The preceding code block creates a node consisting of the concatenation of the key/value pairs of each element in the variable `dict1`. The output from Listing 5.10 is here:

```
List of Keys and Values:
b bbb
c ccc
a aaa
```

CUSTOM CLASSES AND PRIORITY QUEUES

In Chapter 3, you learned about the `Queue` data structure. In this section, you will see how to create and populate a priority queue with objects. Listing 5.11 displays the contents of `PriorityQueue.py` that illustrates how to create a priority queue and populate the queue with instances of the custom `Task` class.

Listing 5.11: PriorityQueue.py

```
import Queue
from random import randint

pLevel      = ''
taskCount   = 4
minPriority = 3
maxPriority = 10

q = Queue.PriorityQueue()

class Task(object):
```

```
  def __init__(self, priority, name):
      self.priority = priority
      self.name= name
      print 'Added a new task:', name
  def __cmp__(self, other):
      return cmp(self.priority, other.priority)

def displayTasks():
  while not q.empty():
    curr_Task = q.get()
    print 'Processing Task:', curr_Task.name

def addTasks():
  for i in range(0,taskCount):
    p = randint(minPriority, maxPriority);

    if(p < minPriority+maxPriority/4):
       pLevel = 'Low Priority'
    elif(p < minPriority+maxPriority/2):
       pLevel = 'Medium Priority'
    else:
       pLevel = 'High Priority'
    q.put(Task(p, pLevel))
  print

if __name__ == '__main__':
  addTasks()
  displayTasks()
```

Listing 5.11 starts by initializing the variable `q`, which is an instance of the `PriorityQueue` class in `Python`. Next, Listing 5.11 defines a `Task` class that performs some initialization in the `__init__` method and defines how to compare two items in the `__cmp__` method. In addition, the `displayTasks()` method displays the current set of tasks, and the `addTasks()` method adds a new task in the priority queue. The `addTasks()` method generates a random number for the priority of each new task, and then uses conditional logic to determine whether the task has low, medium, or high priority.

The final portion of Listing 5.11 invokes the `addTasks()` method, followed by the `displayTasks()` method. The output from Listing 5.11 is here:

```
Added a new task: Low Priority
Added a new task: Medium Priority
Added a new task: Medium Priority
Added a new task: Medium Priority

Processing Task: Low Priority
Processing Task: Medium Priority
Processing Task: Medium Priority
Processing Task: Medium Priority
```

OVERLOADING OPERATORS

By way of illustration, suppose that you want to extend the `add` operator in `Python`. You can do so by overloading the `__add__` method as follows:

```
class Test(object):
    def __init__(self): self.prop = 3
    def __add__(self, x):
        return self.prop + x
```

If you enter the following command at the command line:

```
Test() + 4
```

You will see the following output:

```
7
```

SERIALIZE AND DESERIALIZE DATA

Pickling is the process whereby a `Python` object hierarchy is converted into a byte stream. Generally you can pickle (serialize) any object if you can pickle every attribute of that object. Keep in mind that you cannot pickle classes, functions, and methods.

With `pickle` protocol v1, you cannot pickle open file objects, network connections, or database connections; however, you *can* pickle open file objects with `pickle` protocol v2.

`Python` enables you to pickle data in lists, dictionaries, and so forth, after which you can "depickle" (deserialize) that data.

NOTE

Pickle files can be hacked, so be careful if you receive a raw pickle file over the network, because it could contain malicious code to run arbitrary `Python` *when you attempt to depickle*

it.

Listing 5.12 displays the contents of `Serialize1.py` that illustrates how to serialize and then deserialize a `Python` object.

Listing 5.12: Serialize1.py

```
import pickle

# Some Python object
data = [1,2,3,4,5]
print 'original data:', data

f = open('testfile', 'wb')
pickle.dump(data, f)

s = pickle.dumps(data)

# Restore from a file
f = open('testfile', 'rb')
data = pickle.load(f)

# Restore from a string
data = pickle.loads(s)

print 'restored data:', data
```

Listing 5.12 starts with an `import` statement, followed by the `data` variable that is initialized as a list containing five numbers. Next, the file `testfile` is created and the pickled contents of data are stored in that file. The remainder of Listing 5.12 reverses the process and prints the depickled contents that match the original contents of the `data` variable. The output from Listing 5.12 is here:

```
original data: [1, 2, 3, 4, 5]
restored data: [1, 2, 3, 4, 5]
```

A minimalistic example of pickling a class is here:

```
import pickle

class MyClass:
  attribute = 'a simple attribute'

picklestring = pickle.dumps(MyClass)
```

ENCAPSULATION

One of the main reasons for public accessors and mutators is their ability to retrieve and update the values of private variables. The ability to "shield" instances of other classes from the internal implementation details of a given class is called *encapsulation*.

The advantage of encapsulation is the ability to change the inner workings of a class without changing the signature of the API. As a result, instances of other classes, as well as public APIs, can continue working correctly without worrying about updating the signature of the API (provided that the method is not deprecated and replaced with a new method that has a different signature).

SINGLE INHERITANCE

There are two types of single inheritance that you will encounter in programming languages. One common type is called "classical" *class-based inheritance* that you will encounter in strongly typed languages that perform compile-time checking for variables (such as Java and C++). The second type is *prototype-based inheritance* that you will encounter in functional languages such as `JavaScript`.

Class mechanisms in `Python` are slightly closer to C++, partly because both support multiple inheritance (discussed in the next section). On the other hand, `Java` supports only single class inheritance (but a `Java` class can implement multiple interfaces). However, all methods in `Python` and `Java` are virtual.

Languages such as `JavaScript` treat functions as "first class citizens" in the sense that they have the same "parity" as objects, whereas methods have a "subordinate" role in classic languages such as `Java` and C++ because methods only exist as part of a class definition.

Another consideration is whether or not functions (and methods) have so-called side effects, such as modifying the value of global or static variables. In the `XSLT` world all variables are treated as read-only variables, which eliminates the problems associated with side effects, but at the same time, `XSLT` is a specialized functional programming language that is arguably more difficult to master than imperative languages such as `Java` and C++.

As a simple example, Listing 5.13 displays the contents of `SingleInherit1.py` that illustrates inheritance in `Python`.

Listing 5.13: SingleInherit1.py

```
class ClassA:
   def __init__(self):
     print 'Hello from A'

   def func(self):
     print 'Hello again from A'

class ClassB(ClassA):
   def __init__(self):
     print 'Hello from B'

  #def func(self):
  #  print 'Hello again from B'

if __name__ == '__main__':
    instanceA = ClassA()
    instanceB = ClassB()

    print
    print 'instanceA:'
    instanceA.func()
    print 'instanceB:'
    instanceB.func()
```

Listing 5.13 defines `ClassA` with a `print()` statement in the function `__init__` as well as a `print()` statement in function `func()`. Next, `ClassB` is defined as a subclass or derived class of `ClassA`. Notice that `func()` is "commented out" in `ClassB`, and that the `__init__` function also contains a print statement.

The final code block in Listing 5.13 instantiates `instanceA` of `ClassA` and `instanceB` of `ClassB`, followed by some `print` statements. The output from Listing 5.13 is here:

```
Hello from A
Hello from B

instanceA:
Hello again from A
instanceB:
Hello again from A
```

A CONCRETE EXAMPLE OF INHERITANCE

Listing 5.14 displays the contents of `SingleInherit1.py` that contains an abstract base class (i.e., it cannot be instantiated) and two subclasses `Vegetarian` and `Carnivore`, along with various relevant methods.

Listing 5.14: SingleInherit1.py

```
import numpy as np

class FoodPrefs:
  def __init__(self):
    print("Inside __init__ of FoodPrefs")

  def add(self, items):
    raise NotImplementedError

  def remove(self, items):
    raise NotImplementedError

  def show_list(self):
    raise NotImplementedError

  def blocked_list(self):
    raise NotImplementedError

class Vegetarian(FoodPrefs):
  def __init__(self):
    super().__init__()
    self.myprefs = np.array([])
    print("[vegetarian] myprefs = ",self.myprefs)
    self.blocked_items = np.array(["Steak", "Chicken", "Sardines"])

  def add(self, items):
    add_list = np.array([])
    blocked = self.blocked_list()

    for item in items:
      if item not in blocked:
```

```
        add_list = np.append(add_list,item)
      else:
        print("=> Cannot add item: ",item)

    self.myprefs = np.append(self.myprefs,add_list)
    print(f"[vegetarian] added items to preferences: {items}")
    #print("[vegetarian] updated list of items: ",self.myprefs)

  def remove(self, items):
    print(f"[vegetarian] removed items: {items}")

  def show_list(self):
    print("[vegetarian] full list of food items:")
    print(self.myprefs)

  def blocked_list(self):
    return self.blocked_items

class Carnivore(FoodPrefs):
  def __init__(self):
    super().__init__()
    self.myprefs = np.array([])
    print("[carnivore] myprefs = ",self.myprefs)

  def add(self, items):
    self.myprefs = np.append(self.myprefs,items)

    print(f"[carnivore] added items to preferences: {items}")
    #print("[carnivore] updated list of items: ",self.myprefs)

  def remove(self, items):
    print(f"[carnivore] removed items: {items}")

  def show_list(self):
    print("[carnivore] full list of food items:")
    print(self.myprefs)

  def blocked_list(self):
    self.blocked_items = []

print("--------------------------")
veggie = Vegetarian()
veggie.add(["Chickpeas", "Lentils", "Kale"])
veggie.add(["Tomatoes", "Garlic"])
veggie.add(["Steak", "Chicken"])
veggie.show_list()
print("--------------------------\n")

print("--------------------------")
carnie = Carnivore()
carnie.add(["Steak", "Chicken", "Sardines"])
carnie.show_list()
print("--------------------------\n")
```

Listing 5.14 defines `FoodPrefs` with a `print()` statement in the function `__init__` simply to show that this method has been executed later in the code.

The next portion of Listing 5.14 defines the `Vegetarian` class as a subclass of `FoodPrefs`, which contains the methods `add()`, `remove()`, `show_list()`, and `blocked_list()` in order to add, remove, display items, and specify invalid items;respectively, for a vegetarian diet.

The next portion of Listing 5.14 defines the `Carnivore` class as a subclass of `FoodPrefs`, which also contains the methods `add()`, `remove()`, and `show_list()` in order to add, remove, display items, and specify invalid items, respectively, for the diet of a carnivore. Note that the method `blocked_list()` is an empty `NumPy` array, which you can modify if you need to specify invalid food items.

The final code block in Listing 5.14 instantiates `veggie` of type `Vegetarian` and `carnie` of type `Carnivore`, followed by the invocation of several methods. Now launch the code in Listing 5.15 and you will see the following output:

```
--------------------------
Inside __init__ of FoodPrefs
[vegetarian] myprefs =  []
[vegetarian] added items to preferences: ['Chickpeas', 'Lentils', 'Kale']
[vegetarian] added items to preferences: ['Tomatoes', 'Garlic']
=> Cannot add item:  Steak
=> Cannot add item:  Chicken
[vegetarian] added items to preferences: ['Steak', 'Chicken']
[vegetarian] full list of food items:
['Chickpeas' 'Lentils' 'Kale' 'Tomatoes' 'Garlic']

--------------------------


--------------------------
Inside __init__ of FoodPrefs
[carnivore] myprefs =  []
```

```
[carnivore] added items to preferences: ['Steak', 'Chicken', 'Sardines']
[carnivore] full list of food items:
['Steak' 'Chicken' 'Sardines']

-------------------------
```

INHERITANCE AND OVERRIDING METHODS

If class A is a subclass (also called a "derived" class) of class B, then everything in B is accessible in A. In addition, class A can define methods that:

1. are unavailable in B
2. override methods in B

If class B and class A both contain a method called func(), then func() in class B can override the func() in class A. As strange as it might seem, a method of class A can call another method in class A that in turn can invoke a method of class B that overrides it. Python has two built-in functions that work with inheritance:

* the isinstance() method checks the type of an instance
* the issubclass() method checks class inheritance

For example, isinstance(myObj, int) evaluates to True only if myObj.__class__ is int or a subclass of int, whereas issubclass(bool, int) evaluates to True because bool is a subclass of int. On the other hand, issubclass(unicode, str) evaluates to False because unicode is not a subclass of str.

MULTIPLE INHERITANCE

The previous section showed you how to work with single inheritance and this section briefly discusses multiple inheritance in Python. *Multiple inheritance* means that a class can have more than one parent class.

If you do decide to use multiple inheritance, keep the following point in mind. Suppose that ClassC is a subclass of ClassA and ClassB as follows:

```
class ClassC(ClassA, ClassB):
```

In addition, suppose that Class A and Class B *both* contain a function called func() that is not defined in ClassC. Now consider the following code snippet, where classC is an instance of ClassC.

```
classC.func()
```

Since ClassC does not contain the definition of the function func(), Python searches for func() in the parent classes. Since the search is performed in a left-to-right fashion, the preceding code snippet execute the method func() that is defined in ClassA and not the method func() in ClassB.

As another example, suppose that ClassC is a subclass (from left to right) of ClassA1, ClassA2, and ClassA3 (in this order), and that the method func() is defined only in ClassA2 and ClassA3 but not in ClassA1 or in ClassC. Again consider the following snippet, where classC is an instance of ClassC:

```
classC.func()
```

Because of the left-to-right search rule, the preceding code snippet invokes the method func() in ClassA2 and not the function func() in ClassA3. Make sure that you remember this fact when you define classes that contain more than one parent class.

