Intermediate Python

O. Campesato

Intermediate Python

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.

Intermediate Python

Oswald Campesato

Mercury Learning and Information
Dulles, Virginia
Boston, Massachusetts
New Delhi

Publisher: David Pallai

Mercury Learning and Information
121 High Street, 3rd Floor
Boston, MA 02110
info@merclearning.com
www.merclearning.com
800-232-0223

O. Campesato. *Intermediate Python.*
ISBN: 978-1-50152-174-4

Library of Congress Control Number: 2023942701
232425321 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 endeavors to provide you with relevant information about intermediate Python 3.x for a variety of topics, such as comprehensions, iterators, generators, regular expressions, OOP, queues and stacks, and recursion. Each chapter contains an assortment of code samples that illustrate the topics in the chapter material.

THE TARGET AUDIENCE

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

GETTING THE MOST FROM THIS BOOK

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

Moreover, some programmers want to run the code first, see what it does, and then return to the code to delve into the details (while 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?

Knowledge of other programming languages (such as Java) can also be helpful because of the exposure to programming concepts and object-oriented programming. Knowledge of data structures, such as stacks and queues, is also helpful. 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.

O. Campesato
August 2023

CHAPTER 1

DATA STRUCTURES IN PYTHON

This chapter introduces an assortment of Python data structures, including lists, vectors, matrices, queues, tuples, and dictionaries, as well as a section that discusses functional programming.

The first part of this chapter discusses lists and operations that you can perform on Python lists. This section also contains simple examples of comprehensions. 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, along with code samples. The final part of this chapter discusses functional programming.

NOTE *The Python files in this book are for Python 3.x.*

With the preceding points in mind, let's take a look at the `list` data type, 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 containing multiple 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 in a given list. 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, (2024,03,31)]
>>> x1,x2,x3,date1 = line
>>> x1
'a'
>>> x2
10
>>> x3
20
>>> date1
(2024, 3, 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, (2024,03,31)]
>>> x1,x2,x3,(year,month,day) = line
>>> x1
'a'
>>> x2
10
>>> x3
20
>>> year
2024
>>> month
3
>>> day
31
```

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

```
>>> line = ['a', 10, 20, (2024,03,31)]
>>> 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, as shown here:

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

The `sort()` method sorts a list "in place," so you do not assign the result to a variable:

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

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

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

Yet another way to reverse a list is as follows:

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

Keep in mind that `reversed(alist)` is an *iterable* (discussed in more detail in Chapter 2) and not a list. However, you can convert the reversed list to a list with this code snippet:

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

Listing 1.1 contains examples of *comprehensions*, some of which contain conditional logic.

LISTING 1.1: 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 1.1 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 = [3,1,4,2]
x.sort()
min1 = min(x)
x.reverse()
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)
```

Lists and Filter-Related Operations

Python enables you to filter a list, which is also called a *list comprehension* (discussed in greater detail in Chapter 2), 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]
squared = [ n*n   for n in nums ]
cubes   = [ n*n*n for n in nums ]

print('nums:   ',nums)
print('squared:',cubes)
print('cubes:  ',cubes)
```

The output from the preceding code block is here:

```
nums:    [1, 2, 3, 4]
squared: [1, 4, 9, 16]
cubes:   [1, 8, 27, 64]
```

SORTING LISTS OF NUMBERS AND STRINGS

Listing 1.2 shows the content of the Python file `sorted1.py` that determines whether two lists are sorted.

LISTING 1.2: 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 1.2 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 1.2 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 because the collating sequence for ASCII places uppercase letters (decimal 65 through decimal 90) before lowercase letters (decimal 97 through decimal 122). 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 the elements in a list, as shown here:

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

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

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

Either of the following is preferred:

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

THE PYTHON RANGE() FUNCTION

This section involves 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 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 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 and Uppercase and Lowercase Letters

Listing 1.3 shows the content of the Python file `counter1.py` that counts the occurrences of digits and letters in a string.

LISTING 1.3: 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 1.3 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 1.3 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 a built-in array type (`import  array`), which is essentially a heterogeneous list, the array type provides a slight savings 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 1.4 shows the content of `count_word2.py` that displays the top three words with the greatest frequency.

LISTING 1.4: count_word2.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 1.4 initializes the variable `mywords` as a list 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 1.4 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]
```

Let's 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 update portions of a list. Listing 1.5 displays the content of `list_slices1.py` that illustrates how to extract a substring from a Python list.

LISTING 1.5: 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 1.5 initializes the variable `list1` with the integers from 0 to 7 and displays its contents. The next code snippet inserts 4 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 replaces 7 with [5000]. Launch Listing 1.5, 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 1.6 displays the content of `substrings1.py` that illustrates how to split a text string into various substrings.

LISTING 1.6: 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 1.6 initializes the variable `my_str` as a text string, followed by the variables `str1`, `str2`, and `str3`. Launch Listing 1.6, 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 1.7 displays the content of `substrings2.py` that illustrates how to split a text string into various substrings.

LISTING 1.7: 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 1.7 initializes the variable `my_str` as a text string, followed by the variables `str1`, `str2`, and `str3`. Launch Listing 1.7, 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()`. Python also supports the functions `index()`, `count()`, `sort()`, and `reverse()`. Examples of these functions are illustrated in the following code snippets.

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 1.8 shows the content of `my_vectors.py` that illustrates how to perform vector-based operations.

LISTING 1.8: my_vectors.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 1.8 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 1.8 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 1.9 shows the content of `my2dmatrix.py` that illustrates how to create and populate 2 two-dimensional matrices.

LISTING 1.9: 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 1.9 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 1.9 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 1.9 prints the contents of `my2DMatrix`. The output from Listing 1.9 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. Thus, the oldest item in a queue is removed when a new item is added to a queue that is already full. Note that queues are discussed in more detail in Chapter 5. Although you can use a list to emulate a queue, 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 either with or without brackets (square brackets are for lists, round brackets are for tuples, 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 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 is as follows:

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

The intersection of s and n is as follows:

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

The exclusive-or of s and n is as follows:

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

DICTIONARIES

Python has a key/value structure called a *dictionary* (dict), which 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 dictionary 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 specifies 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` exception. 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:

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

for key, value in my_dict.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 1.10 contains an example of doing so.

LISTING 1.10: interpolate_dict.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 1.10 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 the `(key,value)` pairs in a given dictionary in arbitrary order. However, 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. The following code samples show you various aspects of an `OrderedDict`.

```
from collections import OrderedDict
print("Dictionary:\n")
d = {}
d['a'] = 10
d['b'] = 20
d['c'] = 30

for key, value in d.items():
  print("key:",key,"value:",value)

print("\nOrdered Dictionary:\n")
od = OrderedDict()
od['a'] = 10
od['b'] = 20
od['c'] = 30

for key, value in od.items():
  print("key:",key,"value:",value)

# updating values in OrderedDict:
od = OrderedDict()
od['a'] = 10
od['b'] = 20
od['c'] = 30
for key, value in od.items():
  print(key, value)

print("\nAfter updating key 'c':")
od['c'] = 5
for key, value in od.items():
  print(key, value)

print("\nAfter appending key 'bb':")
od['bb'] = 40
for key, value in od.items():
  print(key, value)

print("\nAfter moving key 'a':")
od.move_to_end('a')
for key, value in od.items():
  print(key, value)
```

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 from a dictionary 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 sort 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. You can even sort just the keys, as shown here:

```
my_dict = {12:1024, 3:8, 2:777, 4:531, 1:100}
sorted1 = sorted(my_dict)

print("initial:",my_dict)
print("sorted: ",sorted1)
```

The output of the preceding code block is here:

```
initial: {12: 1024, 3: 8, 2: 777, 4: 531, 1: 100}
sorted:  [1, 2, 3, 4, 12]
```

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'
```

Multiple Dictionaries

You can define entries in a dictionary so that they reference lists or other types of structures. Listing 1.11 shows the content of `multi_dictionary1.py` that illustrates how to define more complex dictionaries.

LISTING 1.11: multi_dictionary1.py

```
from collections import defaultdict

d = {'a' : [1, 2, 3], 'b' : [4, 5]}
print('first:  ',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 1.11 starts by defining the dictionary `d` and printing its contents. The next portion of Listing 1.11 specifies a list-oriented dictionary, and then modifies the values for the keys `a` and `b`. The final portion of Listing 1.11 specifies a set-oriented dictionary, and then modifies the values for the keys `a` and `b` as well.

The output from Listing 1.11 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

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

You can also iterate through a sequence and retrieve the position index and corresponding value at the same time using the `enumerate()` function, as shown here:

```
>>> 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. Keep in mind that `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. Let us now consider mutable and immutable types.

MUTABLE AND IMMUTABLE TYPES

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 1.12.

LISTING 1.12: mutability1.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 1.12 is here:

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

Thus, a type is immutable if its value cannot be changed 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, otherwise you will see this type of error message:

```
"TypeError: unhashable type"
```

Since strings are immutable in Python, you cannot insert a string in the "middle" of a given text string: however, you can construct a second string using concatenation. For example, suppose you have the following 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, which 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 statement in 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]
```

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. 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 1.13 displays the contents of `serialize1.py` that illustrates how to serialize and then deserialize a Python object.

LISTING 1.13: 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 1.13 starts with an `import` statement, followed by the `data` variable that is initialized as a list containing 5 numbers. Next, the file `testfile` is created and the pickled contents of data are stored in that file. The remainder of Listing 1.13 reverses the process and prints the de-pickled contents that match the original contents of the `data` variable. The output from Listing 1.13 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)
```

MODULES VERSUS PACKAGES

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.xmlparse`, 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
```

USER-DEFINED FUNCTIONS

A user-defined function involves the following:

- The keyword `def`
- The name of the function
- Arguments for the function (if any)
- The body of the function

The following code samples illustrate two very simple user-defined functions. The first example shows you how to define a Python function `add_numbers` that adds two numbers and returns the sum.

```
def add_numbers(a,b):
  return a+b
result = add_numbers(3,5)
print("Sum of",a,"and",b,"equals",result)
```

The next example shows you how to define a Python function that concatenates two strings and returns the result.

```
def concat_strings(str1,str2):
  return str1 + str2

result = concat_strings("hello","world")
print("Concatenation:",result)
```

Chapter 2 contains additional examples of user-defined functions that show you how to specify the data types of the arguments and execute the functions in a list of functions.

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 a function and a method is whether the function is defined in a class. Functions in a module are just functions, whereas functions in a class are *methods* of the class or methods of the resulting objects.

In addition, "function" is used in a general manner (all methods are functions, but not all functions are methods). Hence, the word "function" is used as a generic term, whereas the word "method" is used specifically regarding classes or objects (the method of the list type, the methods of the `str` type, and so forth). One more detail: if you define a standalone Python function (i.e., it's not part of a Python class) and you place that function inside a Python class, then it's called a method instead of a function.

Functions with Generic Arguments

Suppose you want to define a function that can take any number of arguments and returns the sum. The Python-based solution involves specifying an argument that begins with an asterisk ("*"), followed by any remaining arguments. The first argument is then a tuple of the other unmatched values. Here is an example:

```
def add_numbers(*values):
    my_sum = 0
    for val in values:
        my_sum += val
    return my_sum

add_numbers(1,2,3,4,5)
```

The preceding function `add_numbers()` specifies the argument `*values`, which means that it will be a list of all the values that are specified when this function is invoked. The body of the function calculates the sum of the numbers in the list `*values`, which in this example equals 15.

Functions that Specify *args

The * operator constructs a dictionary that contains retrieve all the *named* values. The typical name associated with the ** operator is `kwargs`, but it is just a convention. Consider the following function:

```
def my_func(*args):
    print("args:", args)

my_func("Chicago", "Guiness", "Spicy")
```

Place the preceding code block in a Python file and when you launch the code, you will see the following output:

```
args: ('Chicago', 'Guiness', 'Spicy')
```

Functions that Specify **kwargs

The ** operator constructs a dictionary that consists of all the *named* values that appear as the arguments of a Python function. The typical name associated with the ** operator is `kwargs`, but it is just a convention. For example, consider the following Python function:

```
def my_func(**kwargs):
    print("kwargs:", kwargs)
    if 'pizza' in kwargs:
        print("The value of pizza is:", kwargs['pizza'])
    else:
        print("No pizza available")

my_func(pizza="Chicago", beer="Guiness")
```

Place the preceding code block in a Python file, and when you launch the code, you will see the following output:

```
kwargs: {'pizza': 'Chicago', 'beer': 'Guiness'}
The value of pizza is: Chicago
```

The * and ** operators enable you to define Python functions that involve lists of arguments of arbitrary length.

SUMMARY

This chapter started with a discussion of lists and operations, such as splicing and updating lists. Next, you learned about the `range()` function and how to count digits in a string, as well as converting strings to uppercase and lowercase. You also learned about the `split()` function and how to take slices of lists.

In addition, you learned how to work with vectors and matrices, along with some simple code samples. Then you got an introduction to queues, tuples, and sets. You also learned how to work with a dictionary, such creating a dictionary and displaying its contents, as well as iterating through its keys and values.

Next, you learned how to pack and unpack sequences, how to swap pairs of values, and how to iterate through the values in a sequence. Finally, you learned how to create user-defined functions, and about the purpose of `**kwarga` as a function argument.

CHAPTER 2

COMPREHENSIONS, ITERATORS, AND GENERATORS

This chapter contains an eclectic mix of topics that includes comprehensions, iterators, and generators. In addition, this chapter assumes that you are familiar with lists, dictionaries, and sets in `Python`.

The first part of this chapter discusses lambda expressions and comprehensions, followed by an introduction to magic methods. The second portion of the chapter shows you various examples of working with the `filter()` function and `map()` function, and how to combine these two functions.

The third portion of the chapter explains the difference between a function and a method, along with an introduction to Python classes. The final portion of this chapter shows you how to work with function annotations.

Before you read this chapter, please keep in mind the following point. Some concepts in this chapter (such as iterables) are discussed in the context of `Python` classes, and the latter are discussed in more detail in Chapter 4. Therefore, if you are unfamiliar with `Python` classes, you can postpone these topics until after you have read Chapter 4, after which you can return to the material in this chapter.

With the preceding points in mind, let's take a look at a lambda expression, which is discussed in the next section.

LAMBDA EXPRESSIONS

Listing 2.1 shows the content of `lambda1.py`, which illustrates how to create a simple lambda function.

LISTING 2.1: lambda1.py

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

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

print(x1)
print(x2)
```

Listing 2.1 defines the lambda expression `add` that accepts two input parameters and then returns their sum (for numbers) or their concatenation (for strings).

The output from Listing 2.1 is as follows:

```
12
HelloPython
```

COMPREHENSIONS

`Python` enables you to define *comprehensions* to create lists as well as dictionaries. `Python` also supports set-based comprehensions and generator-based comprehensions (generators are discussed later in this chapter). Let's look at some examples that illustrate how to generate comprehensions.

For example, the following comprehension creates a list of integers from 2 to 9, inclusive:

```
num_list1 = [x for x in range(2,10)]
```

The following comprehension creates a list of even integers from 2 to 8, inclusive:

```
num_list2 = [x for x in range(2,10) if x % 2 == 0]
```

The following comprehension creates a list of the cube of even integers from 2 to 8, inclusive:

```
num_list3 = [x*x*x for x in range(2,10) if x % 2 == 0]
```

The following comprehension creates a *dictionary* of integers:

```
num_dict = {x: 2*x for x in range(2,10)}
```

The following comprehension creates a list of words whose character in index 0 equals the letter `A`:

```
my_list = ["Arizona", "Alabama", "California", "Florida"]
lettera = [w for w in my_list if w[0] == "A"]
```

The following code block defines a comprehension that determines only the consonants in a given string:

```
vowels="aeiou"
word="Chicago"
consonants = "".join([ch for ch in word if ch.lower() not in "aeiou"])
print("original   = ",word)
print("consonants = ",consonants)
```

The output of the preceding code block is here:

```
original   =  Chicago
consonants =  Chcg
```

MAGIC METHODS (DUNDERS)

This section contains a very brief description of *magic methods* (also called "Double Underscores" or "Dunders"). Such methods can be classified in terms of their functionality. For example, the first subset of the following magic methods overrides various Boolean operators, and the second subset overrides binary arithmetic operators:

- `__le__()`
- `__lt__()`
- `__ge__()`
- `__gt__()`
- `__add__()`
- `__sub__()`
- `__mul__()`
- `__div__()`

In addition, Python supports magic methods for handling tasks such as iteration and displaying the properties of an object or data structure, some of which are shown here:

- `__iter__()`
- `__len__()`
- `__repr__()`
- `__str__()`

In addition to the preceding list of magic methods, there are some associated functions that are typically invoked instead of (or in conjunction with) the methods in the preceding list, as shown here:

- `iter()`
- `len()`
- `repr()`
- `str()`

An extensive list of Python magic methods is available online:

https://www.geeksforgeeks.org/dunder-magic-methods-python/

https://stackoverflow.com/questions/56238263/list-of-all-python-dunder-magic-methods-which-ones-do-you-need-to-implement-to

You will learn more details regarding magic methods in Chapter 6, which discusses magic methods in the context of custom Python classes.

The Iterator Protocol

The *iterator protocol* is a mechanism that enables you to iterate through container objects. Specifically, a container object (such as an instance of a custom class) must define the method

`container.__iter__()` to support iteration for a container object. Moreover, the method `container.__iter__()` returns an `iterator` object, which in turn must support the following pair of methods that constitute the iterator protocol:

```
iterator.__iter__()
iterator.__next__()
```

The preceding pair of methods serve different purposes. The first method `iterator.__iter__()` returns the iterator object itself so that containers and iterators can be used with various Python statements, such as `for` statements and `in` statements. The second method iterator `__next__()` is the method that returns the next item from the container. As mentioned earlier, iterators are available for comprehensions, loop constructs, and other built-in functions including `map()`, `filter()`, `reduce()`, and `zip()`.

The next section discusses the `iter()` function and `__iter__()` method and how they work together in Python code.

The iter() Function and __iter__() Method

The `iter()` function is a built-in function that returns an iterator of a given object, an example of which is here:

```
iter(object)
```

The `iter()` function requires an argument that is either an iterable or a sequence. In general, the `object` argument can be *any* object that supports either an iteration or sequence protocol.

When you invoke the `iter()` function on an object, the function first searches for the presence of an `__iter__()` method of that object. If the `__iter__()` method exists, the `iter()` function calls that method to get an iterator.

If the `__iter__()` method does not exist, then the `iter()` function will search for a `__getitem__()` method. If the `__getitem__()` exists, the `iter()` function creates an iterator object and returns that object. If the `__getitem__()` does not exist, a `TypeError` exception is raised.

You also have the option to invoke either the `iter()` function or `_iter__()` method to achieve the same result, with one important difference. The `iter()` function performs some additional type checking to determine whether the `__iter__()` method returns an iterator object. If an iterator object is not returned, then `iter()` throws an error. This same functionality is performed in the pair `next()` and `__next__()`.

Keep in mind that the methods `__iter__()` and `__next__()` are user-defined methods if you want to ensure that a custom class is meant to be an iterator. Note that an iterator cannot be reused after all items have been "iterated."

For instance, a list is an ordered collection of items that is also an iterable because a list object has the `__iter__()` method that returns an iterator. Here is an example:

```
numbers = [1, 2, 3]
number_iterator = numbers.__iter__()
print(type(number_iterator))
```

The preceding code block generates the following output:

```
<class 'list_iterator'>
```

Dictionaries and Iterators

The operators `.keys()`, `.values()`, and `.items()` provide iterators for the list of keys, values, and items, respectively, of a dictionary. Consequently, any modification of a dictionary results in a corresponding change in the iterators. Here is an example:

```
mydict = {'Dave': 35, 'Sara': 28}
mykeys = mydict.keys()
mydict['Peter'] = 40
for key in mykeys:
    print(key)
```

The output of the preceding code block is here:

```
Dave
Sara
Peter
```

If we want to iterate through a dictionary that *ignores* subsequent modifications to the dictionary, we can do so by constructing a list from the dictionary, as shown here:

```
mydict = {'Dave': 35, 'Sara': 28}
mylist = list(mydict.keys())
mykeys = mydict.keys()
mydict['Peter'] = 40

for item in mylist:
  print("item:",item)

for key in mykeys:
  print("key:",key)
```

The output of the preceding code block is here:

```
Dave
Sara
```

As you can see, the contents of `mylist` are "decoupled" from `mydict`, and therefore modifications to `mydict` are not reflected in `mylist`.

EXAMPLES OF ITERATORS

The `range()` function is an iterator, which has the ability to generate a sequence of integers when it is invoked. Thus, the `range()` function must be invoked to generate a list of values. However, a list occupies actual memory locations. Specifically, the code snippet `range(10000)` does not generate 10,000 numbers unless this function is invoked, an example of which is shown here:

```
for num in range(10):
  print("num:",num)
```

The following code snippet contains a list of integers that does occupy memory locations:

```
my_list = [1,2,3,4,5,6,7,8,9,10]
```

The preceding distinction between a list and the `range()` function is important because there are situations where you would needlessly generate values that are not processed.

For example, suppose that the function `one_million()` returns the first one million integers, and the function `add_three()` invokes the method `one_million()` and then calculates the sum of the first three integers and ignores the remaining 999,997 numbers. Obviously, this type of code is extremely inefficient (and also unlikely, of course).

Instead of generating an entire set of numbers, a better approach is to generate and process only the numbers that are required. This approach is achievable by defining generators, which are described later in this chapter.

Range versus a List

We can use the `range()` function to iterate through a sequence of numbers, as shown here:

```
for num in range(10):
  print("num: ", num)
```

The output from the preceding code snippet is here:

```
num:  0
num:  1
num:  2
num:  3
num:  4
num:  5
num:  6
num:  7
num:  8
num:  9
```

We can also create a list of values from the `range()` function, as shown here:

```
x = list(range(10))
print(x)
```

The output from the preceding code snippet is here:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

As you can see, when you invoke the `list()` function as a "wrapper" around the `range()` method, the result is a list of values. Hence, `list()` is different from the `range()` function. We can find the type of the `range()` function with this code snippet:

```
x = range(10)
print(type(x))
```

In essence, you can think of the `range()` function as a function you can call repeatedly to obtain successive values from an underlying sequence of numbers, thereby eliminating the need to store a list of numbers. As an example, the following code snippet initializes the variable `x` with a list of 100 numbers, and therefore requires memory allocation:

```
x = list(range(100))
print(x)
```

By contrast, the following code snippet does not create a list or allocate memory:

```
x = range(100)
print(x)
```

The following code block requires memory only for the variables `i` and `j` :

```
j = 0
for i in range(100):
  j += i

print(j)
```

However, the following code block requires memory for `i`, `j`, and the list of 100 integers:

```
j = 0
for i in list(range(100)):
  j += i
print(j)
```

FUNCTIONAL PROGRAMMING: THE MAP() FUNCTION

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

The `map()` function is a built-in function that applies a function to each item in an iterable. Specifically, 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 2.2 displays the content of `map1.py` that illustrates how to use the `map()` function.

LISTING 2.2: 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 2.2 starts with the 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 6 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` with 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 2.2 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 2.3 displays the content of `map2.py` that illustrates how to use the `map()` function.

LISTING 2.3: 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 2.3 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 2.2: 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` as a list of the

integer-based remainder of division by 4 of all the values in `numbers`. The output from Listing 2.3 is as follows:

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

Listing 2.4 displays the content of `map3.py` that illustrates how to define a lambda expression with the `map()` function to display the uppercase version of a list of strings.

LISTING 2.4: 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 2.4 involves strings instead of integers and initializes the variable `iterator` in a slightly different manner than what you saw in Listing 2.2 and Listing 2.3. 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 2.4 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 2.5 displays the content of `map4.py` that illustrates how to invoke the `map()` function with the `upper()` function to display the lowercase version of a list of strings.

LISTING 2.5: 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 2.5 is a variant of Listing 2.4: 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 is *not* a function invocation) as the first parameter to the `map()` function. The output from Listing 2.5 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 2.6 displays the content of `map5.py` that illustrates how to calculate the mean of a 2D array of random values.