As a concrete example, Listing 5.15 displays the contents of the Python file MultipleInherit1.py that illustrates multiple inheritance in Python.

Listing 5.15: MultipleInherit1.py

```
class ClassA:
   def __init__(self):
     print 'Hello from A'

   def func(self):
     print 'Hello again from A'

class ClassB:
   def __init__(self):
     print 'Hello from B'

   def func(self):
     print 'Hello again from B'

class ClassC(ClassA, ClassB):
   def __init__(self):
```

```
     print 'Hello from C'

if __name__ == '__main__':
    instanceA = ClassA()
    instanceB = ClassB()
    instanceC = ClassC()

    print
    print 'instanceA:'
    instanceA.func()
    print 'instanceB:'
    instanceB.func()
    print 'instanceC:'
    instanceC.func()
```

Listing 5.15 contains code that is very similar to the code in the previous section, except that in this case `ClassC` is a derived class of the custom classes `ClassA` and `ClassB`. In addition, both `ClassA` and `ClassB` contain a function `func()` that is not defined in `ClassC`. The output from Listing 5.15 is here:

```
Hello from A
Hello from B
Hello from C

instanceA:
Hello again from A
instanceB:
Hello again from B
instanceC:
Hello again from A
```

Now reverse the order of the parent classes in the definition of `ClassC`:

```
class ClassC(ClassA, ClassB):
```

The only difference in the output is the final print statement, as shown here:

```
instanceC:
Hello again from B
```

Although there is no reason for the following class definition, `Python` allows you to specify multiple occurrences of the same parent class:

```
class ClassC(ClassA, ClassB, ClassA, ClassB):
```

POLYMORPHISM

In very simplified terms, `Python` polymorphism allows you to define methods that "accept" instances of different classes (as parameters) and yet perform the intended calculations correctly.

As another concrete example, Listing 5.16 displays the contents of `Polymorphism1.py` that defines two custom `Python` classes and a method that can be invoked with instances of both custom `Python` classes.

Listing 5.16: Polymorphism1.py

```
class Rect:
   def perimeter(self):
     print 'Perimeter of a rectangle'

class Square:
   def perimeter(self):
     print 'Perimeter of a square'

def calcPerimeter(obj):
     obj.perimeter()

if __name__ == '__main__':
    instanceR = Rect()
    instanceS = Square()

    print 'instanceR:'
    calcPerimeter(instanceR)
    print 'instanceS:'
    calcPerimeter(instanceS)
```

Listing 5.16 starts with the definition of the custom `Python` classes `Rect` and `Square`, each of which defines a `perimeter()` method. Next the function `calcPerimeter()` is defined, which takes one argument that can be an instance of the `Rect` class or the `Square` class.

The final portion of Listing 5.16 defines `instanceR` and `instanceS` that are instances of the custom classes `Rect` and `Square`, respectively. The `calcPerimeter()` method is invoke with each of these instances, and the correct method is invoked in both cases. The output from Listing 5.16 is here:

```
instanceR:
```

```
Perimeter of a rectangle
instanceS:
Perimeter of a square
```

There are several points to keep in mind when you work with polymorphism in your custom Python code. First, other languages might require Rect and Square to be derived classes of a common class. In this example, squares and rectangles are also parallelograms, so you could define the parent class PGram that contains properties of a parallelogram.

Second, there is the notion of "coding to an interface," which essentially means that you specify a base class as the argument of a method so that you can pass in any derived class of the base class. In the Java world, you specify an *interface* as an argument to a method, and that way you can pass in any concrete class that implements the specified interface.

A third point is that polymorphic behavior in idiomatic Python relies on "duck-typing" that is described succinctly here:

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

THE PYTHON ABC MODULE

Although Python does not provide interfaces (such as Java) or contracts, the Python abc (abstract base class) module provides abstract base classes a mechanism for specifying what methods must be implemented by implementation subclasses.

For example, you would expect that the semantics of a print() method involve printing some data, and not deleting data. The use of ABCs provides a sort of "understanding" about methods and their expected behavior. Thus, ABCs provide an intermediate solution between the free-form of Python and the stricter enforcement of statically typed languages.

Although this is an advanced topic (and actually beyond the intended scope of this book), more information about the Python abc module is here:

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

SUMMARY

This chapter introduced you to some useful Python functions that can simplify your custom Python code. Next you learned how to create your own custom Python classes in Python, and also how to work with linked lists in Python. You also got an introduction to encapsulation, single inheritance, multiple inheritance, and polymorphism in Python.

CHAPTER 6

RECURSION AND COMBINATORICS

This chapter consists of two main sections, the first of which contains various Python code samples that illustrate how to use recursion in Python. In addition to the typical form of recursion you will also learn how to use tail recursion in Python. The other significant section introduces you to some concepts in combinatorics, such as combination and permutations of objects.

The first part of this chapter contains a gentle introduction to recursion that you will learn through the Python code samples. For example, you will learn how to calculate the sum of an arithmetic series (e.g., the numbers from 1 to n) as well as a geometric series, along with how to calculate factorial values and Fibonacci numbers. In some cases, there are closed form solutions and also iterative solutions for these tasks.

If you are new to recursion, some of this material might be slightly daunting, but don't be alarmed. Usually, several iterations of reading the material and code samples will lead to a better understanding of recursion.

The second part of this chapter discusses concepts in combinatorics, such as permutations and combinations. Since a detailed coverage of combinatorics can fill an entire undergraduate course in mathematics, this section focuses on some rudimentary concepts in combinatorics.

If you prefer, you can read the material regarding combinatorics before the material for recursion. Perhaps you don't need recursion in your current tasks, so the first half of this chapter has limited relevance. However, keep in mind that recursion is unavoidable if you ever need to work with algorithms for trees and graphs. In fact, almost every algorithm that creates, updates, deletes elements (called nodes) in a tree structure or a graph involves recursion.

WHAT IS RECURSION?

Recursion-based algorithms can provide very elegant solutions to tasks that would be difficult to implement via iterative algorithms. For some tasks, such as calculating factorial values, the recursive solution and the iterative solution have comparable code complexity.

As a simple example, suppose that we want to add the integers from 1 to n (inclusive), and let n = 10 so that we have a concrete example. If we denote S as the partial sum of successively adding consecutive integers, then we have the following:

```
S = 1
S = 1 + 2
S = 1 + 2 + 3
. . .
S = 1 + 2 + 3 + . . . + 10
```

Let's generalize the preceding sequence and denote S(n) as the sum of the first n positive integers, which leads to the following relationship:

```
S(1) = 1
S(n) = S(n-1) + n for n > 1
```

With the preceding observations in mind, the next section contains code samples for calculating the sum of the first n positive integers using an iterative approach and then with recursion.

ARITHMETIC SERIES

This section shows you how to calculate the sum of a set of positive integers, such as the numbers from 1 to n inclusive. The first algorithm uses an iterative approach and the second algorithm uses recursion.

Before delving into the code samples, there is a simple way to calculate the closed form sum of the integers from 1 to n inclusive, which we will denote as S. Then there are two

ways to calculate S, as shown here:

```
S = 1 + 2     + 3     + . . . + (n-1) + n
S = n + (n-1) + (n-2) + . . . + 2     + 1
```

There are `n` columns, and each column has the sum equal to `(n+1)`, and therefore the sum of the right-side of the equals sign is `n*(n+1)`. Since the left-side of the equals sign has the sum `2*S`, we have the following result:

```
2*S = n*(n+1)
```

Divide both sides by 2 and we get the well-known formula for the arithmetic sum of the first `n` positive integers:

```
S = n*(n+1)/2
```

Incidentally, the preceding formula was derived by a young student who was bored with performing the calculation manually: that student was Karl F Gauss (in third grade).

Calculating Arithmetic Series (Iterative)

Listing 6.1 displays the contents of the `arith_sum.py` that illustrates how to calculate the sum of the numbers from 1 to `n` inclusive using an iterative approach.

Listing 6.1: arith_sum.py

```
def arith_sum(n):
  sum = 0
  for i in range(1,n+1):
    sum += i
  return sum

max = 20
for j in range(2,max+1):
  sum = arith_sum(j)
  print("sum from 1 to",j,"=",sum)
```

Listing 6.1 starts with the function `arith_sum()` that contains a loop that literately adds the numbers from 1 to `n`. The next portion of Listing 6.1 also contains a loop that iterates through the numbers from 2 to 20 inclusive, and then invokes `arith_sum()` with each value of the loop variable to calculate the sum of the integers from 1 to that value. Launch the code in Listing 6.1 and you will see the following output:

```
sum from 1 to 2 = 3
sum from 1 to 3 = 6
sum from 1 to 4 = 10
sum from 1 to 5 = 15
sum from 1 to 6 = 21
sum from 1 to 7 = 28
sum from 1 to 8 = 36
sum from 1 to 9 = 45
sum from 1 to 10 = 55
sum from 1 to 11 = 66
sum from 1 to 12 = 78
sum from 1 to 13 = 91
sum from 1 to 14 = 105
sum from 1 to 15 = 120
sum from 1 to 16 = 136
sum from 1 to 17 = 153
sum from 1 to 18 = 171
sum from 1 to 19 = 190
sum from 1 to 20 = 210
```

Modify the code in Listing 6.1 to calculate the sum of the squares, cubes, and fourth powers of the numbers from 1 to `n`, along with your own variations of the code.

Calculating Arithmetic Series (Recursive)

Listing 6.2 displays the contents of the `arith_sum_recursive.py` that illustrates how to calculate the sum of the numbers from 1 to `n` inclusive using a recursion.

Listing 6.2: arith_sum_recursive.py

```
def arith_sum(n):
  if(n == 0):
    return n
  else:
    return n + arith_sum(n-1)

max = 20
for j in range(2,max+1):
  sum = arith_sum(j)
  print("sum from 1 to",j,"=",sum)
```

Listing 6.2 starts with the recursive function `arith_sum()` that uses conditional logic to return `n` if `n` equals the value 0 (which is the terminating case); otherwise the code returns the value of `n` *plus* the value of `arith_sum(n-1)`. Launch the code in Listing 6.2 and you will see the same output as the previous section.

Calculating Partial Arithmetic Series

Listing 6.3 displays the contents of the `arith_partial_sum.py` that illustrates how to calculate the sum of the numbers from `m` to `n` inclusive, where m and `n` are two positive integers such that `m <= n`, using an iterative approach.

Listing 6.3: arith_partial_sum.py

```
def arith_partial_sum(m,n):
  if(m >= n):
    return 0
  else:
    return n*(n+1)/ - m*(m+1)/2

max = 20
for j in range(2,max+1):
  sum = arith_partial_sum(j)
  print("sum from 1 to",j,"=",sum)
```

Listing 6.3 is straightforward: The function `arith_partial_sum()` returns the sum of squares from 1 to `n` *minus* the sum of squares from 1 to `m`. This function is invoked in a loop in the second part of Listing 6.3, which calculates the difference of the sum of squares from 2 to 20. Launch the code in Listing 6.3 and you will see the following output:

```
arithmetic sum from 2 to 2 = 2
arithmetic sum from 2 to 3 = 3
arithmetic sum from 2 to 4 = 7
arithmetic sum from 2 to 5 = 12
arithmetic sum from 2 to 6 = 18
arithmetic sum from 3 to 3 = 3
arithmetic sum from 3 to 4 = 4
arithmetic sum from 3 to 5 = 9
arithmetic sum from 3 to 6 = 15
arithmetic sum from 4 to 4 = 4
arithmetic sum from 4 to 5 = 5
arithmetic sum from 4 to 6 = 11
arithmetic sum from 5 to 5 = 5
arithmetic sum from 5 to 6 = 6
```

Now that you have seen some examples involving arithmetic expressions, let's turn to geometric series, which is the topic of the next section.

GEOMETRIC SERIES

This section shows you how to calculate the geometric series of a set of positive integers, such as the numbers from 1 to `n` inclusive. The first algorithm uses an iterative approach and the second algorithm uses recursion.

Before delving into the code samples, there is a simple way to calculate the closed form sum of the geometric series of integers from 1 to `n` inclusive, where `r` is the ratio of consecutive terms in the geometric series. Let `s` denote the sum, which we can express as follows:

```
S   = 1+ r + r^2 + r^3 + . . . + r^(n-1) + r^n
r*S =    r + r^2 + r^3 + . . . + r^(n-1) + r^n + r^(n+1)
```

Subtract each term in the second row above from the corresponding term in the first row, and we have the following result:

```
S - r*S = 1 - r^(n+1)
```

Factor `s` from both terms on the left side of the preceding equation and we get the following result:

```
S*(1 - r) = 1 - r^(n+1)
```

Divide both sides of the preceding equation by the term `(1-r)` to get the formula for the sum of the geometric series of the first `n` positive integers:

```
S = [1 - r^(n+1)]/(1-r)
```

If `r = 1` then the preceding equation returns an infinite value, which makes sense because `s` is the sum of an infinite number of occurrences of the number 1.

Calculating a Geometric Series (Iterative)

Listing 6.4 displays the contents of the geom_sum.py that illustrates how to calculate the sum of the numbers from 1 to n inclusive using an iterative approach.

Listing 6.4: geom_sum.py

```
def geom_sum(n,ratio):
  partial = 0
  power   = 1
  for i in range(1,n+1):
    partial += power
    power *= ratio
  return partial

max = 10
ratio = 2
for j in range(2,max+1):
  prod = geom_sum(j,ratio)
  print("geometric sum for ratio=",ratio,"from 1 to",j,"=",prod)
```

Listing 6.4 starts with the function geom_sum() that contains a loop that calculates the sum of the powers of the numbers from 1 to n, where the power is the value of the variable ratio. The second part of Listing 6.4 contains a loop that invokes the function geom_sum() with the values 2, 3, . . . , n and a fixed value of 2 for the variable ratio. Launch the code in Listing 6.4 and you will see the following output:

```
geometric sum for ratio= 2 from 1 to 2 = 3
geometric sum for ratio= 2 from 1 to 3 = 7
geometric sum for ratio= 2 from 1 to 4 = 15
geometric sum for ratio= 2 from 1 to 5 = 31
geometric sum for ratio= 2 from 1 to 6 = 63
geometric sum for ratio= 2 from 1 to 7 = 127
geometric sum for ratio= 2 from 1 to 8 = 255
geometric sum for ratio= 2 from 1 to 9 = 511
geometric sum for ratio= 2 from 1 to 10 = 1023
```

Calculating Geometric Series (Recursive)

Listing 6.5 displays the contents of the geom_sum_recursive.py that illustrates how to calculate the sum of the geometric series of the numbers from 1 to n inclusive using recursion. The following code sample uses a technique called *tail recursion*.

Listing 6.5: geom_sum_recursive.py

```
def geom_sum(n,ratio,term,sum):
  if(n == 1): # this is the terminating condition
    return sum
  else:
    term *= ratio
    sum += term
    return geom_sum(n-1,ratio,term,sum)

max = 10
ratio = 2
sum = 1
term = 1

for j in range(2,max+1):
  prod = geom_sum(j,ratio,term,sum)
  print("geometric sum for ratio=",ratio,"from 1 to",j,"=",prod)
```

Listing 6.5 contains the function geom_sum() that takes four parameters: n (which is initially the number of terms that will be added), ratio (which is the exponent 2 in this code sample), term (which is the intermediate quantity that is added to the variable sum), and the variable sum that keeps track of the sum of terms (of the form r^k).

As you can see, the code returns the value 1 when n equals 1; otherwise, the values of term and sum are updated. Now let's look closely at the return statement in the Python function geom_sum() that recursively invokes the geom_sum() function. First of all, notice that n-1 (which is also shown in bold) is specified, whereas the value n (shown in bold) is specified in the function definition.

Next, the other parameters in the function geom_sum() that is invoked in the return statement involve the *same* value for the variable ratio; a newly updated value for the variable sum; and a newly updated value for the variable term.

Thus, we have the following behavior for the parameters of the geom_sum() function:

- n: decreases by 1 during each invocation of geom_sum()
- ratio: constant value during each invocation of geom_sum()
- term: increases during each invocation of geom_sum()
- sum: increases during each invocation of geom_sum()

Here is the key point: The recursive invocation of the function geom_sum() terminates when

n is decreased to the value 1, at which point the variable sum will equal the sum of a geometric series.

This code sample in this section illustrates the concept of *tail recursion*, which is more efficient than regular recursion, and perhaps a little more intuitive as well.

The second part of Listing 6.5 contains a loop that invokes the function `geom_sum()` as the loop iterates from 2 to `max` inclusive. Launch the code in Listing 6.5 and you will see the same output as the previous section.

FACTORIAL VALUES

This section contains three code samples for calculating factorial values: One code samples uses a loop and the other two code samples use recursion.

As a reminder, the *factorial* value of a positive integer `n` is the product of all the numbers from 1 to `n` (inclusive). Hence, we have the following values:

```
Factorial(2)  = 2*1 = 2
Factorial(3)  = 3*2*1 = 6
Factorial(4)  = 4*3*2*1 = 24
Factorial(5)  = 5*4*3*2*1 = 120
Factorial(6)  = 6*5*4*3*2*1 = 720
Factorial(7)  = 7*6*5*4*3*2*1 = 5040
```

If you look at the preceding list of calculations, you can see some interesting relationships among factorial numbers:

```
Factorial(3) = 3 * Factorial(2)
Factorial(4) = 4 * Factorial(3)
Factorial(5) = 5 * Factorial(4)
Factorial(6) = 6 * Factorial(5)
Factorial(7) = 7 * Factorial(6)
```

Based on the preceding observations, it's reasonably intuitive to infer the following relationship for factorial numbers:

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

Although it might not be obvious, the following is also true:

```
Factorial(0) = 1
```

The next section uses the preceding formula in order to calculate the factorial value of various numbers.

Calculating Factorial Values (Iterative)

Listing 6.6 displays the contents of the `Factorial1.py` that illustrates how to calculate factorial numbers using an iterative approach.

Listing 6.6: Factorial1.py

```
def factorial(n):
  prod = 1
  for i in range(1,n+1):
     prod *= i
  return prod

max = 20
for n in range(0,max):
  result = factorial(n)
  print("factorial",n,"=",result)
```

Listing 6.6 starts with the function `factorial()` that contains a loop to multiply the numbers from 1 to `n` and storing the product in the variable prod whose initial value is 1. The second part of Listing 6.6 contains a loop that invokes `factorial()` with the loop variable that ranges from 0 to `max`. Launch the code in Listing 6.6 and you will see the following output:

```
factorial 0 = 1
factorial 1 = 1
factorial 2 = 2
factorial 3 = 6
factorial 4 = 24
factorial 5 = 120
factorial 6 = 720
factorial 7 = 5040
factorial 8 = 40320
factorial 9 = 362880
factorial 10 = 3628800
factorial 11 = 39916800
factorial 12 = 479001600
factorial 13 = 6227020800
factorial 14 = 87178291200
```

```
factorial 15 = 1307674368000
factorial 16 = 20922789888000
factorial 17 = 355687428096000
factorial 18 = 6402373705728000
factorial 19 = 121645100408832000
```

Calculating Factorial Values (Recursive)

Listing 6.7 displays the contents of the `Factorial2.py` that illustrates how to calculate factorial values using recursion. Unlike the earlier recursive example of calculating factorial values, this code sample does not use tail recursion.

Listing 6.7: Factorial2.py

```
def factorial(n):
  if(n <= 1):
    return 1
  else:
    return n * factorial(n-1)

max = 20
for n in range(0,max):
  result = factorial(n)
  print("factorial",n,"=",result)
```

Listing 6.7 starts with the function `factorial()` whose code is recursive (without tail recursion) instead of iterative. Once again, notice that the `n` (shown in bold) in the definition of `factorial()` is decreased by 1 in the `return` statement. Consequently, the recursive invocation of the `factorial()` function will terminate when `n` is decreased to the value 1.

The second portion of Listing 6.7 is identical to the second portion of Listing 6.6. Launch the code in Listing 6.7 and you will see the same output as the preceding example.

Calculating Factorial Values (Tail Recursion)

Listing 6.8 displays the contents of the `Factorial3.py` that illustrates how to modify Listing 6.7 to use tail recursion during the calculation of factorial values.

Listing 6.8: Factorial3.py

```
def factorial(n, prod):
  if(n <= 1):
     return prod
  else:
     return factorial(n-1, n*prod)

max = 20
for n in range(0,max):
  result = factorial(n, 1)
  print("factorial",n,"=",result)
```

Listing 6.8 starts with the recursive function `factorial()` that uses tail recursion, which is somewhat analogous to the tail recursion in Listing 6.5. The second portion of Listing 6.8 is the same as the second portion of Listing 6.5. Launch the code in Listing 6.8 and you will see the same output as the preceding example.

FIBONACCI NUMBERS

Fibonacci numbers are simple yet interesting, and also appear in nature (such as the pattern of sunflower seeds). Here is the definition of the Fibonacci sequence:

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

Note that it's possible to specify different "seed" values for `Fib(0)` and `Fib(1)`, but the values 0 and 1 are the most commonly used values.

Calculating Fibonacci Numbers (Recursive)

Listing 6.9 displays the contents of the `Fibonacci1.py` that illustrates how to calculate Fibonacci numbers using recursion.

Listing 6.9: Fibonacci1.py

```
# very inefficient:
def fibonacci(n):
  if n <= 1:
    return n
  else:
```

```
    return fibonacci(n-2) + fibonacci(n-1)

max=20
for i in range(0,max):
  fib = fibonacci(i)
  print("fibonacci",i,"=",fib)
```

Listing 6.9 starts the the recursive function `fibonacci()` that returns 1 if `n` equals 1. If `n` is greater than 1, the code returns the sum of *two* invocations of `fibonacci()`: the first with the value `n-2` and the second invocation with the value `n-1`.

The second part of Listing 6.9 contains another loop that invokes the function `fibonacci()` with the values of the loop variable that iterates from 0 to `max`. Now launch the code in Listing 6.9 and you will see the following output:

```
fibonacci 0 = 0
fibonacci 1 = 1
fibonacci 2 = 1
fibonacci 3 = 2
fibonacci 4 = 3
fibonacci 5 = 5
fibonacci 6 = 8
fibonacci 7 = 13
fibonacci 8 = 21
fibonacci 9 = 34
fibonacci 10 = 55
fibonacci 11 = 89
fibonacci 12 = 144
fibonacci 13 = 233
fibonacci 14 = 377
fibonacci 15 = 610
fibonacci 16 = 987
fibonacci 17 = 1597
fibonacci 18 = 2584
fibonacci 19 = 4181
```

Calculating Fibonacci Numbers (Iterative)

Listing 6.10 displays the contents of `Fibonacci2.py` that illustrates how to calculate Fibonacci numbers using an iterative solution.

Listing 6.10: Fibonacci2.py

```
import numpy as np

max=20
arr1 = np.zeros(max)
arr1[0] = 0
arr1[1] = 1

for i in range(2,max):
  arr1[i] = arr1[i-1] + arr1[i-2]
  print("fibonacci",i,"=",arr1[i])
```

Listing 6.10 uses an array to store the *intermediate* Fibonacci numbers as they are calculated. Despite the overhead of an array, this code is much more efficient than the code in Listing 6.9 because there is no recursion. Now launch the code in Listing 6.10 and you will see the same output as the previous section.

TASK: REVERSE AN ARRAY OF STRINGS VIA RECURSION

Listing 6.11 displays the contents of the `Python` file `reverse.py` that illustrates how to use recursion in order to reverse a string.

Listing 6.11: reverse.py

```
import numpy as np
def reverser(str):
   if(str == None or len(str) == 0):
      return str
   print("all-but-first chars:",str[1:])
   return reverser(str[1:])+list(str[0])

names = np.array(["Nancy", "Dave", "Dominic"])

for name in names:
  str_list = list(name)
  result = reverser(str_list)
  print("=> Word: ",name," reverse: ",result)
  print()
```

Listing 6.11 starts with the recursive function `reverser()` that invokes itself with a substring that omits the first character, which is appended to the result of invoking `reverser()` recursively, as shown here:

```
return reverser(str[1:])+list(str[0])
```

The second part of Listing 6.11 contains a loop that invokes the `reverser()` method with different strings that belong to an array. Launch the code in Listing 6.11 and you will see the following output:

```
all-but-first chars: ['a', 'n', 'c', 'y']
all-but-first chars: ['n', 'c', 'y']
all-but-first chars: ['c', 'y']
all-but-first chars: ['y']
all-but-first chars: []
=> Word:  Nancy  reverse:  ['y', 'c', 'n', 'a', 'N']

all-but-first chars: ['a', 'v', 'e']
all-but-first chars: ['v', 'e']
all-but-first chars: ['e']
all-but-first chars: []
=> Word:  Dave  reverse:  ['e', 'v', 'a', 'D']

all-but-first chars: ['o', 'm', 'i', 'n', 'i', 'c']
all-but-first chars: ['m', 'i', 'n', 'i', 'c']
all-but-first chars: ['i', 'n', 'i', 'c']
all-but-first chars: ['n', 'i', 'c']
all-but-first chars: ['i', 'c']
all-but-first chars: ['c']
all-but-first chars: []
=> Word:  Dominic  reverse:  ['c', 'i', 'n', 'i', 'm', 'o', 'D']
```

TASK: CHECK FOR BALANCED PARENTHESES

This task involves only round parentheses, and a more challenging task involves checking for balanced parentheses that includes square brackets and curly braces. Here are some example of strings that contain round parentheses:

```
S1 = "()()()"
S2 = "(()()())"
S3 = "()("
S4 = "(())"
S5 = "()()("
```

As you can see, `S1`, `S3`, and `S4` have balanced parentheses, whereas `S2` and `S5` has unbalanced parentheses.

Listing 6.12 displays the contents of the `Python` file `balanced_parentheses.py` that illustrates how to determine whether or not a string consists of balanced parentheses.

Listing 6.12: balanced_parentheses.py

```
import numpy as np

def check_balanced(text):
  counter = 0
  text_len = len(text)

  for i in range(text_len):
    if (text[i] == '('):
      counter += 1
    else:
      if (text[i] == ')'):
        counter -= 1

    if (counter < 0):
      break

  if (counter == 0):
    print("balanced string:",text)
  else:
    print("unbalanced string:",text)
  print()

exprs = np.array(["()()()", "(()()())","()(","(())","()()("])

for str in exprs:
  check_balanced(str)
```

Listing 6.12 starts with the iterative function `check_balanced()` that uses conditional logic to check the value of the current variable in the input string. The code increments the variable counter if the current character is a left parenthesis “(” , and decrements the variable counter if the current character is a right parentheses “)”. The only way for an expression to consist of a balanced set of parentheses is for counter to equal 0 when the loop has finished execution.