LISTING 2.6: 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 2.6 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 2.6 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 2.7 displays the content of `map6.py` that illustrates how to invoke the `map()` function with an array of string values.

LISTING 2.7: 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 2.7 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()`, which returns a single string, via the `join()` function, based on the strings in the variable `friends`, which 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`, which is initialized with the result of invoking the built-in `map()` function with the parameters `single_line` and `friends`. The output from Listing 2.7 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 2.8 displays the content of `map_lambda_cond.py` that illustrates how to invoke the `map()` function and a lambda function with conditional logic.

LISTING 2.8: 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 2.8 is similar to Listing 2.7 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 2.8 is as follows:

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

Listing 2.9 displays the content of `simple_comprehension.py` that illustrates how to achieve the same results as Listing 2.8 without using a `map()` function or lambda expression.

LISTING 2.9: 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 2.9 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 2.9 is as follows:

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

FUNCTIONAL PROGRAMMING: THE FILTER() FUNCTION

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

The *filter* function enables you to extract a subset of values based on conditional logic. Listing 2.10 displays the content of `filter1.py` that illustrates how to use the `filter()` function.

LISTING 2.10: 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 2.10 initializes the variable `numbers` with a list of integers. It then initializes the variable `even_vals` as a list of values that is returned by the `filter()` function, which uses a lambda expression to return only even integers from the integers in the variable `numbers`. The output from Listing 2.10 is as follows:

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

Combining the filter() and map() Functions

Listing 2.11 displays the content of `filter_map1.py` that illustrates how to combine the `filter()` and `map()` functions.

LISTING 2.11: 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()

div4_squared_nums = filter(lambda num: num % 4 == 0, map(lambda num: num
** 2, range(1,10)))
print("div4 squared numbers:")
print(list(div4_squared_nums))
```

Listing 2.11 defines the variable `square_even_nums` that returns the square of the even numbers in the range of integers from 1 to 9, inclusive. Next, the variable `div4_squared_nums` is initialized as the list of the squares of numbers between 1 and 9, inclusive, that are a multiple of 4. The output from Listing 2.11 is as follows:

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

div4 squared numbers:
[4, 16, 36, 64]
```

Listing 2.12 displays the content of `filter_map2.py` that illustrates how to combine the `filter()` and `map()` functions.

LISTING 2.12: 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 2.12 is similar to Listing 2.11, with the new criterion that specifies numbers that are multiples of 4 instead of multiples of 2. The output from Listing 2.12 is as follows:

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

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

The 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 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 the 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.

WHAT IS A PIPE?

A pipe is a logical concept, which is to say it is not physical object. By way of illustration, consider the following scenario:

1. Person A stands inside a truck and gives boxes to person B outside the truck.
2. Person A can only provide one box at a time.
3. Person B can only move one box at a time (let's say to the backyard).

The preceding scenario can be described as a producer/consumer or as a writer/reader:

1. The producer puts "stuff" in the pipe and the consumer reads that "stuff."
2. If the pipe is full, the producer waits for the consumer to read "stuff."
3. If the pipe is empty, the consumer waits for the producer to put "stuff" in the pipe.

One variation of the preceding scenario includes the writer placing a single item in the pipe and then waiting until the reader removes the item from the pipe. Another variation involves writing to a pipe and reading from a pipe at both "ends" of the pipe.

With the preceding concepts in mind, you are in a position to understand how Python generators work, which is the topic of the next section.

WORKING WITH GENERATORS

A Python generator function enables you to pause a function and then resume the function later by means of the `yield` statement. An example of a generator is the `range()` function but not the `list()` function. You can think of a generator function as a "producer," and the code that invoked the generator function is the "consumer." Hence, the consumer uses a pull-based mechanism to obtain values from the producer. Moreover, a generator produces values until it is completed or the `yield` statement is encountered, which is discussed in the next section.

Please read the following for more information:

https://realpython.com/introduction-to-python-generators/

https://www.pythontutorial.net/advanced-python/python-generators/

The Yield Keyword

Let's consider what happens when a `return` statement appears in the body of a function: the function is exited and the state of any variables in that function is not retained.

By contrast, a generator function *does* retain the state. Specifically, the `yield` keyword "emits" a value to the "calling" code and then pauses the execution of the generator function. The next time that the calling code invokes the generator function, the code in that generator functions resumes at the location of the `yield` keyword. By way of analogy, the `yield` keyword acts like a writer to a pipe: the writer writes ("emits") a value to the pipe, and then pauses execution. When the reader reads the value in the pipe, the writer writes ("emits") another value to the pipe (an imperfect analogy, but you get the idea).

To illustrate the functionality of the `yield` keyword, Listing 2.13 displays the content of `yield1.py` that illustrates how to use the `yield` keyword to generate a list of integers.

LISTING 2.13: yield1.py

```
def make_numbers(m):
  i = 0
  while i < m:
    yield i
    i += 1
```

```
for i in make_numbers(10):
  print("i is now", i)

x = make_numbers(5)
print(type(x))
```

Listing 2.13 starts with the definition of the custom Python generator `make_numbers()` that contains a loop with a `yield` statement that "emits" the current value of the variable `i` (which is initialized as 0) to the `for` statement that invokes this generator.

The next portion of Listing 2.13 is a loop that invokes the `make_numbers()` generator 10 times, followed by a code snippet that initializes the variable `x` and displays its type. Launch the code in Listing 2.13, and you will see the following output:

```
i is now 0
i is now 1
i is now 2
i is now 3
i is now 4
i is now 5
i is now 6
i is now 7
i is now 8
i is now 9
```

Generators and Comprehensions

The previous section showed you how to define a generator, and this section contains an example of a generator. As a reminder, the following code snippet is an example of a list comprehension that generates the squares of the integers between 0 and 9, inclusive:

```
x = [i**2 for i in range(10)]
```

Now replace the square brackets with parentheses in the preceding comprehension to define a generator comprehension:

```
y = (i**2 for i in range(10))
print(y)
```

The next code block shows you how to define a generator in conjunction with the `sum()` function to calculate the sum of the squares of the integers between 0 and 99, inclusive:

```
def square_numbers(m):
    i = 0
    while i < m:
        yield i**2
        i += 1

x = sum(square_numbers(100))
print(f"Sum of the first hundred square numbers {x}")
```

The output of the preceding code block is shown here:

```
Sum of the first hundred square numbers 328350
```

A Generator Without a Loop

Listing 2.14 displays the content of `simple_gen.py` that illustrates how to "step" through different portions of code in a custom Python generator.

LISTING 2.14: simple_gen.py

```
def simple_gen():
  print("First time")
  yield 1
  print("Second time")
  yield 2
  print("Third time")
  yield 3

result = simple_gen()
#print("result:",result)
print("result:",next(result))
print("result:",next(result))
print("result:",next(result))
print("----------\n")

print("=> for loop:")
result = simple_gen()
for ndx in range(0,3):
  print("result:",next(result))
```

Listing 2.14 contains a generator that yields three integers (1, 2, and 3) during each invocation of this generator. The three invocations occur in the middle portion of Listing 2.14. The next portion of Listing 2.14 initializes the variable `result`, which is a generator object.

The final portion of Listing 2.14 contains a loop that invokes `next(result)` to summon the generator. When `next()` is invoked, Python will invoke the `__next__()` method on the function that you pass in as a parameter. Launch the code in Listing 2.14, and you will see the following output:

```
First time
result: 1
Second time
result: 2
Third time
result: 3
----------

=> for loop:
First time
result: 1
Second time
result: 2
Third time
result: 3
```

MISCELLANEOUS EXAMPLES OF GENERATORS

The following subsections contain an assortment of code samples that are also generators. You will see examples of generating the squares of integers, generating a list of integers, and finding prime numbers. Now let's proceed to the next section that shows you how to generate the squares of numbers.

Generate Squares of Numbers

Listing 2.15 displays the content of `gen_squares.py` that illustrates how to use the `yield keyword` to determine the prime numbers in a list of integers.

LISTING 2.15: gen_squares.py

```
def square_numbers(m):
  i = 0
  while i < m:
    yield i**2
    i += 1

max = 5
print("=> squares of integers:")
for value in square_numbers(max):
   print("value:",value)

print("=> sum of squares:")
the_sum = sum(square_numbers(max))
print(f"Sum of squared integers: {the_sum}")

# invoke the next() built-in function
# to execute the body of the function:
print("=> single invocation:")
answer = square_numbers(5)
print("answer:",answer)
print("answer:",next(answer))
```

Listing 2.15 starts with the definition of the generator `square_numbers()` that contains a loop with a `yield` statement that "emits" the square of the variable `I` that is initialized with the value 0. The next portion of Listing 2.15 contains a loop that invokes the `square_numbers()` generator and then prints the value that is "emitted" by this generator.

The next portion of Listing 2.15 initializes the variable `the_sum` with the sum of the squared values that are returned by the generator `square_numbers()`. The final portion of Listing 2.15 initializes the variable `answer` with the result of invoking `square_numbers(5)`. In addition, the values of `answer` and `next(answer)` are displayed. Launch the code in Listing 2.15, and you will see the following output:

```
=> squares of integers:
value: 0
value: 1
value: 4
value: 9
```

```
value: 16
=> sum of squares:
Sum of squared integers: 30
=> single invocation:
answer: <generator object square_numbers at 0x10094d040>
answer: 0
```

Generate an Infinite List of Integers

The code sample in this section can be used to generate an arbitrarily large list of integers simply by changing the value of max_value. Listing 2.16 displays the content of `gen_infinite.py` that illustrates how to generate an arbitrarily large list of integers.

LISTING 2.16: gen_infinite.py

```
def infinite_integers():
  num = 0
  while True:
    yield num
    num += 1

max_value = 20
for num in infinite_integers():
  print("num:",num)
  # removing the 'if' statement
  # generates an infinite list:
  if(num > max_value):
    break
```

Listing 2.16 defines the generator function `infinite_integers()` that initializes the variable `num` with 0, and then enters a loop. As you can see, the loop consists of two code snippets involving a yield statement and a code snippet to increment the value of the variable `num`. Notice there is no conditional logic for exiting the loop: this logic is part of the loop that invokes the function `infinite_integers()`.

The next portion of Listing 2.16 contains a loop that invokes `infinite_integers()` and prints the value "emitted" by the generator function. The other portion of the loop is conditional logic that exits the loop when the value of `num` exceeds the variable `max_value` (initialized as 20). Launch the code in Listing 2.16, and you will see the following output:

```
num: 0
num: 1
num: 2
num: 3
num: 4
num: 5
num: 6
num: 7
num: 8
num: 9
num: 10
num: 11
```

```
num: 12
num: 13
num: 14
num: 15
num: 16
num: 17
num: 18
num: 19
num: 20
num: 21
```

Find Prime Numbers

Listing 2.17 displays the content of `primes1.py` that illustrates how to use the `yield` keyword to determine the prime numbers in a list of integers.

LISTING 2.17: primes1.py

```
def primeNumbers(n):
  if n < 2: return
  numbers = list(range(2, n+1))
  print("entering while loop:")

  while numbers:
    prime = numbers[0]
    #print("prime:",prime)
    yield prime

    numbers = [num for num in numbers if num % prime !=0]
    print("numbers:",numbers)

for num in primeNumbers(30):
    print("=> generator returned prime:",num)
```

Listing 2.17 starts with the function `primes()` that finds prime numbers in the range of 2 and n, where n is initialized with the value 30 (later in the code). In addition, this function initializes the variable `numbers`, which is a list of numbers in the range 2 through n inclusive.

Next, this function contains a loop that processes the variable `prime`, which is the first element in the variable `numbers`. The next code snippet in the loop is a yield statement that "emits" the current value of variable `prime`. Let's examine what happens when the value of `prime` is "emitted," which is described in the next paragraph.