The second part of Listing 6.12 contains a loop that invokes the function `check_balanced()` with different strings that are part of an array of strings. Launch the code in Listing 6.12 and you will see the following output:

```
exprs = np.array(["()()()", "(()()())","()(","(())","()()("])
balanced string: ()()()

balanced string: (()()())

unbalanced string: ()(

balanced string: (())

unbalanced string: ()()(
```

TASK: CALCULATE THE NUMBER OF DIGITS

Listing 6.13 displays the contents of the `Python` file `count_digits.py` that illustrates how to calculate the number of digits in positive integers.

Listing 6.13: count_digits.py

```
import numpy as np
def count_digits(num, result):
  if( num == 0 ):
    return result
  else:
    #print("new result:",result+1)
    #print("new number:",int(num/10))
    return count_digits(int(num/10), result+1)

numbers = np.array([1234, 767, 1234321, 101])

for num in numbers:
  result = count_digits(num, 0)
  print("Digits in ",num," = ",result)
```

Listing 6.13 starts with the `Python` function `count_digits()` that recursively invokes itself with the term `int(num/10)`, where num is the input parameter. Moreover, each invocation of `count_digits()` increments the value of the parameter `result`. Eventually `num` will be equal to 0 (the terminating condition), at which point the value of `result` is returned. If the logic of this code is not clear to you, try tracing through the code with the numbers 5, 25, 150, and you will see that the function `count_digits()` returns the values 1, 2, and 3, respectively. Launch the code in Listing 6.13 and you will see the following output:

```
Digits in  1234  =  4
Digits in  767  =  3
Digits in  1234321  =  7
Digits in  101  =  3
```

TASK: DETERMINE IF A POSITIVE INTEGER IS PRIME

Listing 6.14 displays the contents of the `Python` file `check_prime.py` that uses iteration to calculate the number of digits in positive integers.

Listing 6.14: check_prime.py

```
import numpy as np
PRIME = 1
COMPOSITE = 0

def is_prime(num):
  div = 2

  while(div*div < num):
   if( num % div != 0):
      div += 1
   else:
      return COMPOSITE
  return PRIME

upperBound = 20

for num in range(2, upperBound):
  result = is_prime(num)
  if(result == True):
    print(num,": is prime")
  else:
    print(num,": is not prime")
```

Listing 6.14 starts with the `Python` function `is_prime()` that contains a loop that checks whether or not any integer in the range of 2 to `sqrt(num)` divides the parameter `num`, and then returns the appropriate result.

The second portion of Listing 6.14 contains a loop that iterates through the numbers from 2 to `upperBound` (which has the value 20) to determine which numbers are prime. Launch

the code in Listing 6.14 and you will see the following output:

```
2 : is prime
3 : is prime
4 : is not prime
5 : is prime
6 : is not prime
7 : is prime
8 : is not prime
9 : is not prime
10 : is not prime
11 : is prime
12 : is not prime
13 : is prime
14 : is not prime
15 : is not prime
16 : is not prime
17 : is prime
18 : is not prime
19 : is prime
```

TASK: FIND THE PRIME FACTORIZATION OF A POSITIVE INTEGER

Listing 6.15 displays the contents of the `Python` file `prime_divisors.py` that illustrates how to find the prime divisors of a positive integer.

Listing 6.15: prime_divisors.py

```
import numpy as np
PRIME = 1
COMPOSITE = 0

def is_prime(num):
  div = 2

  while(div < num):
    if( num % div != 0):
      div += 1
    else:
      return COMPOSITE

  #print("found prime:",num)
  return PRIME

def find_prime_divisors(num):
  div = 2
  prime_divisors = ""

  while(div <= num):
    prime = is_prime(div)

    if(prime == True):
      #print("=> prime number:",div)
      if(num % div == 0):
        prime_divisors += " "+str(div)
        num = int(num/div)
     else:
        div += 1
    else:
      div += 1

  return prime_divisors

upperBound = 20

for num in range(4, upperBound):
  result = find_prime_divisors(num)
  print("Prime divisors of ",num,":",result)
```

Listing 6.15 starts with the `Python` function `is_prime()` from Listing 6.14 that determines whether or not a positive integer is a prime number. Next, the `Python` function `find_prime_divisors()` contains a loop that iterates through the integers from 2 to `num` that checks which of those numbers is a prime number.

When a prime number is found, the code checks if that prime number is also a divisor of `num`: If so, that prime divisors is appended to the string `prime_divisors` . The final portion of Listing 6.15 returns the string `prime_divisors` that contains the prime factorization of the parameter `num`. Now launch the code in Listing 6.15 and you will see the following output:

```
Prime divisors of  2 :  2
Prime divisors of  4 :  2 2
Prime divisors of  5 :  5
Prime divisors of  6 :  2 3
Prime divisors of  7 :  7
Prime divisors of  8 :  2 2 2
Prime divisors of  9 :  3 3
```

```
Prime divisors of  10 :  2 5
Prime divisors of  11 :  11
Prime divisors of  12 :  2 2 3
Prime divisors of  13 :  13
Prime divisors of  14 :  2 7
Prime divisors of  15 :  3 5
Prime divisors of  16 :  2 2 2 2
Prime divisors of  17 :  17
Prime divisors of  18 :  2 3 3
Prime divisors of  19 :  19
```

TASK: GOLDBACH'S CONJECTURE

Goldbach's conjecture states that every even number greater than 4 can be expressed as the sum of two odd prime numbers: Despite its simplicity, this conjecture has never been proven or disproven. However, we can write `Python` code to check whether or not it's true for some small even numbers.

Listing 6.16 displays the contents of the `Python` file `goldbach_conjecture.py` that illustrates how to determine a pair of prime numbers whose sum equals a given even number.

Listing 6.16: goldbach_conjecture.py

```
import numpy as np
PRIME = 1
COMPOSITE = 0

def prime(num):
  div = 2

  while(div < num):
   if( num % div != 0):
      div += 1
   else:
      return COMPOSITE
  return PRIME

def find_prime_factors(even_num):
  for num in range(3, int(even_num/2)):
    if(prime(num) == 1):
      if(prime(even_num-num) == 1):
        print(even_num , " = " , num , "+" , (even_num-num))

upperBound = 30

for num in range(4, upperBound):
  find_prime_factors(num)
```

Listing 6.16 also starts with the function `prime()` that determines whether or not the parameter `num` is a prime number. Next, the function `find_prime_factors()` contains a loop with the loop variable `num`: This variable iterates from 3 to half the value of the parameter `even_num`. If `num` is a prime number, then the conditional logic in Listing 6.16 invokes `prime()` with the number `even_num-num`.

If both `num` and `even_num` are prime, then they are a solution to Goldbach's conjecture because the sum of these two numbers equals the parameter `even_num`. Now launch the code in Listing 6.16 and you will see the following output:

```
8  =  3 + 5
10  =  3 + 7
12  =  5 + 7
14  =  3 + 11
16  =  3 + 13
16  =  5 + 11
18  =  5 + 13
18  =  7 + 11
20  =  3 + 17
20  =  7 + 13
22  =  3 + 19
22  =  5 + 17
24  =  5 + 19
24  =  7 + 17
24  =  11 + 13
26  =  3 + 23
26  =  7 + 19
28  =  5 + 23
28  =  11 + 17
```

As you can see from the preceding output, the numbers 16, 18, 20, 22, 26, and 28 have two solutions to Goldbach's conjecture, and the number 24 has three such solutions.

TASK: CALCULATE THE GCD (GREATEST COMMON DIVISOR)

Listing 6.17 displays the contents of the `Python` file `gcd.py`, which is the first of two solutions for calculating the GCD of two positive integers (both solutions rely on Euclid's algorithm).

Listing 6.17: gcd.py

```
import numpy as np
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)

arr1 = np.array([24, 36, 50, 100, 200])
arr2 = np.array([10, 18, 11,  64, 120])

for i in range(0,len(arr1)):
  num1 = arr1[i]
  num2 = arr2[i]
  result = gcd(num1,num2)
  print("The GCD of",num1,"and",num2,"=",result)
```

Listing 6.17 starts with the `Python` function `gcd()` that takes two parameters and repeatedly subtracts the smaller from the larger, and simultaneously invoking itself recursively. Eventually `num1 % num2` equals zero, at which point the GCD equals `num2`, which is the value that is returned.

The second portion of Listing 6.17 contains a loop that iterates through the values of two arrays of positive integers; during each iteration, the function `gcd()` is invoked with a pair of corresponding numbers from the two arrays. Now launch the code in Listing 6.17 and you will see the following output:

```
The GCD of 24 and 10 = 2
The GCD of 36 and 18 = 18
The GCD of 50 and 11 = 1
The GCD of 100 and 64 = 4
The GCD of 200 and 120 = 40
```

Listing 6.18 displays the contents of `simple_gcd.py` that is a more concise way to compute the GCD of two positive integers (and also uses recursion).

Listing 6.18: simple_gcd.py

```
import numpy as np

def gcd(x1, x2):
  if not x2:
    return x1
  return gcd(x2, x1 % x2)

arr1 = np.array([10, 24, 50, 17, 100])
arr2 = np.array([24, 10, 15, 17, 1250])

for idx in range(0,len(arr1)):
  num1 = arr1[idx]
  num2 = arr2[idx]
  result = gcd(num1,num2)
  print("gcd of",num1,"and",num2,"=",result)
```

Listing 6.18 is a more compact implementation of Euclid's algorithm that achieves the same result as Listing 6.17. If the logic is unclear, review the details of Listing 6.17 to convince yourself that the logic in both code samples is the same. Launch the code in Listing 6.18 and you will see the following output:

```
gcd of 10 and 24 = 2
gcd of 24 and 10 = 2
gcd of 50 and 15 = 5
gcd of 17 and 17 = 17
gcd of 100 and 1250 = 50
```

Listing 6.18 displays the contents of `map_gcd.py` that illustrates how to invoke a recursive function from the `map()` function that is applied to a list of numbers. Read the relevant portion of Chapter 3 if you need to review how to use the `Python map()` function.

Listing 6.18: map_gcd.py

```
import numpy as np

def gcd(x,y):
    if x < y:
      x,y = y,x
    while( y != 0 ):
      x, y = y, x % y
    return x

x_vals = [24, 15, 18, 243, 37]
y_vals = [10,  3, 12,  81, 74]
```

```
print("x values:",x_vals)
print("y values:",y_vals)

gcd_vals = map(gcd, x_vals, y_vals)
gcds = list(gcd_vals)

print("GCD values:")
print(gcds)
```

Listing 6.18 defines the Python function gcd() that calculate the GCD of two positive integers x and y. This function switches x and y if x is less than y, followed by a while loop that repeatedly performs the following operation until y equals 0:

```
x, y = y, x % y
```

When y reaches the value 0, this function returns the value of x. The next portion of Listing 6.18 initializes the lists x_vals and y_vals with a set of integers and displays their contents.

The next code snippet uses the map() method to invoke the gcd() function with pair-wise values from the lists x_vals and y_vals, the result of which is assigned to the variable gcd_vals. Next, the variable gcds is initialized with the result of converting the contents of gcd_vals to a Python list, after which the GCD values are displayed. Launch the code in Listing 6.18 and you will see the following output:

```
x values: [24, 15, 18, 243, 37]
y values: [10, 3, 12, 81, 74]
GCD values:
[2, 3, 6, 81, 37]
```

Now that we can calculate the GCD of two positive integers, we can use this code to easily calculate the LCM (lowest common multiple) of two positive integers, as discussed in the next section.

TASK: CALCULATE THE LCM (LOWEST COMMON MULTIPLE)

Listing 6.19 displays the contents of the Python file simple_lcm.py that illustrates how to calculate the LCM of two positive integers.

Listing 6.19: simple_lcm.py

```
import numpy as np
def gcd(x1, x2):
  if not x2:
    return x1
  return gcd(x2, x1 % x2)

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

  return lcm1

arr1 = np.array([24, 36, 50, 100, 200])
arr2 = np.array([10, 18, 11,  64, 120])

for i in range(0,len(arr1)):
  num1 = arr1[i]
  num2 = arr2[i]
  result = lcm(num1,num2)
  print("The LCM of",num1,"and",num2,"=",result)
```

Listing 6.19 contains the function gcd() to calculate the GCD of two positive integers. The next function lcm() calculates the LCM of two numbers num1 and num2 by making the following observation:

```
LCM(num1, num2) = num1/GCD(num1,num2)*num2/GCD(num1,num2)
```

The final portion of Listing 6.19 contains a loop that iterates through two arrays of positive integers to calculate the LCM of pairs of integers. Launch the code in Listing 6.19 and you will see the following output:

```
The LCM of 24 and 10 = 60.0
The LCM of 36 and 18 = 2.0
The LCM of 50 and 11 = 550.0
The LCM of 100 and 64 = 400.0
The LCM of 200 and 120 = 15.0
```

This concludes the portion of the chapter regarding recursion. The next section introduces you to combinatorics (a well-known branch of mathematics), along with some code samples for calculating combinatorial values and the number of permutations of

objects.

WHAT IS COMBINATORICS?

In simple terms, combinatorics involves finding formulas for counting the number of objects in a set. For example, how many different ways can five books can be ordered (i.e., displayed) on a book shelf? The answer involves permutations, which in turn is a factorial value; in this case, the answer is 5! = 120.

As a second example, suppose how many different ways can you select three books from a shelf that contains five books? The answer to this question involves combinations. Keep in mind that if you select three books labeled A, B, and C, then any permutation of these three books is considered the same (the set A, B, and C and the set B, A, and C are considered the same selection).

As a third example, how many 5-digit binary numbers contain exactly three 1 values? The answer to this question also involves calculating a combinatorial value. The answer is `C(5,3) = 5!/[3! * 2!] = 10`, provided that we allow for leading zeroes. In fact, this is also the answer to the preceding question about selecting different subsets of books.

You can generalize the previous question by asking how many 4-digit, 5-digit, and 6-digit numbers contain exactly three 1s? The answer is the sum of these values (provided that leading zeroes are permitted):

```
C(4,3) + C(5,3) + C(6,3) = 4 + 10 + 20 = 34
```

Working With Permutations

Consider the following task: Given six books, how many ways can you display them side-by-side? The possibilities are listed here:

position #1: six choices

position #2: five choices

position #3: four choices

position #4: three choices

position #5: two choices

position #6: one choice

The answer is 6x5x4x3x2x1 = 6! = 720. In general, if you have n books, there are n! different ways that you can order them (i.e., display them side-by-side).

Working With Combinations

Let’s look at a slightly different question: How many ways can you select three books from those six books? Here’s the first approximation:

position #1: six choices

position #2: five choices

position #3: four choices

Since the number of books in any position is independent of the other positions, the first answer might be 6x5x4 = 120. However, this answer is incorrect because it includes different orderings of three books, but the sequence of books (A,B,C) is the same as (B,A,C) and every other recording of the letters A, B, and C.

As a concrete example, suppose that the books are labeled sequentially as book #1, book #2, . . . , book #6, and suppose that you also select book #1, book #2, and book #3 (i.e., the first three books). Here list a list of all the different orderings of those three books:

```
123
132
213
231
312
321
```

The number of different orderings of three books is 3x2x1 = 3! = 6. However, from the standpoint of purely selecting three books, we must treat all six orderings as the same. Hence the correct answer is 6x5x4/[3x2x1] = 120/3! = 120/6 = 20.

Now let’s multiply the numerator and the denominator by 3x2x1, which gives us this number: 6x5x4x3x2x1/[3x2x1 * 3x2x1] = 6!/[3! * 3!]

If we perform the preceding task of selecting three books from eight books instead of six books, we get this result:

```
8x7x6/[3x2x1] = 8x7x6x5x4x3x2x1/[3x2x1 * 5x4x3x2x1] = 8!/[3! * 5!]
```

Suppose you select 12 books from a set of 30 books. The number of ways that this can be done is shown here:

```
30x29x28x...x19/[12x11x...x2x1]
= 30x29x28x...x19x18x17x16x...x2x1/[12x11x...x2x1 * 18x17x16x...x2x1]
= 30!/[12! * 18!]
```

The general formula for calculating the number of ways to select `k` books from `n` books is `n!/[k! * (n-k)!]`, which is denoted by the term `C(n,k)`. Incidentally, if we replace k by n-k in the preceding formula we get this result:

```
n!/[(n-k)! * (n-(n-k))!] = n!/[(n-k)! * k)!] = C(n,k)
```

Notice that the left-side of the preceding snippet equals `C(n,n-k)`, and therefore we have shown that `C(n,n-k) = C(n,k)`

TASK: CALCULATE THE SUM OF BINOMIAL COEFFICIENTS

Recall from the previous section that the value of the binomial coefficient `C(n,k)` can be computed as follows:

```
C(n,k) =  n!/[k! * (n-k)!]
```

If n is a positive integer, the following is true (details are in the next section):

```
2**n = C(n,0)+C(n,1)+C(n,2)+. . . +C(n,n-1)+C(n,n)
```

Listing 6.20 displays the contents of the `Python` file `sum_binomial.py` that calculates the sum of a set of binomial coefficients.

Listing 6.20: sum_binomial.py

```
import numpy as np

def factorial(num):
  fact = 1
  for i in range(0,num):
    fact *= (i+1)
  return int(fact)

def binom_coefficient(n,k):
  global fact_values
  coeff = fact_values[n]/[fact_values[k] * fact_values[n-k]]
  #print("calculated coeff:",coeff)
  return int(coeff)

def sum_binomials(exp):
  binomials = np.array([]).astype(int)
  coeff_sum = 0
  for num in range(0,exp+1):
    coeff_value = binom_coefficient(exp,num)
    #print("n:",exp-2,"k:",num,"found coeff_value:",coeff_value)
    coeff_sum += coeff_value

  print("sum of binomial coefficients for",exp,"=",int(coeff_sum))

exponent = 12
# populate an array with factorial values:
fact_values = np.array([]).astype(int)
for j in range(0,exponent):
  fact = factorial(j)
  fact_values = np.append(fact_values,fact)

for exp in range(1,exponent-1):
  sum_binomials(exp)
```

Listing 6.20 starts with the function `factorial()` to calculate the factorial value of a positive integer (whose code you saw earlier in this chapter). Next, the `Python` function `binom_coefficient()` calculates the binomial of two integers whose formula was derived in a previous section.

The third function is `sum_binomials()` that calculate the sum of a range of binomial values by invoking the function `binom_coefficient()`, where the latter invokes the function `factorial()`. Now launch the code in Listing 6.20 and you will see the following output:

```
sum of binomial coefficients for 1 = 2
sum of binomial coefficients for 2 = 4
sum of binomial coefficients for 3 = 8
sum of binomial coefficients for 4 = 16
sum of binomial coefficients for 5 = 32
```

```
sum of binomial coefficients for 6 = 64
sum of binomial coefficients for 7 = 128
sum of binomial coefficients for 8 = 256
sum of binomial coefficients for 9 = 512
sum of binomial coefficients for 10 = 1024
```

THE NUMBER OF SUBSETS OF A FINITE SET

In the preceding section, if we allow `k` to vary from 0 to `n` inclusive, then we are effectively looking at all possible subsets of a set of `n` elements, and the number of such sets equals `2^n`. We can derive the preceding result in two ways.

Solution #1:

The first way is the shortest explanation (and might seem like clever hand waving) that involves visualizing a row of n books. In order to find every possible subset of those n books, we need only consider that there are two actions for the first position: Either the book is selected or it is not selected.

Similarly, there are two actions for the second position: either the second book is selected or it is not selected. In fact, for every book in the set of n books there are the same two choices. Keeping in mind that the selection (or not) of a book in a given position is independent of the selection of the books in every other position, the number of possible choices equals `2x2x...x2 (n times) = 2^n.`

Solution #2:

Recall the following formulas from algebra:

```
(x+y)^2 = x^2 + 2*x*y + y^2
        = C(2,0)*x^2 + C(2,1)*x*y + C(2,2)*y^2


(x+y)^3 = x^3 + 4*x^2*y + 6*x^x*y^2 + 4*x*y^2 + y^3
= C(3,0)*x^3 + C(3,0)*x^2*y + C(3,0)*x^x*y^2 + C(3,0)*x*y^2 + C(3,0)*y^3
```

In general, we have the following formula:

```
              n
(x+y)^n =  SUM C(n,k)*x^k*y^(n-k)
           k=0
```

Now set `x=y=1` in the preceding formula and we get the following result:

```
         n
2^n =  SUM C(n,k)
       k=0
```

The right-side of the preceding formula is the sum of the number of all possible subsets of a set of n elements, which the left side shows is equal to 2^n.

SUMMARY

This chapter started with an introduction to recursion, along with various code samples that involve recursion, such as calculating factorial values, Fibonacci numbers, the sum of an arithmetic series, the sum of a geometric series, the GCD of a pair of positive integers, and the LCM of a pair of positive integers.

Finally, you learned about concepts in combinatorics, and how to derive the formula for the number of permutations and the number of combinations of sets of objects.

INTRODUCTION TO PANDAS

This appendix introduces you to Pandas and provides code samples that illustrate some of its useful features. If you are familiar with these topics, skim through the material and peruse the code samples, just in case they contain information that is new to you.

The first part of this appendix contains a brief introduction to Pandas. This section contains code samples that illustrate some features of data frames and a brief discussion of series, which are two of the main features of Pandas.

The second part of this appendix discusses various types of data frames that you can create, such as numeric and Boolean data frames. In addition, we discuss examples of creating data frames with NumPy functions and random numbers.

NOTE

Several code samples in this chapter reference the NumPy library for working with arrays and generating random numbers, which you can learn from online articles.

WHAT IS PANDAS?

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

```
pip3 install pandas
```

In many ways, the semantics of the APIs in the Pandas library are similar to a spreadsheet, along with support for xsl, xml, html, and csv file types. Pandas provides a data type called a data frame (similar to a Python dictionary) with an extremely powerful functionality.

Pandas data frames support a variety of input types, such as ndarray, list, dict, or series.

The data type series is another mechanism for managing data. In addition to performing an online search for more details regarding Series, the following article contains a good introduction:

https://towardsdatascience.com/20-examples-to-master-pandas-series-bc4c68200324

Pandas Options and Settings

You can change the default values of environment variables, an example of which is shown below:

```
import pandas as pd

display_settings = {
    'max_columns': 8,
    'expand_frame_repr': True,  # Wrap to multiple pages
    'max_rows': 20,
    'precision': 3,
    'show_dimensions': True
}

for op, value in display_settings.items():
pd.set_option("display.{}".format(op), value)
```

Include the preceding code block in your own code if you want Pandas to display a maximum of 20 rows and 8 columns, and floating point numbers displayed with three decimal places. Set expand_frame_rep to True if you want the output to "wrap around" to multiple pages. The preceding for loop iterates through display_settings and sets the options equal to their corresponding values.

In addition, the following code snippet displays all Pandas options and their current values in your code:

```
print(pd.describe_option())
```

There are various other operations that you can perform with options and their values (such as the pd.reset() method for resetting values), as described in the Pandas user guide:

https://pandas.pydata.org/pandas-docs/stable/user_guide/options.html

Pandas Data Frames

In simplified terms, a Pandas data frame is a two-dimensional data structure, and it's convenient to think of the data structure in terms of rows and columns. Data frames can be labeled (rows as well as columns), and the columns can contain different data types. The source of the dataset for a Pandas data frame can be a data file, a database table, and a Web service. The data frame features include

- data frame methods
- data frame statistics
- grouping, pivoting, and reshaping
- handle missing data
- join data frames

The code samples in this appendix show you almost all the features in the preceding list.

Data Frames 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 (and some might be optional). All of the following steps can be performed with a Pandas data frame:

- Read data into a data frame
- Display top of data frame
- Display column data types
- Display 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

This appendix contains sections that illustrate how to perform many of the steps in the preceding list.

Alternatives to Pandas

Before delving into the code samples, there are alternatives to Pandas that offer very useful features, some of which are shown in the following list:

- PySpark (for large datasets)
- Dask (for distributed processing)
- Modin (faster performance)
- Datatable (R data.table for Python)

The inclusion of these alternatives is not intended to diminish Pandas. Indeed, you might not need any of the functionality in the preceding list. However, you might need such functionality in the future, so it's worthwhile for you to know about these alternatives now (and there may be even more powerful alternatives at some point in the future).

A PANDAS DATA FRAME WITH A NUMPY EXAMPLE

Listing A.1 shows the content of pandas_df.py that illustrates how to define several data frames and display their contents.

Listing A.1: pandas_df.py

```
import pandas as pd
import numpy as np

myvector1 = np.array([1,2,3,4,5])
```

```
print("myvector1:")
print(myvector1)
print()

mydf1 = pd.Data frame(myvector1)
print("mydf1:")
print(mydf1)
print()

myvector2 = np.array([i for i in range(1,6)])
print("myvector2:")
print(myvector2)
print()

mydf2 = pd.Data frame(myvector2)
print("mydf2:")
print(mydf2)
print()

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

mydf3 = pd.Data frame(myarray)
print("mydf3:")
print(mydf3)
print()
```

Listing A.1 starts with standard `import` statements for `Pandas` and `NumPy`, followed by the definition of two one-dimensional `NumPy` arrays and a two-dimensional `NumPy` array. Each `NumPy` variable is followed by a corresponding `Pandas` data frame (`mydf1`, `mydf2`, and `mydf3`). Launch the code in Listing A.1 to see the following output, and you can compare the `NumPy` arrays with the `Pandas` data frames:

```
myvector1:
[1 2 3 4 5]

mydf1:
   0
0  1
1  2
2  3
3  4
4  5

myvector2:
[1 2 3 4 5]

mydf2:
   0
0  1
1  2
2  3
3  4
4  5

myarray:
[[  10   30   20]
[  50   40   60]
[1000 2000 3000]]

mydf3:
      0     1     2
0    10    30    20
1    50    40    60
2  1000  2000  3000
```

By contrast, the following code block illustrates how to define two `Pandas Series` that are part of the definition of a `Pandas` data frame:

```
names = pd.Series(['SF', 'San Jose', 'Sacramento'])
sizes = pd.Series([852469, 1015785, 485199])
df = pd.Data frame({ 'Cities': names, 'Size': sizes })
print(df)
```

Create a `Python` file with the preceding code (along with the required `import` statement) and when you launch that code, you will see the following output:

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

DESCRIBING A PANDAS DATA FRAME

Listing A.2 shows the content of `pandas_df_describe.py`, which illustrates how to define a `Pandas` data frame that contains a 3×3 `NumPy` array of integer values, where the rows and columns of the data frame are labeled. Other aspects of the data frame are also displayed.