The second portion of Listing 2.17 is a `for` loop that invokes the generator function `primeNumbers()`, which receives the value "emitted" by the `yield` statement in the generator function `primeNumbers()`. After the "emitted" value is printed, the loop invokes the function `primeNumbers()` again, at which point the function regenerates a comprehension of integers, as shown here:

```
numbers = [num for num in numbers if num % prime !=0]
```

The result of regenerating the comprehension numbers is that only prime numbers will be printed. Launch the code in Listing 2.17, and you will see the following output:

```
entering while loop:
=> generator returned prime: 2
numbers: [3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29]
=> generator returned prime: 3
numbers: [5, 7, 11, 13, 17, 19, 23, 25, 29]
=> generator returned prime: 5
numbers: [7, 11, 13, 17, 19, 23, 29]
=> generator returned prime: 7
numbers: [11, 13, 17, 19, 23, 29]
=> generator returned prime: 11
numbers: [13, 17, 19, 23, 29]
=> generator returned prime: 13
numbers: [17, 19, 23, 29]
=> generator returned prime: 17
numbers: [19, 23, 29]
=> generator returned prime: 19
numbers: [23, 29]
=> generator returned prime: 23
numbers: [29]
=> generator returned prime: 29
numbers: []
```

Incidentally, Listing 2.17 is an implementation of the Sieve of Eratosthenes, which is one of the oldest algorithms for finding prime numbers.

CLOSURES

Python allows you to define a function inside another function. The innermost function is called a *nested function*, an example of which is shown here:

```
def response():
    msg = 'Hello'

    def display_msg():
        print(msg)

    return display_msg()

response()
```

As you can see in the preceding code block, the `response()` function contains another function called `display_msg()`, which in turn prints the value of the string `msg`. Notice that `msg` is defined inside `response()`, but outside of the `display_msg()` function. Thus, the combination of the function `display_msg()` and the variable `msg` is the closure. Of course, a closure can reference more than one variable from its enclosing scope.

Notice that invoking the `response()` function returns the `display_msg()` function, and the latter prints the value of the variable `msg`, which is the string `Hello`.

DECORATORS

Python functions are actually objects. Hence, a function can be assigned to a variable, an example of which is shown here:

```
def func1():
  print("inside func1")
x = func1
```

Moreover, a function can contain another function, and the latter can be returned to the code that invoked the outer function. We can now describe a *decorator* as a function that has the following properties:

- It is a function that returns another function (i.e., the closure).
- Such a function can accept a function as an argument.
- The closure can accept positional arguments and keyword-only arguments.
- The inner function references the argument from its enclosing scope or the decorator function.

Examples of Decorators

The following code block defines an outer function `func1()` that contains an inner function `func2()`:

```
def func1():
  print("inside func1")
  name="abc"

  def func2():
    print("name:",name)

func1()
```

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

```
inside func1
```

Consider the modified version of the preceding code block whereby the inner function is returned to the calling function:

```
def func1():
  print("inside func1")
  name="abc"

  def func2():
    print("name:",name)
```

```
    return func2()
func1()
```

The preceding code sample is a decorator whereby invoking the outer function results in the execution of the inner function, as shown here:

```
inside func1
name: abc
```

The next example shows you an example of a decorator in which a function takes another function as an argument.

```
def func2():
  print("inside func2")
  name="abc"
  print("name:",name)

def func1(myfunc):
  print("inside func1")
  myfunc()

func1(func2)
```

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

```
inside func1
inside func2
name: abc
```

IMPORTING CUSTOM MODULES

In addition to importing Standard Library modules, you can import your custom modules into other custom modules. Listing 2.19 and Listing 2.20 display the contents of `Double.py` and `call_double.py` that illustrate this functionality.

LISTING 2.19: Double.py

```
def double(num):
    return 2*num

result = double(5)
print('double 5 =', result)
```

Listing 2.19 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 2.19 from the Python interpreter or launch the program from the command line, you will see the following output:

```
double 5 = 10
```

LISTING 2.20: call_double.py

```
import Double
```

Listing 2.20 contains one line of code: an `import` statement that imports the `Double` module that is displayed in Listing 2.19. Launch Listing 2.20 from the command line. The output is shown here:

```
double 5 = 10
```

The combination of Listing 2.19 and Listing 2.20 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 2.21 and Listing 2.22 display the contents of `Triple.py` and `call_triple1.py` that illustrate how to achieve this flexibility.

LISTING 2.21: Triple.py

```
def triple(num):
    return(3*num)
```

Listing 2.21 contains only the definition of the function `triple()`, which returns 3 times its argument, and there are no invocations of that function or any `print()` statements.

LISTING 2.22: call_triple1.py

```
from Triple import triple

print('3 times 4 is:', triple(4))
print('3 times 9 is:', triple(9))
```

Launch Listing 2.22 from the command line, and you will see the following output:

```
3 times 4 is: 12
3 times 9 is: 27
```

Suppose that `triple1.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 triple1 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 *
```

COMPILED MODULES

The directory that contains the module `Triple.py` will also contain the directory `__pycache__` that contains the compiled version called `Triple.cpython-39.pyc`, which is

automatically generated by Python to improve performance. The contents of the file `Triple.cpython-39.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. 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 high-level terms, a *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. 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

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 Java, 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.

This concludes the brief overview regarding how to import custom modules.

FUNCTION ANNOTATIONS

Starting from Python 3.5, you can specify function annotations, which means that you can specify argument types and the return type in functions. For example, consider the following function that adds two numbers and returns their sum:

```
def add1(a,b):
  return a+b
```

Here is the definition of a function with function annotations that also returns the sum of two integers:

```
def add1(a:int,b:int) -> int:
  return a+b
```

The following example uses function annotations to define a function that returns the sum of two floating point numbers:

```
def add3(a:float,b:float) -> float:
  return a+b
```

Incidentally, languages such as Java and C++ require you to specify argument types and the return type of methods.

Function Annotations (1)

Python allows you to specify the definition of a function in multiple locations. For example, Listing 2.23 shows the content of `function_syntax1.py`, which contains three definitions of the function `mysum`.

LISTING 2.23: function_syntax1.py

```
def mysum(a:int, b:int) -> int:
  print("adding numbers",a,"and",b)
  return a + b

def mysum(a:str, b:str) -> str:
  print("concatenating strings",a,"and",b)
  return a + b

def mysum(a:int, b:int) -> int:
  print("adding numbers",a,"and",b)
  return a + b

result = mysum(3,5)
print("result:",result)

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

print("x1 =",x1)
print("x2 =",x2)
```

Listing 2.23 starts with the `mysum()` function that has two integer arguments and a return type of `int`. As you can see, this function calculates the sum of its arguments and then returns that sum.

The next portion of Listing 2.23 also defines the `mysum()` function, but this time the function has two string arguments and a return type of `str`. The next portion of Listing 2.23 has another copy of the original `mysum()` function that has two integer arguments and a return type of `int`.

The next portion of Listing 2.23 invokes the `mysum()` function with the values 3 and 5 and then prints the result. The final portion of Listing 2.23 initializes the values of `x1` and `x2` by invoking the `mysum()` function with a pair of integers and then again with a pair of strings, respectively. Launch the code in Listing 2.23, and you will see the following output:

```
adding numbers 3 and 5
result: 8
adding numbers 5 and 7
adding numbers Hello and Python
x1 = 12
x2 = HelloPython
```

Function Annotations (2)

Python allows you to specify the definition of a function in multiple locations. For example, Listing 2.24 shows the content of `function_syntax1.py` that contains three definitions of the function `mysum`.

LISTING 2.24: function_syntax2.py

```
def mysum(a:int, b:int) -> int:
  print("adding numbers",a,"and",b)
  return a + b

def string_cat(a:str, b:str) -> str:
  print("concatenating strings",a,"and",b)
  return a + b

result = mysum(3,5)
print("result:",result)

result = string_cat("abc","def")
print("result:",result)
```

Listing 2.24 starts with the function `mysum()` that takes two integer-valued arguments and returns an integer value, which equals the sum of its two arguments. The second function string_cat takes two string-valued arguments and returns a string value which equals the concatenation of its two arguments.

The next portion of Listing 2.24 invokes both of these functions (obviously, with different values) and prints the results. Launch the code in Listing 2.24, and you will see the following output:

```
adding numbers 3 and 5
result: 8
concatenating strings abc and def
result: abcdef
```

Function Annotations (3)

Python allows you to specify the definition of a function in multiple locations. For example, Listing 2.25 shows the content of `function_syntax3.py`, which contains the definition of the function `show_list()`.

LISTING 2.25: function_syntax3.py

```
def show_list(a:list, b:list) -> list:
  print("combining lists",a,"and",b)
  return a + b

list1 = ["this","is","list","one"]
list2 = ["list","two"]

result = show_list(list1, list2)
print("resulting list:",result)
```

Listing 2.25 starts with the function `show_list()`, which takes two list arguments and returns a list consisting of the concatenation of the two arguments. The next portion of Listing 2.25 invokes the `show_list()` function with two lists and displays the results. Launch the code in Listing 2.25, and you will see the following output:

```
combining lists ['this', 'is', 'list', 'one'] and ['list', 'two']
resulting list: ['this', 'is', 'list', 'one', 'list', 'two']
```

Function Annotations (4)

Python allows you to specify the definition of a function in multiple locations. For example, Listing 2.26 shows the content of `function_syntax4.py` that defines three functions: `execute_list()`, `mysum()`, and `string_cat()`.

LISTING 2.26: function_syntax4.py

```
def execute_list(func_list:list) -> list:
  print("executing functions in a list")
  new_list = []

  parm1 = 5
  parm2 = 8
  for func in func_list:
    result = func(parm1,parm2)
    new_list.append(result)

  return new_list

def mysum(a:int, b:int) -> int:
  print("adding numbers",a,"and",b)
  return a + b

def string_cat(a:str, b:str) -> str:
  print("concatenating strings",a,"and",b)
  return a + b

#####################################
# this list executes each function:
#list1 = [mysum(3,5), string_cat(3,5)]
# this list *references* the functions:
list1 = [mysum, string_cat]
#####################################
```

```
result = execute_list(list1)
print("result:",result)
```

Listing 2.26 starts with the function `execute_list()`, which takes a list of functions as its lone argument and returns a list consisting of the result of executing each function in the argument list.

The next portion of Listing 2.26 consists of the definition of the functions `mysum()` and `string_cat()` that you have already seen in a previous code sample. The final portion of Listing 2.26 creates a list of functions and then invokes the `execute_list()` function with that list. Launch the code in Listing 2.26, and you will see the following output:

```
executing functions in a list
adding numbers 5 and 8
concatenating strings 5 and 8
result: [13, 13]
```

Function Annotations (5)

Python allows you to specify the definition of a function in multiple locations. For example, Listing 2.27 shows the content of `function_syntax5.py`, which contains the function `execute_list` whose first argument is a list of Python functions.

LISTING 2.27: function_syntax5.py

```
def execute_list(func_list:list,my_string:str) -> list:
  print("executing functions in a list")
  new_list = []

  for func in func_list:
    print("invoking",func,"with string:",my_string)
    result = func(my_string)
    print("=> result", result)
    new_list.append(result)
  return new_list

my_string = "AbCdEf"
list1 = [str.isalnum, str.isalpha, str.isdigit, str.islower,  str.
isupper]
result = execute_list(list1,my_string)
print("result:",result)
```

Listing 2.27 starts with the function `execute_list()` that takes a list of functions as its first argument and a string as its second argument. It then returns a list consisting of the result of executing each function in the first argument with the second argument as the argument for each function in the first argument list.

The next portion of Listing 2.27 initializes the variable `my_string` with a string, and then initializes the variable `list1` with a list of functions that exist in the str class. The final portion of Listing 2.27 invokes the `execute_list()` function with the previously constructed list of functions and prints the results. Launch the code in Listing 2.27, and you will see the following output:

```
executing functions in a list
invoking <method 'isalnum' of 'str' objects> with string: AbCdEf
=> result True
invoking <method 'isalpha' of 'str' objects> with string: AbCdEf
=> result True
invoking <method 'isdigit' of 'str' objects> with string: AbCdEf
=> result False
invoking <method 'islower' of 'str' objects> with string: AbCdEf
=> result False
invoking <method 'isupper' of 'str' objects> with string: AbCdEf
=> result False
result: [True, True, False, False, False]
```

SUMMARY

This chapter started with a discussion of lambda expressions and comprehensions in Python. Next, you learned about iterators, and the difference between the `iter()` function and `__iter__()` method. You also saw iterators in conjunction with dictionaries, and examples of other iterators.

Next, you were introduced to functional programming by means of the `filter()` function and `map()` function. In addition, you learned about Python generators, which utilize the `yield` keyword to "emit" data to a code block that invoked the generator while also maintaining state until subsequent invocations of a generator. Furthermore, you saw examples of defining generators to generate an infinite list of integers and generate prime numbers.

In addition, you learned how to define Python classes (discussed in more detail in Chapter 4), and the difference between methods and functions. Finally, you saw several examples of defining functions that specify the type of arguments of a function, as well as the return type of a function.

CHAPTER 3

REGULAR EXPRESSIONS

This chapter introduces you to regular expressions, which is a powerful language feature. Since regular expressions are available in other programming languages (such as JavaScript and Java), the knowledge that you gain from the material in this chapter will be useful to you outside of Python.

This chapter contains code blocks and complete code samples, with varying degrees of complexity, that are suitable for beginners as well as people who have had limited exposure to regular expressions. In fact, you have probably used (albeit simple) regular expressions in a command line on a laptop, whether it be Windows, Unix, or Linux-based systems. In this chapter, you will learn how to define and use more complex regular expressions than the regular expressions that you have used from the command line.

The first part of this chapter shows you how to define regular expressions with digits and letters (uppercase as well as lowercase), and how to use character classes in regular expressions. You will also learn about character sets and character classes.

The second portion discusses the `re` module, which contains several useful methods, such as the `re.match()` method for matching groups of characters, the `re.search()` method to perform searches in character strings, and the `findAll()` method. You will also learn how to use character classes (and how to group them) in regular expressions.

The final portion of this chapter contains an assortment of code samples, such as modifying text strings, splitting text strings with the `re.split()` method, and substituting text strings with the `re.sub()` method.

One additional point about this chapter: you will encounter many concepts and facets of regular expressions that might make you feel overwhelmed with the density of the material if you are a novice. However, practice and repetition will help you become comfortable with regular expressions.

WHAT ARE REGULAR EXPRESSIONS?

Regular expressions are referred to as REs, regexes, or regex patterns, and they enable you to specify expressions that can match specific portions of a string. For instance, you can define a regular expression to match a single character or digit, a telephone number, a zip code, or an

email address. You can use metacharacters and character classes (defined in the next section) as part of regular expressions to search text documents for specific patterns. As you learn how to use REs, you will find other ways to use them as well.

The `re` module (added in Python 1.5) provides Perl-style regular expression patterns. Note that earlier versions of Python provided the `regex` module that was removed in Python 2.5. The `re` module provides an assortment of methods (discussed later in this chapter) for searching text strings or replacing text strings, which is similar to the basic search and replace functionality that is available in word processors (but usually without regular expression support). The `re` module also provides methods for splitting text strings based on regular expressions.

Before delving into the methods in the `re` module, you need to learn about metacharacters and character classes, which are the topic of the next section.

METACHARACTERS

Python supports a set of metacharacters, most of which are the same as the metacharacters in other scripting languages such as Perl, as well as programming languages such as JavaScript and Java. As a reminder, the complete list of metacharacters in Python is here:

```
. ^ $ * + ? { } [ ] \ | ( )
```

The meaning of the preceding metacharacters is as follows:

- `?` (matches 0 or 1): the expression `a?` matches the string `a` (but not `ab`)
- `*` (matches 0 or more): the expression `a*` matches the string `aaa` (but not `baa`)
- `+` (matches 1 or more): the expression `a+` matches `aaa` (but not `baa`)
- `^` (beginning of line): the expression `^[a]` matches the string `abc` (but not `bc`)
- `$` (end of line): `[c]$` matches the string `abc` (but not `cab`)
- (a single dot): matches any character (except newline)

Sometimes you need to match the metacharacters themselves rather than their representation, which can be done in two ways. The first way involves by "escaping" their symbolic meaning with the backslash ("\") character. Thus, the sequences `\?`, `\*`, `\+`, `\^`, `\$`, and `\.` represent the literal characters instead of their symbolic meaning. You can also "escape" the backslash character with the sequence "`\\.`" If you have two consecutive backslash characters, you need an additional backslash for each of them, which means that "`\\\\`" is the "escaped" sequence for "`\\.`"

The second way is to list the metacharacters inside a pair of square brackets. For example, `[+?]` treats the two characters "+" and "?" as literal characters instead of metacharacters. The second approach is obviously more compact and less prone to error (it is easy to forget a backslash in a long sequence of metacharacters). As you might surmise, the methods in the `re` module support metacharacters.

NOTE *The "^" character that is to the left (and outside) of a sequence in square brackets (such as `^[A-Z]`) "anchors" the regular expression to the beginning of a line, whereas the "^" character that is the first character inside a pair of square brackets negates the regular expression (such as `[^A-Z]`) inside the square brackets.*

The interpretation of the "^" character in a regular expression depends on its location in a regular expression, as shown here:

- "^[a-z]" means any string that starts with any lowercase letter
- "[^a-z]" means any string that does *not* contain any lowercase letters
- "^[^a-z]" means any string that starts with anything *except* a lowercase letter
- "^[a-z]$" means a single lowercase letter
- "^[^a-z]$" means a single character (including digits) that is *not* a lowercase letter

As a quick preview of the `re` module that is discussed later in this chapter, the `re.sub()` method enables you to remove characters (including metacharacters) from a text string. For example, the following code snippet removes all occurrences of a forward slash ("/") and the plus sign ("+") from the variable `str1`:

```
>>> import re
>>> str1 = "this string has a / and + in it"
>>> str2 = re.sub("[/]+","",str1)
>>> print('original:',str1)
original: this string has a / and + in it
>>> print('replaced:',str2)
replaced: this string has a  and + in it
```

We can easily remove occurrences of other metacharacters in a text string by listing them inside the square brackets, just as we have done in the preceding code snippet.

Listing 3.1 displays the content of `remove_metachars1.py` that illustrates how to remove other metacharacters from a line of text.

LISTING 3.1: remove_metachars1.py

```
import re

text1 = "metacharacters ? and / and + and ."
text2 = re.sub("[/\.*?=+]+","",text1)

print('text1:',text1)
print('text2:',text2)
```

The regular expression in Listing 3.1 might seem daunting if you are new to regular expressions. Let's demystify its contents by examining the entire expression and then the meaning of each character.

First of all, the term `[/\.*?=+]` matches a forward slash ("/"), a dot (".), a question mark ("?"), an equals sign ("="), or a plus sign ("+"). Notice that the dot "." is preceded by a backslash character "\." Doing so "escapes" the meaning of the "." metacharacter (which matches any single non-whitespace character) and treats it as a literal character.

Thus the term `[/\.*?=+]+` means "one or more occurrences of *any* of the metacharacters, treated as literal characters, inside the square brackets."

Consequently, the expression `re.sub("[/\.*?=+]+","",text1)` matches any occurrence of the previously listed metacharacters, and then replaces them with an empty string in the text string specified by the variable `text1`.

The output from Listing 3.1 is here:

```
text1: metacharacters ? and / and + and .
text2: metacharacters  and  and  and
```

Later in this chapter, you will learn about other functions in the `re` module that enable you to modify and split text strings.

CHARACTER SETS

A single digit in base 10 is a number between 0 and 9, inclusive, which is represented by the sequence `[0-9]`. Similarly, a lowercase letter can be any letter between a and z, which is represented by the sequence `[a-z]`. An uppercase letter can be any letter between `A` and `Z`, which is represented by the sequence `[A-Z]`.

For your convenience, the following code snippets provide a review of how to specify sequences of digits and sequences of character strings using a short-hand notation that is much simpler than specifying every matching digit:

- `[0-9]` matches a single digit
- `[0-9][0-9]` matches 2 consecutive digits
- `[0-9]{3}` matches 3 consecutive digits
- `[0-9]{2,4}` matches 2, 3, or 4 consecutive digits
- `[0-9]{5,}` matches 5 or more consecutive digits
- `^[0-9]+$` matches a string consisting solely of digits

You can define similar patterns using uppercase or lowercase letters in a way that is much simpler than explicitly specifying every lowercase letter or every uppercase letter:

- `[a-z][A-Z]` matches a single lowercase letter that is followed by 1 uppercase letter
- `[a-zA-Z]` matches any upper or lowercase letter

Working with "^" and "\"

The purpose of the "^" character depends on its context in a regular expression. For example, the following expression matches a text string that starts with a digit:

```
^[0-9].*
```

However, the following expression matches a text string that does *not* start with a digit because of the "^" metacharacter that is at the beginning of an expression in square brackets as well as the "^" metacharacter that is to the left (and outside) the expression in square brackets (which you learned in a previous note):

```
^[^0-9].*
```

Thus, the "^" character inside a pair of matching square brackets ("[]") *negates* the expression immediately to its right that is also located inside the square brackets.

The backslash ("\") allows you to "escape" the meaning of a metacharacter. Consequently, a dot "." matches a single character (except for whitespace characters), whereas the sequence "\." matches the dot "." character.

Other examples involving the backslash metacharacter are as follows:

- `\.H.*` matches the string `.Hello`
- `H.*` matches the string `Hello`
- `H.*\.` matches the string `Hello.`
- `.ell.` matches the string `Hello`
- `.*` matches the string `Hello`
- `\..*` matches the string `.Hello`

CHARACTER CLASSES

This section contains a refresher of the material that is discussed in a portion of Chapter 2. Character classes are convenient expressions that are shorter and simpler than their "bare" counterparts that you saw in the previous section. The following are some convenient character sequences that express patterns of digits and letters:

- `\d` matches a single digit
- `\w` matches a single character (digit or letter)
- `\s` matches a single whitespace (space, newline, return, or tab)
- `\b` matches a boundary between a word and a non-word
- `\n`, `\r`, and `\t` represent a newline, a return, and a tab, respectively
- `\` "escapes" any character

Based on the preceding definitions, `\d+` matches one or more digits and `\w+` matches one or more characters, both of which are more compact expressions than using character sets. In addition, we can reformulate the expressions in the previous section:

- `\d` is the same as `[0-9]` and `\D` is the same as `[^0-9]`
- `\s` is the same as `[ \t\n\r\f\v]` and it matches any non-whitespace character, whereas `\S` is the opposite (it matches `[^ \t\n\r\f\v]`)
- `\w` is the same as `[a-zA-Z0-9_]` and it matches any alphanumeric character, whereas `\W` is the opposite (it matches `[^a-zA-Z0-9_]`)

Additional examples are here:

- `\d{2}` is the same as `[0-9][0-9]`
- `\d{3}` is the same as `[0-9]{3}`
- `\d{2,4}` is the same as `[0-9]{2,4}`
- `\d{5,}` is the same as `[0-9]{5,}`
- `^\d+$` is the same as `^[0-9]+$`

The curly braces ("{}") are called quantifiers, and they specify the number (or range) of characters in the expressions that precede them.

MATCHING CHARACTER CLASSES WITH THE RE MODULE

The `re` module provides the following methods for matching and searching one or more occurrences of a regular expression in a text string:

- `match()`: Determines if the RE matches at the *beginning* of the string
- `search()`: Scans through a string, looking for *any* location where the RE matches
- `findall()`: Finds *all* substrings where the RE matches and returns them as a list
- `finditer()`: Finds all substrings where the RE matches and returns them as an iterator

NOTE *The* `match()` *function only matches the pattern to the start of a string.*

Compilation Flags

Compilation flags modify the manner in which regular expressions work, which you saw in the preceding code sample. Flags are available in the `re` module as a long name (such as `IGNORECASE`) and a short, one-letter form (such as `I`). The short form is the same as the flags in pattern modifiers in `Perl`. You can specify multiple flags by using the "|" symbol. For example, `re.I | re.M` sets both the `I` and `M` flags.

You can check the online Python documentation regarding all the available compilation flags. The next section shows you how to use the `match()` function in the `re` module.

USING THE RE.MATCH() METHOD

The `re.match()` method attempts to match the regular expression pattern in a text string (with optional flags), and it has the following syntax:

```
re.match(pattern, string, flags=0)
```

The `pattern` parameter is the regular expression that you want to match in the `string` parameter. The `flags` parameter allows you to specify multiple flags using the bitwise OR operator, represented by the pipe "|" symbol.

The `re.match()` method returns a matched object on success and `None` on failure. Use the `group(num)` or `groups()` function of the matched object to get a matched expression.

- `group(num=0)`: This method returns the entire match (or the specific subgroup `num`).
- `groups()`: This method returns all matching subgroups in a tuple (empty, if there were not any).

NOTE *The* `re.match()` *method only matches patterns from the start of a text string, which is different from the* `re.search()` *method discussed later in this chapter.*

The following code block illustrates how to use the `group()` function in regular expressions:

```
>>> import re
>>> p = re.compile('(a(b)c)de')
```

```
>>> m = p.match('abcde')
>>> m.group(0)
'abcde'
>>> m.group(1)
'abc'
>>> m.group(2)
'b'
```

Notice that the higher numbers inside the `group()` method match more deeply nested expressions that are specified in the initial regular expression.

Listing 3.2 displays the content of `match_group1.py` that illustrates how to use the `group()` function to match an alphanumeric text string and an alphabetic string.

LISTING 3.2 match_group1.py

```
import re

line1 = 'abcd123'
line2 = 'abcdefg'
mixed = re.compile(r"^[a-z0-9]{5,7}$")
line3 = mixed.match(line1)
line4 = mixed.match(line2)

print('line1:',line1)
print('line2:',line2)
print('line3:',line3)
print('line4:',line4)
print('line5:',line4.group(0))

line6 = 'a1b2c3d4e5f6g7'
mixed2 = re.compile(r"^([a-z]+[0-9]+){5,7}$")
line7 = mixed2.match(line6)

print('line6:',line6)
print('line7:',line7.group(0))
print('line8:',line7.group(1))

line9 = 'abc123fgh4567'
mixed3 = re.compile(r"^([a-z]*[0-9]*){5,7}$")
line10 = mixed3.match(line9)
print('line9:',line9)
print('line10:',line10.group(0))
```

The output from Listing 3.2 is here:

```
line1: abcd123
line2: abcdefg
line3: <_sre.SRE_Match object at 0x100485440>
line4: <_sre.SRE_Match object at 0x1004854a8>
line5: abcdefg
line6: a1b2c3d4e5f6g7
line7: a1b2c3d4e5f6g7
```

```
line8: g7
line9: abc123fgh4567
line10: abc123fgh4567
```

Notice that `line3` and `line7` involve two similar but different regular expressions. The variable `mixed` specifies a sequence of lowercase letters followed by digits, where the length of the text string is also between 5 and 7. The string `'abcd123'` satisfies all of these conditions.

However, `mixed2` specifies a pattern consisting of one or more pairs, where each pair contains one or more lowercase letters followed by one or more digits, where the length of the matching pairs is also between 5 and 7. In this case, the string `'abcd123'` and the string `'a1b2c3d4e5f6g7'` both satisfy these criteria.

The third regular expression, `mixed3`, specifies a pair such that each pair consists of zero or more occurrences of lowercase letters and zero or more occurrences of a digit, and that the number of such pairs is between 5 and 7. As you can see from the output, the regular expression in `mixed3` matches lowercase letters and digits in any order.

In the preceding example, the regular expression specified a range for the length of the string, which involves a lower limit of 5 and an upper limit of 7.

However, you can also specify a lower limit without an upper limit (or an upper limit without a lower limit).

Listing 3.3 displays the content of `match_group2.py` that illustrates how to use a regular expression and the `group()` function to match an alphanumeric text string and an alphabetic string.

LISTING 3.3: match_group2.py

```
import re

alphas = re.compile(r"^[abcde]{5,}")
line1 = alphas.match("abcde").group(0)
line2 = alphas.match("edcba").group(0)
line3 = alphas.match("acbedf").group(0)
line4 = alphas.match("abcdefghi").group(0)
line5 = alphas.match("abcdefghi abcdef")

print('line1:',line1)
print('line2:',line2)
print('line3:',line3)
print('line4:',line4)
print('line5:',line5)
```

Listing 3.3 initializes the variable `alphas` as a regular expression that matches any string that starts with one of the letters `a` through `e`, and consists of at least 5 characters. The next portion of Listing 3.3 initializes the four variables, `line1`, `line2`, `line3`, and `line4`, by means of the `alphas RE` that is applied to various text strings. These four variables are set to the first matching group by means of the expression `group(0)`.

The output from Listing 3.3 is here:

```
line1: abcde
line2: edcba
```

```
line3: acbed
line4: abcde
line5: <_sre.SRE_Match object at 0x1004854a8>
```

Listing 3.4 displays the content of `match_group3.py` that illustrates how to use a regular expression with the `group()` function to match words in a text string.

LISTING 3.4: match_group3.py

```
import re

line = "Giraffes are taller than elephants";

matchObj = re.match( r'(.*) are(\.*)', line, re.M|re.I)

if matchObj:
   print("matchObj.group()  : ", matchObj.group())
   print("matchObj.group(1) : ", matchObj.group(1))
   print("matchObj.group(2) : ", matchObj.group(2))
else:
   print("matchObj does not match line:", line)
```

The code in Listing 3.4 produces the following output:

```
matchObj.group()  :  Giraffes are
matchObj.group(1) :  Giraffes
matchObj.group(2) :
```

Listing 3.4 contains a pair of delimiters separated by a pipe ("|") symbol. The first delimiter is `re.M` for "multi-line" (this example contains only a single line of text), and the second delimiter `re.I` means "ignore case" during the pattern matching operation. The `re.match()` method supports additional delimiters, as discussed in the next section.

Options for the re.match() Method

The `match()` method supports various optional modifiers that affect the type of matching that will be performed. As you saw in the previous example, you can also specify multiple modifiers separated by the `OR` ("|") symbol. Additional modifiers that are available for `RE` are shown here:

- `re.I` performs case-insensitive matches (see previous section)
- `re.L` interprets words according to the current locale
- `re.M` makes $ match the end of a line and makes ^ match the start of any line
- `re.S` makes a period (".") match any character (including a newline)
- `re.U` interprets letters according to the Unicode character set

Experiment with these modifiers by writing code that uses them in conjunction with different text strings.

MATCHING CHARACTER CLASSES WITH THE RE.SEARCH() METHOD

As you saw earlier in this chapter, the `re.match()` method only matches from the beginning of a string, whereas the `re.search()` method can successfully match a substring anywhere in a text string.

The `re.search()` method takes two arguments, a regular expression pattern and a string, and then searches for the specified pattern in the given string. The `search()` method returns a match object (if the search was successful) or `None`. As a simple example, the following searches for the pattern `tasty`, followed by a five-letter word:

```
import re

str = 'I want a tasty pizza'
match = re.search(r'tasty \w\w\w\w\w', str)

if match:
  ## 'found tasty pizza'
  print('found', match.group())
else:
  print('Nothing tasty here')
```

The output of the preceding code block is here:

```
found tasty pizza
```

The following code block further illustrates the difference between the `match()` and `search()` methods:

```
>>> import re
>>> print(re.search('this', 'this is the one').span())
(0, 4)
>>>
>>> print(re.search('the', 'this is the one').span())
(8, 11)
>>> print(re.match('this', 'this is the one').span())
(0, 4)
>>> print(re.match('the', 'this is the one').span())
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'NoneType' object has no attribute 'span'
```

MATCHING CHARACTER CLASSES WITH THE FINDALL() METHOD

Listing 3.5 displays the content of the file `reg_ex1.py` that illustrates how to define simple character classes that match various text strings.

LISTING 3.5: reg_ex1.py

```
import re

str1 = "123456"
```

```
matches1 = re.findall("(\d+)", str1)
print('matches1:',matches1)

str1 = "123456"
matches1 = re.findall("(\d\d\d)", str1)
print('matches1:',matches1)

str1 = "123456"
matches1 = re.findall("(\d\d)", str1)
print('matches1:',matches1)

print()
str2 = "1a2b3c456"
matches2 = re.findall("(\d)", str2)
print('matches2:',matches2)

print()
str2 = "1a2b3c456"
matches2 = re.findall("\d", str2)
print('matches2:',matches2)

print()
str3 = "1a2b3c456"
matches3 = re.findall("(\w)", str3)
print('matches3:',matches3)
```

Listing 3.5 contains simple regular expressions for matching digits in the variables `str1` and `str2`. The final code block of Listing 3.5 matches every character in the string `str3`, effectively "splitting" `str3` into a list where each element consists of one character. The output from Listing 3.5 is here (notice the blank lines after the first three output lines):

```
matches1: ['123456']
matches1: ['123', '456']
matches1: ['12', '34', '56']

matches2: ['1', '2', '3', '4', '5', '6']

matches2: ['1', '2', '3', '4', '5', '6']

matches3: ['1', 'a', '2', 'b', '3', 'c', '4', '5', '6']
```

Finding Capitalized Words in a String

Listing 3.6 displays the content of the file `find_capitalized.py` that illustrates how to define simple character classes that match various text strings.

LISTING 3.6: find_capitalized.py

```
import re

str = "This Sentence contains Capitalized words"
caps = re.findall(r'[A-Z][\w\.-]+', str)
```

```
print('str: ',str)
print('caps:',caps)
```

Listing 3.6 initializes the string variable `str` and the regular expression `caps` that matches any word that starts with a capital letter because the first portion of `caps` is the pattern `[A-Z]`, which matches any capital letter between `A` and `Z` inclusive. The output of Listing 3.6 is here:

```
str:  This Sentence contains Capitalized words
caps: ['This', 'Sentence', 'Capitalized']
```

ADDITIONAL MATCHING FUNCTIONS FOR REGULAR EXPRESSIONS

After invoking any of the methods `match(), search(), findAll(), or finditer()`, you can invoke additional methods on the matching object. An example of this functionality using the `match()` method is here:

```
import re

p1 = re.compile('[a-z]+')
m1 = p1.match("hello")
```

In the preceding code block, the `p1` object represents the compiled regular expression for one or more lowercase letters, and the matching object, `m1`, supports the following methods:

- `group()` returns the string matched by the RE
- `start()` returns the starting position of the match
- `end()` returns the ending position of the match
- `span()` returns a tuple containing the (start, end) positions of the match

As a further illustration, Listing 3.7 displays the content of the file `search_function1.py` that illustrates how to use the `search()` and `group()` methods.

LISTING 3.7: search_function1.py

```
import re

line = "Giraffes are taller than elephants";

searchObj = re.search( r'(.*) are(\.*)', line, re.M|re.I)

if searchObj:
   print("searchObj.group()  : ", searchObj.group())
   print("searchObj.group(1) : ", searchObj.group(1))
   print("searchObj.group(2) : ", searchObj.group(2))
else:
   print("searchObj does not match line:", line)
```

Listing 3.7 contains the variable line that represents a text string and the variable `searchObj` is an `RE` involving the `search()` method and a pair of pipe-delimited modifiers (discussed in

more detail in the next section). If searchObj is not null, the if/else conditional code in Listing 3.7 displays the content of the three groups resulting from the successful match with the contents of the variable line. The output from Listing 3.7 is here:

```
searchObj.group()  :  Giraffes are
searchObj.group(1) :  Giraffes
searchObj.group(2) :
```

GROUPING WITH CHARACTER CLASSES IN REGULAR EXPRESSIONS

In addition to the character classes that you have seen earlier in this chapter, you can specify sub-expressions of character classes. Listing 3.8 displays the content of grouping1.py that illustrates how to use the search() method.

LISTING 3.8: grouping1.py

```
import re

p1 = re.compile('(ab)*')
print('match1:',p1.match('abababababab').group())
print('span1: ',p1.match('abababababab').span())

p2 = re.compile('(a)b')
m2 = p2.match('ab')
print('match2:',m2.group(0))
print('match3:',m2.group(1))
```

Listing 3.8 starts by defining the RE p1 that matches zero or more occurrences of the string ab. The first print() statement displays the result of using the match() function of p1 (followed by the group() function) against a string, and the result is a string. This illustrates the use of "method chaining," which eliminates the need for an intermediate object (as shown in the second code block). The second print() statement displays the result of using the match() function of p1, followed by applying the span() function, against a string. In this case, the result is a numeric range (see output below).

The second part of Listing 3.8 defines the RE p2 that matches an optional letter a followed by the letter b. The variable m2 invokes the match method on p2 using the string ab. The third print() statement displays the result of invoking group(0) on m2, and the fourth print() statement displays the result of involving group(1) on m2. Both results are substrings of the input string ab.

Recall that group(0) returns the highest level match that occurred, and group(1) returns a more "specific" match that occurred, such as one that involves the parentheses in the definition of p2. The higher the value of the integer in the expression group(n), the more specific the match. The output from Listing 3.8 is here:

```
match1: abababababab
span1:  (0, 10)
match2: ab
match3: a
```

USING CHARACTER CLASSES IN REGULAR EXPRESSIONS

This section contains some examples that illustrate how to use character classes to match various strings and delimiters to split a text string. For example, one common date string involves a date format of the form `MM/DD/YY`. Another common scenario involves records with a delimiter that separates multiple fields. Usually, such records contain one delimiter, but as you will see, Python makes it easy to split records using multiple delimiters.

Matching Strings with Multiple Consecutive Digits

Listing 3.9 displays the content of the file `match_patterns1.py` that illustrates how to define simple regular expressions to split the contents of a text string based on the occurrence of one or more consecutive digits.

Although the regular expressions `\d+/\d+/\d+` and `\d\d/\d\d/\d\d\d\d` both match the string `08/13/2014`, the first regular expression matches more patterns than the second regular expression, which is an "exact match" with respect to the number of matching digits that are allowed.

LISTING 3.9: match_patterns1.py

```
import re

date1 = '02/28/2013'
date2 = 'February 28, 2013'

# Simple matching: \d+ means match one or more digits
if re.match(r'\d+/\d+/\d+', date1):
  print('date1 matches this pattern')
else:
  print('date1 does not match this pattern')

if re.match(r'\d+/\d+/\d+', date2):
  print('date2 matches this pattern')
else:
  print('date2 does not match this pattern')
```

The output from launching Listing 3.9 is here:

```
date1 matches this pattern
date2 does not match this pattern
```

Reversing Words in Strings

Listing 3.10 displays the content of the file `reverse_words1.py` that illustrates how to reverse a pair of words in a string.

LISTING 3.10: reverse_words1.py

```
import re

str1 = 'one two'
match = re.search('([\w.-]+) ([\w.-]+)', str1)
```

```
str2 = match.group(2) + ' ' + match.group(1)
print('str1:',str1)
print('str2:',str2)
```

The output from Listing 3.10 is here:

```
str1: one two
str2: two one
```

Now that you understand how to define regular expressions for digits and letters, let's look at some more sophisticated regular expressions.

For example, the following expression matches a string that is any combination of digits, uppercase letters, or lowercase letters (i.e., no special characters):

```
^[a-zA-Z0-9]$
```

Here is the same expression rewritten using character classes:

```
^[\w\W\d]$
```

MODIFYING TEXT STRINGS WITH THE RE MODULE

The `re` module contains several methods for modifying strings. The `split()` method uses a regular expression to "split" a string into a list. The `sub()` method finds all substrings where the regular expression matches, and then replaces them with a different string.

The method `subn()` performs the same functionality as `sub()`, and also returns the new string and number of replacements. The following subsections contain examples illustrating how to use the functions `split()`, `sub()`, and `subn()` in regular expressions.

SPLITTING TEXT STRINGS WITH THE RE.SPLIT() METHOD

Listing 3.11 displays the content of the file `reg_ex2.py` that illustrates how to define simple regular expressions to split the contents of a text string.

LISTING 3.11: reg_ex2.py

```
import re

line1 = "abc def"
result1 = re.split(r'[\s]', line1)
print('result1:',result1)

line2 = "abc1,abc2:abc3;abc4"
result2 = re.split(r'[,:;]', line2)
print('result2:',result2)

line3 = "abc1,abc2:abc3;abc4 123 456"
result3 = re.split(r'[,:;\s]', line3)
print('result3:',result3)
```

Listing 3.11 contains three blocks of code, each of which uses the `split()` method in the `re` module to tokenize three different strings. The first regular expression specifies a whitespace, the second regular expression specifies three punctuation characters, and the third regular expression specifies the combination of the first two regular expressions. The output from launching `RegEx2.py` is here:

```
result1: ['abc', 'def']
result2: ['abc1', 'abc2', 'abc3', 'abc4']
result3: ['abc1', 'abc2', 'abc3', 'abc4', '123', '456']
```

SPLITTING TEXT STRINGS USING DIGITS AND DELIMITERS

Listing 3.12 displays the content of `split_char_class1.py` that illustrates how to use regular expression consisting of a character class, the "." character, and a whitespace to split the contents of two text strings.

LISTING 3.12: split_char_class1.py

```
import re

line1 = '1. Section one 2. Section two 3. Section three'
line2 = '11. Section eleven 12. Section twelve 13. Section thirteen'

print(re.split(r'\d+\. ', line1))
print(re.split(r'\d+\. ', line2))
```

Listing 3.12 contains two text strings that can be split using the same regular expression `'\d+\. '` . Note that if you use the expression `'\d+\. '` , only the *first* text string will split correctly. The result of launching Listing 3.12 is here:

```
['', 'Section one ', 'Section two ', 'Section three']
['', 'Section eleven ', 'Section twelve ', 'Section thirteen']
```

SUBSTITUTING TEXT STRINGS WITH THE RE.SUB() METHOD

Earlier in this chapter, you saw a preview of using the `sub()` method to remove all the metacharacters in a text string. The following code block illustrates how to use the `re.sub()` method to substitute alphabetic characters in a text string.

```
>>> import re
>>> p = re.compile( '(one|two|three)')
>>> p.sub( 'some', 'one book two books three books')
'some book some books some books'
>>>
>>> p.sub( 'some', 'one book two books three books', count=1)
'some book two books three books'
```

The following code block uses the `re.sub()` method to insert a line feed after each alphabetic character in a text string:

```
>>> line = 'abcde'
>>> line2 = re.sub('', '\n', line)
>>> print('line2:',line2)
line2:
a
b
c
d
e
```

MATCHING THE BEGINNING AND END OF TEXT STRINGS

Listing 3.13 displays the content of the file `reg_ex3.py` that illustrates how to find substrings using the `startswith()` function and `endswith()` function.

LISTING 3.13: reg_ex3.py

```
import re

line2 = "abc1,Abc2:def3;Def4"
result2 = re.split(r'[,:;]', line2)

for w in result2:
  if(w.startswith('Abc')):
    print('Word starts with Abc:',w)
  elif(w.endswith('4')):
    print('Word ends with 4:',w)
  else:
    print('Word:',w)
```

Listing 3.13 starts by initializing the string `line2` (with punctuation characters as word delimiters) and regular expression `result2`, which uses the `split()` function with a comma, colon, and semi-colon as "split delimiters" to tokenize the string variable `line2`.

The output after launching Listing 3.13 is here:

```
Word: abc1
Word starts with Abc: Abc2
Word: def3
Word ends with 4: Def4
```

Listing 3.14 displays the content of the file `match_lines1.py` that illustrates how to find substrings using character classes.

LISTING 3.14: match_lines1.py

```
import re

line1 = "abcdef"
line2 = "123,abc1,abc2,abc3"
line3 = "abc1,abc2,123,456f"
```

```
if re.match("^[A-Za-z]*$", line1):
  print('line1 contains only letters:',line1)

# better than the preceding snippet:
line1[:-1].isalpha()
print('line1 contains only letters:',line1)

if re.match("^[\w]*$", line1):
  print('line1 contains only letters:',line1)

if re.match(r"^[^\W\d_]+$", line1):
  print('line1 contains only letters:',line1)
print

if re.match("^[0-9][0-9][0-9]", line2):
  print('line2 starts with 3 digits:',line2)

if re.match("^\d\d\d", line2):
  print('line2 starts with 3 digits:',line2)
print
```

Listing 3.14 starts by initializing three string variables `line1`, `line2`, and `line3`. The first regular expression contains an expression that matches any line containing uppercase or lowercase letters (or both):

```
if re.match("^[A-Za-z]*$", line1):
```

The following two snippets also test for the same thing:

```
line1[:-1].isalpha()
```

The preceding snippet starts from the right-most position of the string and checks if each character is alphabetic. The next snippet checks if `line1` can be tokenized into words (a word contains only alphabetic characters):

```
if re.match("^[\w]*$", line1):
```

The next portion of Listing 3.14 checks if a string contains three consecutive digits:

```
if re.match("^[0-9][0-9][0-9]", line2):
  print('line2 starts with 3 digits:',line2

if re.match("^\d\d\d", line2):
```

The first snippet uses the pattern `[0-9]` to match a digit, whereas the second snippet uses the expression `\d` to match a digit. The output from Listing 3.14 is here:

```
line1 contains only letters: abcdef
line1 contains only letters: abcdef
```

```
line1 contains only letters: abcdef
line1 contains only letters: abcdef
line2 starts with 3 digits: 123,abc1,abc2,abc3
line2 starts with 3 digits: 123,abc1,abc2,abc3
```

COMPOUND REGULAR EXPRESSIONS

Listing 3.15 displays the content of `match_mixedcase1.py` that illustrates how to use the pipe ("|") symbol to specify two regular expressions in the same `match()` function.

LISTING 3.15: match_mixedcase1.py

```
import re

line1 = "This is a line"
line2 = "That is a line"

if re.match("^[Tt]his", line1):
  print('line1 starts with This or this:')
  print(line1)
else:
  print('no match')

if re.match("^This|That", line2):
  print('line2 starts with This or That:')
  print(line2)
else:
  print('no match')
```

Listing 3.15 starts with two string variables, `line1` and `line2`, followed by an if/else conditional code block that checks if `line1` starts with the regular expression `[Tt]his`, which matches the string `This` as well as the string `this`.

The second conditional code block checks if `line2` starts with the string `This` or the string `That`. Notice the "^" metacharacter, which in this context anchors the regular expression to the beginning of the string. The output from Listing 3.15 is here:

```
line1 starts with This or this:
This is a line
line2 starts with This or That:
That is a line
```

COUNTING CHARACTER TYPES IN A STRING

You can use a regular expression to check whether a character is a digit, a letter, or some other type of character. Listing 3.16 displays the content of the file `CountDigitsAndChars.py` that performs this task.

LISTING 3.16: CountDigitsAndChars.py

```
import re

charCount  = 0
digitCount = 0
otherCount = 0

line1 = "A line with numbers: 12 345"

for ch in line1:
   if(re.match(r'\d', ch)):
     digitCount = digitCount + 1
   elif(re.match(r'\w', ch)):
     charCount = charCount + 1
   else:
     otherCount = otherCount + 1

print('charcount:',charCount)
print('digitcount:',digitCount)
print('othercount:',otherCount)
```

Listing 3.16 initializes three numeric counter-related variables, followed by the string variable `line1`. The next part of Listing 3.16 contains a loop that processes each character in the string `line1`. The body of the loop contains a conditional code block that checks whether the current character is a digit, a letter, or some other non-alphanumeric character. Each time there is a successful match, the corresponding counter variable is incremented. The output from Listing 3.16 is here:

```
charcount: 16
digitcount: 5
othercount: 6
```

REGULAR EXPRESSIONS AND GROUPING

You can group sub-expressions and even refer to them symbolically. For example, the following expression matches zero or one occurrence of three consecutive letters or digits:

```
^([a-zA-Z0-9]{3,3})?
```

The following expression matches a telephone number (such as 650-555-1212) in the USA:

```
^\d{3,3}[-]\d{3,3}[-]\d{4,4}
```

The following expression matches a zip code (such as `67827` or `94343-04005`) in the USA:

```
^\d{5,5}([-]\d{5,5})?
```

The following code block partially matches an email address:

```
str = 'john.doe@google.com'
  match = re.search(r'\w+@\w+', str)
  if match:
    print(match.group())  ## 'doe@google'
```

EXERCISE *Use the preceding code block as a starting point to define a regular expression for email addresses.*

SIMPLE STRING MATCHES

Listing 3.17 displays the content of the file `reg_ex4.py` that illustrates how to define regular expressions that match various text strings.

LISTING 3.17: reg_ex4.py