Listing A.2: pandas_df_describe.py

```
import numpy as np
import pandas as pd

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

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

mydf = pd.Data frame(myarray, index=rownames, columns=colnames)
print("contents of df:")
print(mydf)
print()

print("contents of January:")
print(mydf['January'])
print()

print("Number of Rows:")
print(mydf.shape[0])
print()

print("Number of Columns:")
print(mydf.shape[1])
print()

print("Number of Rows and Columns:")
print(mydf.shape)
print()

print("Column Names:")
print(mydf.columns)
print()

print("Column types:")
print(mydf.dtypes)
print()

print("Description:")
print(mydf.describe())
print()
```

Listing A.2 starts with two standard `import` statements followed by the variable `myarray`, which is a 3×3 `NumPy` array of numbers. The variables `rownames` and `colnames` provide names for the rows and columns, respectively, of the `Pandas` data frame `mydf`, which is initialized as a `Pandas` data frame with the specified data source (i.e., `myarray`).

The first portion of the following output requires a single `print()` statement (that simply displays the contents of `mydf`). The second portion of the output is generated by invoking the `describe()` method that is available for any `Pandas` data frame. The `describe()` method is useful: You will see various statistical quantities, such as the mean, standard deviation minimum, and maximum performed by *columns* (not rows), along with values for the 25th, 50th, and 75th percentiles. The output of Listing A.2 is here:

```
contents of df:
         January  February  March
apples        10        30     20
oranges       50        40     60
beer        1000      2000   3000

contents of January:
apples       10
oranges      50
beer       1000
Name: January, dtype: int64

Number of Rows:
3

Number of Columns:
3

Number of Rows and Columns:
(3, 3)

Column Names:
Index(['January', 'February', 'March'], dtype='object')

Column types:
```

```
January     int64
February    int64
March       int64
dtype: object

Description:
           January     February        March
count     3.000000     3.000000     3.000000
mean    353.333333   690.000000  1026.666667
std     560.386771  1134.504297  1709.073823
min      10.000000    30.000000    20.000000
25%      30.000000    35.000000    40.000000
50%      50.000000    40.000000    60.000000
75%     525.000000  1020.000000  1530.000000
max    1000.000000  2000.000000  3000.000000
```

PANDAS BOOLEAN DATA FRAMES

Pandas supports Boolean operations on data frames, such as the logical OR, the logical AND, and the logical negation of a pair of Data frames. Listing A.3 shows the content of pandas_boolean_df.py that illustrates how to define a Pandas data frame whose rows and columns are Boolean values.

Listing A.3: pandas_boolean_df.py

```
import pandas as pd

df1 = pd.Data frame({'a': [1, 0, 1], 'b': [0, 1, 1] }, dtype=bool)
df2 = pd.Data frame({'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 A.3 initializes the data frames df1 and df2, and then computes df1 & df2, df1 | df2, and 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 A.3 is as follows:

```
df1 & df2:
       a      b
0  False  False
1  False   True
2   True  False
df1 | df2:
      a     b
0  True  True
1  True  True
2  True  True
df1 ^ df2:
       a      b
0   True   True
1   True  False
2  False   True
```

Transposing a Pandas Data Frame

The T attribute (as well as the transpose function) enables you to generate the transpose of a Pandas data frame, similar to the NumPy ndarray. The transpose operation switches rows to columns and columns to rows. For example, the following code snippet defines a Pandas data frame df1 and then displays the transpose of df1:

```
df1 = pd.Data frame({'a': [1, 0, 1], 'b': [0, 1, 1] }, dtype=int)

print("df1.T:")
print(df1.T)
```

The output of the preceding code snippet is here:

```
df1.T:
   0  1  2
a  1  0  1
b  0  1  1
```

The following code snippet defines Pandas data frames df1 and df2 and then displays their sum:

```
df1 = pd.Data frame({'a' : [1, 0, 1], 'b' : [0, 1, 1] }, dtype=int)
df2 = pd.Data frame({'a' : [3, 3, 3], 'b' : [5, 5, 5] }, dtype=int)
```

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

The output is here:

```
df1 + df2:
   a  b
0  4  5
1  3  6
2  4  6
```

PANDAS DATA FRAMES AND RANDOM NUMBERS

Listing A.4 shows the content of `pandas_random_df.py` that illustrates how to create a `Pandas` data frame with random integers.

Listing A.4: pandas_random_df.py

```
import pandas as pd
import numpy as np

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

print("Contents of data frame:")
print(df)
```

Listing A.4 defines the `Pandas` data frame `df` that consists of five rows and two 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 A.4 is here:

```
a    b
0      1.0  2.0
1      1.0  1.0
2      4.0  3.0
3      3.0  1.0
4      1.0  2.0
sum   10.0  9.0
mean   2.0  1.8
```

Listing A.5 shows the content of `pandas_combine_df.py` that illustrates how to combine `Pandas` data frames.

Listing A.5: pandas_combine_df.py

```
import pandas as pd
import numpy as np

df = pd.Data frame({'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 A.5 defines the `Pandas` data frame `df` that consists of five rows and two columns (labeled "`foo1`" and "`foo2`") of random real numbers between 0 and 5. The next portion of Listing A.5 shows the content of `df` and `foo1`. The output of Listing A.5 is as follows:

```
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
```

READING CSV FILES IN PANDAS

Pandas provides the read_csv() method for reading the contents of CSV files. For example, Listing A.6 shows the contents of sometext.csv that contain labeled data (spam or ham), and Listing A.7 shows the contents of read_csv_file.py that illustrate how to read the contents of a CSV file.

Listing A.6: sometext.csv

```
type    text
ham     I'm telling the truth
spam    What a deal such a deal!
spam    Free vacation for your family
ham     Thank you for your help
spam    Spring break next week!
ham     I received the documents
spam    One million dollars for you
ham     My wife got covid19
spam    You might have won the prize
ham     Everyone is in good health
```

Listing A.7: read_csv_file.py

```
import pandas as pd
import numpy as np

df = pd.read_csv('sometext.csv', delimiter='\t')

print("=> First five rows:")
print(df.head(5))
```

Listing A.7 reads the contents of sometext.csv, whose columns are separated by a tab ("\t") delimiter. Launch the code in Listing A.7 to see the following output:

```
=> First five rows:
   type                          text
0   ham          I'm telling the truth
1  spam      What a deal such a deal!
2  spam  Free vacation for your family
3   ham        Thank you for your help
4  spam        Spring break next week!
```

The default value for the head() method is 5, but you can display the first n rows of a data frame df with the code snippet df.head(n).

Specifying a Separator and Column Sets in Text Files

The previous section showed you how to use the delimiter attribute to specify the delimiter in a text file. You can also use the sep parameter specifies a different separator. In addition, you can assign the names parameter the column names in the data that you want to read. An example of using delimiter and sep is here:

```
df2 = pd.read_csv("data.csv",sep="|",
                  names=["Name","Surname","Height","Weight"])
```

Pandas also provides the read_table() method for reading the contents of CSV files, which uses the same syntax as the read_csv() method.

Specifying an Index in Text Files

Suppose that you know that a particular column in a text file contains the index value for the rows in the text file. For example, a text file that contains the data in a relational table would typically contain an index column.

Fortunately, Pandas allows you to specify the kth column as the index in a text file, as shown here:

```
df = pd.read_csv('myfile.csv', index_col=k)
```

THE LOC() AND ILOC() METHODS IN PANDAS

If you want to display the contents of a record in a Pandas data frame, specify the index of the row in the loc() method. For example, the following code snippet displays the data by feature name in a data frame df:

```
df.loc[feature_name]
```

Select the first row of the "height" column in the data frame:

```
df.loc([0], ['height'])
```

The following code snippet uses the `iloc()` function to display the first eight records of the name column with this code snippet:

```
df.iloc[0:8]['name']
```

CONVERTING CATEGORICAL DATA TO NUMERIC DATA

One common task in machine learning involves converting a feature containing character data into a feature that contains numeric data. Listing A.8 shows the contents of `cat2numeric.py` that illustrate how to replace a text field with a corresponding numeric field.

Listing A.8: cat2numeric.py

```
import pandas as pd
import numpy as np

df = pd.read_csv('sometext.csv', delimiter='\t')

print("=> First five rows (before):")
print(df.head(5))
print("-------------------------")
print()

# map ham/spam to 0/1 values:
df['type'] = df['type'].map( {'ham':0 , 'spam':1} )

print("=> First five rows (after):")
print(df.head(5))
print("-------------------------")
```

Listing A.8 initializes the data frame `df` with the contents of the `csv` file `sometext.csv`, and then displays the contents of the first five rows by invoking `df.head(5)`, which is also the default number of rows to display.

The next code snippet in Listing A.8 invokes the `map()` method to replace occurrences of `ham` with 0 and replace occurrences of `spam` with 1 in the column labeled `type`, as shown here:

```
df['type'] = df['type'].map( {'ham':0 , 'spam':1} )
```

The last portion of Listing A.8 invokes the `head()` method again to display the first five rows of the dataset after having renamed the contents of the column type. Launch the code in Listing A.8 to see the following output:

```
=> First five rows (before):
   type                          text
0   ham          I'm telling the truth
1  spam      What a deal such a deal!
2  spam  Free vacation for your family
3   ham        Thank you for your help
4  spam        Spring break next week!
-------------------------

=> First five rows (after):
   type                          text
0     0          I'm telling the truth
1     1      What a deal such a deal!
2     1  Free vacation for your family
3     0        Thank you for your help
4     1        Spring break next week!

-------------------------
```

As another example, Listing A.9 shows the contents of `shirts.csv` and Listing A.10 shows the contents of `shirts.py`. These examples illustrate four techniques for converting categorical data into numeric data.

Listing A.9: shirts.csv

```
type,ssize
shirt,xxlarge
shirt,xxlarge
shirt,xlarge
shirt,xlarge
shirt,xlarge
shirt,large
shirt,medium
shirt,small
shirt,small
shirt,xsmall
shirt,xsmall
shirt,xsmall
```

Listing A.10: shirts.py

```
import pandas as pd
```

```
shirts = pd.read_csv("shirts.csv")
print("shirts before:")
print(shirts)
print()

# TECHNIQUE #1:
#shirts.loc[shirts['ssize']=='xxlarge','size'] = 4
#shirts.loc[shirts['ssize']=='xlarge', 'size'] = 4
#shirts.loc[shirts['ssize']=='large',  'size'] = 3
#shirts.loc[shirts['ssize']=='medium', 'size'] = 2
#shirts.loc[shirts['ssize']=='small',  'size'] = 1
#shirts.loc[shirts['ssize']=='xsmall', 'size'] = 1

# TECHNIQUE #2:
#shirts['ssize'].replace('xxlarge', 4, inplace=True)
#shirts['ssize'].replace('xlarge',  4, inplace=True)
#shirts['ssize'].replace('large',   3, inplace=True)
#shirts['ssize'].replace('medium',  2, inplace=True)
#shirts['ssize'].replace('small',   1, inplace=True)
#shirts['ssize'].replace('xsmall',  1, inplace=True)

# TECHNIQUE #3:
#shirts['ssize'] = shirts['ssize'].apply({'xxlarge':4, 'xlarge':4, 'large':3, 'medium':2, 'small':1,
'xsmall':1}.get)

# TECHNIQUE #4:
shirts['ssize'] = shirts['ssize'].replace(regex='xlarge', value=4)
shirts['ssize'] = shirts['ssize'].replace(regex='large',  value=3)
shirts['ssize'] = shirts['ssize'].replace(regex='medium', value=2)
shirts['ssize'] = shirts['ssize'].replace(regex='small',  value=1)

print("shirts after:")
print(shirts)
```

Listing A.10 starts with a code block of six statements that uses direct comparison with strings to make numeric replacements. For example, the following code snippet replaces all occurrences of the string `xxlarge` with the value 4:

```
shirts.loc[shirts['ssize']=='xxlarge','size'] = 4
```

The second code block consists of six statements that use the `replace()` method to perform the same updates, an example of which is shown here:

```
shirts['ssize'].replace('xxlarge', 4, inplace=True)
```

The third code block consists of a single statement that uses the `apply()` method to perform the same updates, as shown here:

```
shirts['ssize'] = shirts['ssize'].apply({'xxlarge':4, 'xlarge':4, 'large':3, 'medium':2, 'small':1,
'xsmall':1}.get)
```

The fourth code block consists of four statements that use regular expressions to perform the same updates, an example of which is shown here:

```
shirts['ssize'] = shirts['ssize'].replace(regex='xlarge', value=4)
```

Since the preceding code snippet matches `xxlarge` as well as `xlarge`, we only need *four* statements instead of six statements. (If you are unfamiliar with regular expressions, you can read online tutorials.) Launch the code in Listing A.10 to see the following output:

```
shirts before
     type     size
0   shirt  xxlarge
1   shirt  xxlarge
2   shirt   xlarge
3   shirt   xlarge
4   shirt   xlarge
5   shirt    large
6   shirt   medium
7   shirt    small
8   shirt    small
9   shirt   xsmall
10  shirt   xsmall
11  shirt   xsmall

shirts after:
     type  size
0   shirt     4
1   shirt     4
2   shirt     4
3   shirt     4
4   shirt     4
5   shirt     3
6   shirt     2
7   shirt     1
8   shirt     1
9   shirt     1
```

```
10  shirt     1
11  shirt     1
```

MATCHING AND SPLITTING STRINGS IN PANDAS

Listing A.11 shows the content of `shirts_str.py`, which illustrates how to match a column value with an initial string and how to split a column value based on a letter.

Listing A.11: shirts_str.py

```
import pandas as pd

shirts = pd.read_csv("shirts2.csv")
print("shirts:")
print(shirts)
print()

print("shirts starting with xl:")
print(shirts[shirts.ssize.str.startswith('xl')])
print()

print("Exclude 'xlarge' shirts:")
print(shirts[shirts['ssize'] != 'xlarge'])
print()

print("first three letters:")
shirts['sub1'] = shirts['ssize'].str[:3]
print(shirts)
print()

print("split ssize on letter 'a':")
shirts['sub2'] = shirts['ssize'].str.split('a')
print(shirts)
print()

print("Rows 3 through 5 and column 2:")
print(shirts.iloc[2:5, 2])
print()
```

Listing A.11 initializes the data frame `df` with the contents of the `csv` file `shirts.csv`, and then displays the contents of `df`. The next code snippet in Listing A.11 uses the `startswith()` method to match the shirt types that start with the letters `xl`, followed by a code snippet that displays the shorts whose size does not equal the string `xlarge`.

The next code snippet uses the construct `str[:3]` to display the first three letters of the shirt types, followed by a code snippet that uses the `split()` method to split the shirt types based on the letter “a.”

The final code snippet invokes `iloc[2:5,2]` to display the contents of rows 3 through 5 inclusive, and only the second column. The output of Listing A.11 is as follows:

```
shirts:
     type    ssize
0   shirt  xxlarge
1   shirt  xxlarge
2   shirt   xlarge
3   shirt   xlarge
4   shirt   xlarge
5   shirt    large
6   shirt   medium
7   shirt    small
8   shirt    small
9   shirt   xsmall
10  shirt   xsmall
11  shirt   xsmall

shirts starting with xl:
    type   ssize
2  shirt  xlarge
3  shirt  xlarge
4  shirt  xlarge

Exclude 'xlarge' shirts:
     type    ssize
0   shirt  xxlarge
1   shirt  xxlarge
5   shirt    large
6   shirt   medium
7   shirt    small
8   shirt    small
9   shirt   xsmall
10  shirt   xsmall
11  shirt   xsmall

first three letters:
     type    ssize sub1
0   shirt  xxlarge  xxl
```

```
1   shirt  xxlarge  xxl
2   shirt   xlarge  xla
3   shirt   xlarge  xla
4   shirt   xlarge  xla
5   shirt    large  lar
6   shirt   medium  med
7   shirt    small  sma
8   shirt    small  sma
9   shirt   xsmall  xsm
10  shirt   xsmall  xsm
11  shirt   xsmall  xsm

split ssize on letter 'a':
     type    ssize sub1        sub2
0   shirt  xxlarge  xxl  [xxl, rge]
1   shirt  xxlarge  xxl  [xxl, rge]
2   shirt   xlarge  xla   [xl, rge]
3   shirt   xlarge  xla   [xl, rge]
4   shirt   xlarge  xla   [xl, rge]
5   shirt    large  lar    [l, rge]
6   shirt   medium  med    [medium]
7   shirt    small  sma    [sm, ll]
8   shirt    small  sma    [sm, ll]
9   shirt   xsmall  xsm   [xsm, ll]
10  shirt   xsmall  xsm   [xsm, ll]
11  shirt   xsmall  xsm   [xsm, ll]

Rows 3 through 5 and column 2:
2    xlarge
3    xlarge
4    xlarge
Name: ssize, dtype: object
```

CONVERTING STRINGS TO DATES IN PANDAS

Listing A.12 shows the content of `string2date.py`, which illustrates how to convert strings to date formats.

Listing A.12: string2date.py

```
import pandas as pd

bdates1 = {'strdates':  ['20210413','20210813','20211225'],
           'people': ['Sally','Steve','Sarah']
          }

df1 = pd.Data frame(bdates1, columns = ['strdates','people'])
df1['dates'] = pd.to_datetime(df1['strdates'], format='%Y%m%d')
print("=> Contents of data frame df1:")
print(df1)
print()
print(df1.dtypes)
print()

bdates2 = {'strdates': ['13Apr2021','08Aug2021','25D ec2021'],
'people': ['Sally','Steve','Sarah']
          }

df2 = pd.Data frame(bdates2, columns = ['strdates','people'])
df2['dates'] = pd.to_datetime(df2['strdates'], format='%d%b%Y')
print("=> Contents of data frame df2:")
print(df2)
print()

print(df2.dtypes)
print()
```

Listing A.12 initializes the data frame `df1` with the contents of `bdates1`, and then converts the `strdates` column to dates using the `%Y%m%d` format. The next portion of Listing A.12 initializes the data frame `df2` with the contents of `bdates2`, and then converts the `strdates` column to dates using the `%d%b%Y` format. Launch the code in Listing A.12 to see the following output:

```
=> Contents of data frame df1:
   strdates people      dates
0  20210413  Sally 2021-04-13
1  20210813  Steve 2021-08-13
2  20211225  Sarah 2021-12-25

strdates            object
people              object
dates       datetime64[ns]
dtype: object

=> Contents of data frame df2:
    strdates people      dates
```

```
0  13Apr2021  Sally 2021-04-13
1  08Aug2021  Steve 2021-08-08
2  25Dec2021  Sarah 2021-12-25

strdates            object
people              object
dates       datetime64[ns]
dtype: object
```

WORKING WITH DATE RANGES IN PANDAS

Listing A.13 shows the content of `pand_parse_dates.py` that shows how to work with date ranges in a CSV file.

Listing A.13: pand_parse_dates.py

```
import pandas as pd

df = pd.read_csv('multiple_dates.csv', parse_dates=['dates'])

print("df:")
print(df)
print()

df = df.set_index(['dates'])
start_d = "2021-04-30"
end_d   = "2021-08-31"

print("DATES BETWEEN",start_d,"AND",end_d,":")
print(df.loc[start_d:end_d])
print()

print("DATES BEFORE",start_d,":")
print(df.loc[df.index < start_d])

years = ['2020','2021','2022']
for year in years:
  year_sum = df.loc[year].sum()[0]
  print("SUM OF VALUES FOR YEAR",year,":",year_sum)
```

Listing A.13 starts by initializing the variable `df` with the contents of the CSV file `multiple_dates.csv` and then displaying its contents. The next code snippet sets the dates column as the index column and then initializes the variable `start_d` and `end_d` that contain a start date and an end date, respectively.

The next portion of Listing A.13 displays the dates between `start_d` and `end_d`, and then the list of dates that precede `start_d`. The final code block iterates through a list of years and then calculates the sum of the numbers in the values field for each year in the list. Launch the code in Listing A.13 to see the following output:

```
df:
        dates  values
0  2020-01-31    40.0
1  2020-02-28    45.0
2  2020-03-31    56.0
3  2021-04-30     NaN
4  2021-05-31     NaN
5  2021-06-30   140.0
6  2021-07-31    95.0
7  2022-08-31    40.0
8  2022-09-30    55.0
9  2022-10-31     NaN
10 2022-11-15    65.0

DATES BETWEEN 2021-04-30 AND 2021-08-31 :
            values
dates
2021-04-30     NaN
2021-05-31     NaN
2021-06-30   140.0
2021-07-31    95.0

DATES BEFORE 2021-04-30 :
            values
dates
2020-01-31    40.0
2020-02-28    45.0
2020-03-31    56.0

SUM OF VALUES FOR YEAR 2020 : 141.0
SUM OF VALUES FOR YEAR 2021 : 235.0
SUM OF VALUES FOR YEAR 2022 : 160.0
```

DETECTING MISSING DATES IN PANDAS

Listing A.14 shows the contents of pandas_missing_dates.py that shows how to detect missing date values in a CSV file.

Listing A.14: pandas_missing_dates.py

```
import pandas as pd

# A data frame from a dictionary of lists
data = {'Date': ['2021-01-18', '2021-01-20', '2021-01-21', '2021-01-24'],
        'Name': ['Joe', 'John', 'Jane', 'Jim']}
df = pd.Data frame(data)

# Setting the Date values as index:
df = df.set_index('Date')

# to_datetime() converts string format to a DateTime object:
df.index = pd.to_datetime(df.index)

start_d="2021-01-18"
end_d="2021-01-25"

# display dates that are not in the sequence:
print("MISSING DATES BETWEEN",start_d,"and",end_d,":")
dates = pd.date_range(start=start_d, end=end_d).difference(df.index)

for date in dates:
  print("date:",date)
print()
```

Listing A.14 initializes the dictionary data with a list of values for the Date field and the Name field, after which the variable df is initialized as a data frame whose contents are from the data variable.

The next code snippet sets the Date field as the index of the data frame df, after which the string-based dates are converted to DateTime objects. Another pair of code snippets initialize the variable start_d and end_d with a start date and an end date, respectively.

The final portion of Listing A.14 initializes the variable dates with the list of missing dates between start_d and end_d, after which the contents of dates are displayed. Launch the code in Listing A.14 to see the following output:

```
MISSING DATES BETWEEN 2021-01-18 and 2021-01-25 :
date: 2022-01-19 00:00:00
date: 2022-01-22 00:00:00
date: 2022-01-23 00:00:00
date: 2022-01-25 00:00:00
```

INTERPOLATING MISSING DATES IN PANDAS

Listing A.15 shows the contents of missing_dates.csv and Listing A.16 shows the contents of pandas_interpolate.py that shows how to replace NaN values with interpolated values that are calculated in several ways.

Listing A.15: missing_dates.csv

```
"dates","values"
2021-01-31,40
2021-02-28,45
2021-03-31,56
2021-04-30,NaN
2021-05-31,NaN
2021-06-30,140
2021-07-31,95
2021-08-31,40
2021-09-30,55
2021-10-31,NaN
2021-11-15,65
```

Notice the value 140 (shown in bold) in Listing A.15: This value is an outlier, which will affect the calculation of the interpolated values, and potentially generate additional outliers.

Listing A.16: pandas_interpolate.py

```
import pandas as pd
df = pd.read_csv("missing_dates.csv")

# fill NaN values with linear interpolation:
df1 = df.interpolate()

# fill NaN values with quadratic polynomial interpolation:
df2 = df.interpolate(method='polynomial', order=2)

# fill NaN values with cubic polynomial interpolation:
df3 = df.interpolate(method='polynomial', order=3)
```

```
print("original data frame:")
print(df)
print()
print("linear interpolation:")
print(df1)
print()
print("quadratic interpolation:")
print(df2)
print()
print("cubic interpolation:")
print(df3)
print()
```

Listing A.16 initializes `df` with the contents of the `CSV` file `missing_dates.csv` and then initializes the three data frames `df1`, `df2`, and `df3` that are based on linear, quadratic, and cubic interpolation, respectively, via the `interpolate()` method. Launch the code in Listing A.16 to see the following output:

```
original data frame:
         dates  values
0   2021-01-31    40.0
1   2021-02-28    45.0
2   2021-03-31    56.0
3   2021-04-30     NaN
4   2021-05-31     NaN
5   2021-06-30   140.0
6   2021-07-31    95.0
7   2021-08-31    40.0
8   2021-09-30    55.0
9   2021-10-31     NaN
10  2021-11-15    65.0

linear interpolation:
         dates  values
0   2021-01-31    40.0
1   2021-02-28    45.0
2   2021-03-31    56.0
3   2021-04-30    84.0
4   2021-05-31   112.0
5   2021-06-30   140.0
6   2021-07-31    95.0
7   2021-08-31    40.0
8   2021-09-30    55.0
9   2021-10-31    60.0
10  2021-11-15    65.0

quadratic interpolation:
         dates      values
0   2021-01-31   40.000000
1   2021-02-28   45.000000
2   2021-03-31   56.000000
3   2021-04-30   88.682998
4   2021-05-31  136.002883
5   2021-06-30  140.000000
6   2021-07-31   95.000000
7   2021-08-31   40.000000
8   2021-09-30   55.000000
9   2021-10-31   68.162292
10  2021-11-15   65.000000

cubic interpolation:
         dates      values
0   2021-01-31   40.000000
1   2021-02-28   45.000000
2   2021-03-31   56.000000
3   2021-04-30   92.748096
4   2021-05-31  132.055687
5   2021-06-30  140.000000
6   2021-07-31   95.000000
7   2021-08-31   40.000000
8   2021-09-30   55.000000
9   2021-10-31   91.479905
10  2021-11-15   65.000000
```

OTHER OPERATIONS WITH DATES IN PANDAS

Listing A.17 shows the contents of `pandas_misc1.py` that shows how to extract a list of years from a column in a data frame.

Listing A.17: pandas_misc1.py

```
import pandas as pd
import numpy as np

df = pd.read_csv('multiple_dates.csv', parse_dates=['dates'])
print("df:")
print(df)
print()
```

```
year_list = df['dates']

arr1 = np.array([])
for long_year in year_list:
  year = str(long_year)
  short_year = year[0:4]
  arr1 = np.append(arr1,short_year)

unique_years = set(arr1)
print("unique_years:")
print(unique_years)
print()

unique_arr = np.array(pd.Data frame.from_dict(unique_years))
print("unique_arr:")
print(unique_arr)
print()
```

Listing A.17 initializes `df` with the contents of the `CSV` file `multiple_dates.csv` and then displays its contents. The next portion of Listing A.17 initializes `year_list` with the dates column of `df`.

The next code block contains a loop that iterates through the elements in `year_list`, extracts the first four characters (i.e., the year value), and appends that substring to the `NumPy` array `arr1`. The final code block initializes the variable `unique_arr` as a `Numpy` array consisting of the unique years in the dictionary `unique_years`. Launch the code in Listing A.17 to see the following output:

```
df:
        dates  values
0  2020-01-31    40.0
1  2020-02-28    45.0
2  2020-03-31    56.0
3  2021-04-30     NaN
4  2021-05-31     NaN
5  2021-06-30   140.0
6  2021-07-31    95.0
7  2022-08-31    40.0
8  2022-09-30    55.0
9  2022-10-31     NaN
10 2022-11-15    65.0

unique_years:
{'2022', '2020', '2021'}

unique_arr:
[['2022']
['2020']
['2021']]
```

Listing A.18 shows the contents of `pandas_misc2.py` that shows how to iterate through the rows of a data frame. Keep in mind that row-wise iteration is not recommended because it can result in performance issues in larger datasets.

Listing A.18: pandas_misc2.py

```
import pandas as pd

df = pd.read_csv('multiple_dates.csv', parse_dates=['dates'])

print("df:")
print(df)
print()

print("=> ITERATE THROUGH THE ROWS:")
for idx,row in df.iterrows():
  print("idx:",idx," year:",row['dates'])
print()
```

Listing A.18 initializes the `Pandas` data frame `df`, prints its contents, and then processes the rows of `df` in a loop. During each iteration, the current index and row contents are displayed. Launch the code in Listing A.18 to see the following output:

```
df:
        dates  values
0  2020-01-31    40.0
1  2020-02-28    45.0
2  2020-03-31    56.0
3  2021-04-30     NaN
4  2021-05-31     NaN
5  2021-06-30   140.0
6  2021-07-31    95.0
7  2022-08-31    40.0
8  2022-09-30    55.0
9  2022-10-31     NaN
10 2022-11-15    65.0
```

```
=> ITERATE THROUGH THE ROWS:
idx: 0   year: 2020-01-31 00:00:00
idx: 1   year: 2020-02-28 00:00:00
idx: 2   year: 2020-03-31 00:00:00
idx: 3   year: 2021-04-30 00:00:00
idx: 4   year: 2021-05-31 00:00:00
idx: 5   year: 2021-06-30 00:00:00
idx: 6   year: 2021-07-31 00:00:00
idx: 7   year: 2022-08-31 00:00:00
idx: 8   year: 2022-09-30 00:00:00
idx: 9   year: 2022-10-31 00:00:00
idx: 10  year: 2022-11-15 00:00:00
```

Listing A.19 shows the contents of `pandas_misc3.py` that shows how to display a weekly set of dates that are between a start date and an end date.

Listing A.19: pandas_misc3.py

```
import pandas as pd

start_d="01/02/2022"
end_d="12/02/2022"
weekly_dates=pd.date_range(start=start_d, end=end_d, freq='W')

print("Weekly dates from",start_d,"to",end_d,":")
print(weekly_dates)
```

Listing A.19 starts with initializing the variable `start_d` and `end_d` that contain a start date and an end date, respectively, and then initializes the variable `weekly_dates` with a list of weekly dates between the start date and the end date. Launch the code in Listing A.19 to see the following output:

```
Weekly dates from 01/02/2022 to 12/02/2022 :
DatetimeIndex(['2022-01-02', '2022-01-09', '2022-01-16', '2022-01-23',
                    '2022-01-30', '2022-02-06', '2022-02-13', '2022-02-20',
                    '2022-02-27', '2022-03-06', '2022-03-13', '2022-03-20',
                    '2022-03-27', '2022-04-03', '2022-04-10', '2022-04-17',
                    '2022-04-24', '2022-05-01', '2022-05-08', '2022-05-15',
                    '2022-05-22', '2022-05-29', '2022-06-05', '2022-06-12',
                    '2022-06-19', '2022-06-26', '2022-07-03', '2022-07-10',
                    '2022-07-17', '2022-07-24', '2022-07-31', '2022-08-07',
                    '2022-08-14', '2022-08-21', '2022-08-28', '2022-09-04',
                    '2022-09-11', '2022-09-18', '2022-09-25', '2022-10-02',
                    '2022-10-09', '2022-10-16', '2022-10-23', '2022-10-30',
              '2022-11-06', '2022-11-13', '2022-11-20', '2022-11-27'],
              dtype='datetime64[ns]', freq='W-SUN')
```

MERGING AND SPLITTING COLUMNS IN PANDAS

Listing A.20 shows the contents of `employees.csv` and Listing A.21 shows the contents of `emp_merge_split.py`. These examples illustrate how to merge columns and split columns of a `csv` file.

Listing A.20: employees.csv

```
name,year,month
Jane-Smith,2015,Aug
Dave-Smith,2020,Jan
Jane-Jones,2018,Dec
Jane-Stone,2017,Feb
Dave-Stone,2014,Apr
Mark-Aster,,Oct
Jane-Jones,NaN,Jun
```

Listing A.21: emp_merge_split.py

```
import pandas as pd

emps = pd.read_csv("employees.csv")
print("emps:")
print(emps)
print()

emps['year']  = emps['year'].astype(str)
emps['month'] = emps['month'].astype(str)

# separate column for first name and for last name:
emps['fname'],emps['lname'] = emps['name'].str.split("-",1).str

# concatenate year and month with a "#" symbol:
emps['hdate1'] = emps['year'].astype(str)+"#"+emps['month'].astype(str)

# concatenate year and month with a "-" symbol:
emps['hdate2'] = emps[['year','month']].agg('-'.join, axis=1)
```

```
print(emps)
print()
```

Listing A.21 initializes the data frame `df` with the contents of the `CSV` file `employees.csv`, and then displays the contents of `df`. The next pair of code snippets invoke the `astype()` method to convert the contents of the `year` and `month` columns to strings.

The next code snippet in Listing A.21 uses the `split()` method to split the `name` column into the columns `fname` and `lname` that contain the first name and last name, respectively, of each employee's name:

```
emps['fname'],emps['lname'] = emps['name'].str.split("-",1).str
```

The next code snippet concatenates the contents of the year and month string with a "**#**" character to create a new column called `hdate1`:

```
emps['hdate1'] = emps['year'].astype(str)+"#"+emps['month'].astype(str)
```

The final code snippet concatenates the contents of the `year` and `month` string with a "-" to create a new column called `hdate2`, as shown here:

```
emps['hdate2'] = emps[['year','month']].agg('-'.join, axis=1)
```

Launch the code in Listing A.21 to see the following output:

```
emps:
         name    year month
0  Jane-Smith  2015.0   Aug
1  Dave-Smith  2020.0   Jan
2  Jane-Jones  2018.0   Dec
3  Jane-Stone  2017.0   Feb
4  Dave-Stone  2014.0   Apr
5  Mark-Aster     NaN   Oct
6  Jane-Jones     NaN   Jun

     name       year        month fname lname     hdate1     hdate2
0  Jane-Smith  2015.0   Aug  Jane  Smith  2015.0#Aug  2015.0-Aug
1  Dave-Smith  2020.0   Jan  Dave  Smith  2020.0#Jan  2020.0-Jan
2  Jane-Jones  2018.0   Dec  Jane  Jones  2018.0#Dec  2018.0-Dec
3  Jane-Stone  2017.0   Feb  Jane  Stone  2017.0#Feb  2017.0-Feb
4  Dave-Stone  2014.0   Apr  Dave  Stone  2014.0#Apr  2014.0-Apr
5  Mark-Aster     nan   Oct  Mark  Aster     nan#Oct     nan-Oct
6  Jane-Jones     nan   Jun  Jane  Jones     nan#Jun     nan-Jun
```

There is one other detail regarding the following commented out code snippet:

```
#emps['fname'],emps['lname'] = emps['name'].str.split("-",1).str
```

The following deprecation message is displayed if you uncomment the preceding code snippet:

```
#FutureWarning: Columnar iteration over characters
#will be deprecated in future releases.
```

READING HTML WEB PAGES IN PANDAS

Listing A.22 displays the contents of the HTML Web page `abc.html`, and Listing A.23 shows the contents of `read_html_page.py` that illustrates how to read the contents of an HTML Web page from `Pandas`. Note that this code will only work with Web pages that contain *at least* one `HTML <table>` element.

Listing A.22: abc.html

```
<html>
<head>
</head>
<body>
  <table>
    <tr>
      <td>hello from abc.html!</td>
    </tr>
  </table>
</body>
</html>
```

Listing A.23: read_html_page.py

```
import pandas as pd

file_name="abc.html"
with open(file_name, "r") as f:
  dfs = pd.read_html(f.read())

print("Contents of HTML Table(s) in the HTML Web Page:")
print(dfs)
```

Listing A.23 starts with an `import` statement, followed by initializing the variable `file_name`

to abc.html that is displayed in Listing A.22. The next code snippet initializes the variable dfs as a data frame with the contents of the HTML Web page abc.html. The final portion of Listing A.19 displays the contents of the data frame dsf. Launch the code in Listing A.23 to see the following output:

```
Contents of HTML Table(s) in the HTML Web Page:
[                  0
0  hello from abc.html!]
```

For more information about the Pandas read_html() method, navigate to this URL:

https://pandas.pydata.org/pandas-docs/stable/reference/api/

SAVING A PANDAS DATA FRAME AS AN HTML WEB PAGE

Listing A.24 shows the contents of read_html_page.py that illustrates how to read the contents of an HTML Web page from Pandas. Note that this code will only work with Web pages that contain at least one HTML <table> element.

Listing A.24: read_html_page.py

```
import pandas as pd

emps = pd.read_csv("employees.csv")
print("emps:")
print(emps)
print()

emps['year']  = emps['year'].astype(str)
emps['month'] = emps['month'].astype(str)

# separate column for first name and for last name:
emps['fname'],emps['lname'] = emps['name'].str.split("-",1).str

# concatenate year and month with a "#" symbol:
emps['hdate1'] = emps['year'].astype(str)+"#"+emps['month'].astype(str)

# concatenate year and month with a "-" symbol:
emps['hdate2'] = emps[['year','month']].agg('-'.join, axis=1)

print(emps)
print()

html = emps.to_html()
print("Data frame as an HTML Web Page:")
print(html)
```

Listing A.24 populates the data frame temps with the contents of employees.csv, and then converts the year and month attributes to type string. The next code snippet splits the contents of the name field with the "-" symbol as a delimiter. As a result, this code snippet populates the new "fname" and "lname" fields with the first name and last name, respectively, of the previously split field.

The next code snippet in Listing A.24 converts the year and month fields to strings, and then concatenates them with a "#" as a delimiter. Yet another code snippet populates the hdate2 field with the concatenation of the year and month fields.

After displaying the content of the data frame emps, the final code snippet populate the variable html with the result of converting the data frame emps to an HTML web page by invoking the to_html() method of Pandas. Launch the code in Listing A.24 to see the following output:

```
Contents of HTML Table(s)
emps:
         name    year month
0  Jane-Smith  2015.0   Aug
1  Dave-Smith  2020.0   Jan
2  Jane-Jones  2018.0   Dec
3  Jane-Stone  2017.0   Feb
4  Dave-Stone  2014.0   Apr
5  Mark-Aster     NaN   Oct
6  Jane-Jones     NaN   Jun


      name      year   month  fname  lname    hdate1      hdate2
0  Jane-Smith  2015.0   Aug  Jane  Smith  2015.0#Aug  2015.0-Aug
1  Dave-Smith  2020.0   Jan  Dave  Smith  2020.0#Jan  2020.0-Jan
2  Jane-Jones  2018.0   Dec  Jane  Jones  2018.0#Dec  2018.0-Dec
3  Jane-Stone  2017.0   Feb  Jane  Stone  2017.0#Feb  2017.0-Feb
4  Dave-Stone  2014.0   Apr  Dave  Stone  2014.0#Apr  2014.0-Apr
5  Mark-Aster     nan   Oct  Mark  Aster     nan#Oct     nan-Oct

6  Jane-Jones     nan   Jun  Jane  Jones     nan#Jun     nan-Jun
```

```
Data frame as an HTML Web Page:
<table border="1" class="data frame">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>name</th>
      <th>year</th>
      <th>month</th>
      <th>fname</th>
      <th>lname</th>
      <th>hdate1</th>
      <th>hdate2</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th>0</th>
      <td>Jane-Smith</td>
      <td>2015.0</td>
      <td>Aug</td>
      <td>Jane</td>
      <td>Smith</td>
      <td>2015.0#Aug</td>
      <td>2015.0-Aug</td>
    </tr>
    <tr>
      <th>1</th>
      <td>Dave-Smith</td>
      <td>2020.0</td>
      <td>Jan</td>
      <td>Dave</td>
      <td>Smith</td>
      <td>2020.0#Jan</td>
      <td>2020.0-Jan</td>
    </tr>
    // details omitted for brevity
    <tr>
      <th>6</th>
      <td>Jane-Jones</td>
      <td>nan</td>
      <td>Jun</td>
      <td>Jane</td>
      <td>Jones</td>
      <td>nan#Jun</td>
      <td>nan-Jun</td>
    </tr>
  </tbody>
</table>
```

SUMMARY

This appendix introduced you to `Pandas` for creating labeled data frames and displaying metadata of data frames. Then you learned how to create data frames from various sources of data, such as random numbers and hard-coded data values. In addition, you saw how to perform column-based and row-based operations in `Pandas` data frames.

You also learned how to split strings in `Pandas` data frames, and how to create various types of data frames, such as numeric and Boolean data frames. In addition, you learned how to create data frames with `NumPy` functions and random numbers.

INDEX