```
import re

searchString = "Testing pattern matches"

expr1 = re.compile( r"Test" )
expr2 = re.compile( r"^Test" )
expr3 = re.compile( r"Test$" )

if expr1.search( searchString ):
   print('"Test" was found.')

if expr2.match( searchString ):
   print('"Test" was found at the beginning of the line.')

if expr3.match( searchString ):
   print('"Test" was found at the end of the line.')

result = expr4.findall( searchString )
```

Listing 3.17 starts with the variable `searchString` that specifies a text string, followed by the regular expressions `expr1`, `expr2`, and `expr3`. The regular expression `expr1` matches the string `Test` that occurs anywhere in `searchString`, whereas `expr2` matches `Test` if it occurs at the beginning of `searchString`, and `expr3` matches `Test` if it occurs at the end of `searchString`. The output from Listing 3.17 is here:

```
"Test" was found.
"Test" was found at the beginning of the line.
```

ADDITIONAL TOPICS FOR REGULAR EXPRESSIONS

In addition to the Python-based search/replace functionality that you have seen in this chapter, you can also perform a greedy search and substitution. Perform an Internet search to learn what these features are and how to use them in Python code.

SUMMARY

This chapter showed you how to create various types of regular expressions. First, you learned how to define primitive regular expressions using sequences of digits, lowercase letters, and uppercase letters. Next, you learned how to use character classes, which are more convenient and simpler expressions that can perform the same functionality. You also learned how to use the `re` library to compile regular expressions and then use them to see if they match substrings of text strings.

CHAPTER 4

CUSTOM CLASSES

This chapter introduces you to some built-in functions, how to create custom classes, and object-oriented concepts such as inheritance and polymorphism.

The first part of this chapter discusses functions such as `filter()`, `map()`, and `reduce()`. You will also learn about `lambda` functions, which are often used in conjunction with these 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 classes.

The final portion of this chapter contains a light introduction to encapsulation, single and multiple inheritance, and polymorphism. 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 these topics to write object-oriented Python code.

With the preceding points in mind, the next section introduces you to accessibility conventions in Python.

ACCESSIBILITY CONVENTIONS

Python 3 uses the following conventions to distinguish protected variables and functions, private variables and functions, and magic methods:

- *Protected* variables contain a single underscore "_" prefix in their name.
- *Private* variables contain a double underscore "__" prefix in their name.
- *Magic* methods contain a double underscore "__" prefix and suffix in their name.

The preceding conventions do *not* prevent you from accessing any method, regardless of its accessibility type, in a `Python` class. However, accessibility rules *are* enforced in Java (among other programming languages), as shown in the following code block for variables:

```
public int x = 1;
protected int y = 2;
private int z = 3;
```

Variables and methods with different scopes have different restrictions regarding the objects that can access them, as summarized here:

- *Public* variables and methods can be accessed by any object (i.e., instances of a class).
- *Protected* variables and methods can be accessed only by an instance of the class in which they are defined *or* by instances of a subclass.
- *Private* variables and methods can be accessed only by an instance of the class in which they are defined.

The accessibility rules control the types of objects that can modify the value of a variable (or access a method).

CREATING CUSTOM CLASSES

Python 3 enables you to create a custom class in the following ways:

Method 1:
```
class MyClass:
  pass
```

Method 2:
```
class MyClass():
  pass
```

Method 3:
```
class MyClass(object):
   pass
```

Method #3 specifies the class `object`, which is a special class. In fact, all classes in Python 3 are subclasses of the `object` class, which includes the `super()` method, the `property()` method, double underscore methods, and so forth.

If you do not specify the `object` class, as shown in Method #1 and Method #2, Python 3 will "silently" insert the `object` class so that the custom class definitions will look like the definition in Method #3.

Instance Variables versus Class Variables

Instance variables are specific to each instance of a class, so they can have different values in different instances. By contrast, *class variables* are shared among all instances of a given class, and their values are the same in all instances.

```
class Company:
  # a shared class variable:
  location = "USA"

  def __init__(self, name):
     # an instance variable:
     self.name = name
```

If a custom class explicitly includes subclasses of the `object` class, then that custom class is actually subclassing `builtins.object`, which provides direct access to the built-in functions

that are available. Navigate to this link if you want more information regarding the Python `builtins` library:

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

https://www.geeksforgeeks.org/built-in-objects-in-python-builtins/

EXAMPLES OF CUSTOM CLASSES

Python supports the creation of custom classes, which is also 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 4.1 displays the content of `simple_class.py` that illustrates how to define a simple custom class called `BaseClass`.

LISTING 4.1: simple_class.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 4.1 starts with the definition of the class `BaseClass`, which contains two functions. The built-in magic method `__init__()` (discussed in Chapter 2) assigns the value 3 to the variable `x`. The `main()` function contains two `print()` statements that display the string "inside main" and the value of the variable `x`.

The next portion of Listing 4.1 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 `simple_class.py` is here:

```
inside main
x: 3
```

A Custom Class with an Iterator

As a reminder, a square number is a product of an integer with itself. For example, the square of 2 is 4 (2*2). Listing 4.2 displays the content of `square.py` that illustrates how to define a custom class called `Square` in Python that also defines the methods `__iter__()` and `__next__()`.

LISTING 4.2: square.py

```
class Square:
    def __init__(self, length):
        self.length = length
        self.current = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.current >= self.length:
            raise StopIteration
        self.current += 1
        return self.current ** 2

# two instances of the Square class:
sq1 = Square(5)
sq2 = Square(10)

# iterate through a list of Square instances:
mylist = [sq1,sq2]
for square in mylist:
  print("length:",square.length)

# define an iterator for sq1:
iter_square = iter(sq1)

for item in iter_square:
  print("sq1 item:",item)
```

Listing 4.2 starts with the definition of the `__init__()` method that involves initializing the value of the property length, which represents the length of the side of a square. The next portion of Listing 4.2 defines the methods `__iter__()` and `__next__()`, which makes the `Square` class an iterable class. Launch the code in Listing 4.2, and you will see the following output:

```
square length: 5
square length: 10
sq1 item: 1
sq1 item: 4
sq1 item: 9
sq1 item: 16
sq1 item: 25
```

A Custom Class with an Invalid Iterator

Listing 4.3 displays the content of `wrong_iter.py` that illustrates what happens when the method `__iter__()` contains an invalid definition.

LISTING 4.3: wrong_iter.py

```
class WrongIter:
  def __iter__(self):
    return "abc" # or any other scalar

w = WrongIter()
# this works:
w.__iter__()

# this throws an error:
iter(w)
```

Listing 4.3 defines the class `WrongIter`, whose `__iter__()` method returns the string "abc," which is not an iterator. Although the `__iter__()` method does not detect this error, the function `iter(obj)` will throw a type error. Launch the code in Listing 4.3, and you will see the following output:

```
#Traceback (most recent call last):

#  File "<pyshell#37>", line 1, in <module>
Traceback (most recent call last):
  File "wrong_iter.py", line 11, in <module>
    iter(w)
TypeError: iter() returned non-iterator of type 'str'
```

CONSTRUCTION AND INITIALIZATION OF OBJECTS

In a previous section, you saw an example of the `__init__()` method, 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 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 is better to close resources (such as database connections and sockets) when they are no longer needed.

NOTE *If you are a beginner, you rarely need to use the* `__new__()` *method and the* `__del__()` *method.*

There are *many* other magic methods in Python (for comparisons, numeric functions, conversions, and so forth), and an extensive list of such methods is available online:

http://docs.python.org/2/reference/datamodel.html#specialnames

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 block defines a class `Foo` with a method called `foo()`. The variable `obj` is an instance of the class `Foo`, and notice how it is possible to write `obj.foo` to obtain the result (which is 4). This functionality is possible because of the `@property` decorator.

Consequently, you can define your custom 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.*

THE METHODS `__STR__()` AND `__REPR__()`

The built-in `__str__` method allows you to specify the values of the properties of a class. Listing 4.4 displays an example of the definition of the `__str__()` method.

LISTING 4.4: my_str.py

```
def __str__(self):
  return '<%s %s (%s:%s) %s>' % (
    self.__class__.__name__, self.urlconf_name, self.app_name,
    self.namespace, self.regex.pattern)
```

The `__str__()` method allows you to specify the values of the properties of a class. Listing 4.5 displays an example of how to define the `__repr()__` method.

LISTING 4.5: my_repr.py

```
def __repr__(self):
  return '<%s %s (%s:%s) %s>' % (
    self.__class__.__name__, self.urlconf_name, self.app_name,
    self.namespace, self.regex.pattern)
```

Keep in mind the following distinction: `__repr__()` is more suitable for developers, while `__str__` is more suitable for end users.

If a Python class does not contain a `__str__()` method, the `__repr__()` method is invoked to produce a printable representation. Similarly, if a Python class does not contain a `__repr__()` method, the `__str__()` method is invoked to produce a printable representation. Additionally, invoking `print()` with the name of the class will call `__str__()` by default.

CREATING A POINT3D CUSTOM CLASS

This section contains an example of defining a custom class to represent points in three-dimensional Euclidean space. Listing 4.6 displays the content of the custom class `Point3D.py` that handles three-dimensional points in the Euclidean plane.

LISTING 4.6: Point3D.py

```
#!/usr/bin/env python

class Point3D(object):
    def __init__(self,x,y,z):
        self.x = x
        self.y = y
        self.z = z

    def x_value(self):
        return self.x

    def y_value(self):
        return self.y

    def z_value(self):
        return self.z
```

```
    def main(self,point):
        print("Contents of current point:")
        print("x:",point.x)
        print("y:",point.y)
        print("z:",point.z)

if __name__ == '__main__':
    p1 = Point3D(1,2,3)
    p1.main(p1)

    p2 = Point3D(10,20,30)
    p2.main(p2)
```

Listing 4.6 contains the definition of the `Point3D` class, which defines three functions `x_value()`, `y_value()`, and `z_value()` that return the x-coordinate, the y-coordinate, and the z-coordinate of the current point.

In addition, the `__init__()` method 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 4.6 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 `p1` and `p2`, both of which are instances of the `Point3D` class.

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.

COMPARING TWO INSTANCES OF A CUSTOM CLASS

This section extends the `Point3D` custom class in the previous section in order to compare two instances of the `Point3D` class. Listing 4.7 displays the content of the custom class `ComparePoint3D.py` that contains the additional code that is shown in bold.

LISTING 4.7: ComparePoint3D.py

```
#!/usr/bin/env python

class Point3D(object):
    def __init__(self,x,y,z):
        self.x = x
        self.y = y
        self.z = z

    def x_value(self):
        return self.x

    def y_value(self):
        return self.y

    def z_value(self):
        return self.z
```

```
        def _lt_(self, other):
            if self.x < other.x and self.y < other.y and self.z < other.z:
               return true
            else:
               return false

    def main(self,point):
        print("Contents of current point:")
        print("x:",point.x)
        print("y:",point.y)
        print("z:",point.z)

if __name__ == '__main__':
    p1 = Point3D(1,2,3)
    p1.main(p1)

    p2 = Point3D(10,20,30)
    p2.main(p2)

    if(p1 < p2): print("p1 < p2")
    elif (p2 < p1): print("p2 < p1")
    else: print("cannot be compared")
```

Listing 4.7 contains the definition of the `Point3D` class that you saw in Listing 4.6. The new code block is the method `__lt__()` that is shown in bold, which enables you to compare two instances of the `Point3D` class by performing a component-by-component comparison of the values for x, y, and z.

The other new block of code (also shown in bold) compares the two instances of `p1` and `p2` of the `Point3D` class, which determines that `p1` is less than `p2`.

THE METHODS __ADD__() AND __RADD__()

This section shows you how to add an instance of a class with a scalar value by implementing the `__add__()` method, as well as the context in which the magic method `__radd__()` becomes relevant.

For example, suppose that the variable `myobj1` is an instance of your custom class `MyObject`, and you want to perform the following addition:

```
myobj1 + 7
```

To execute the preceding code snippet, you need to implement the magic method `__add__()` in the `MyObject` class.

Since addition is commutative (i.e., you can reverse the order), it might seem plausible that you could then execute this code snippet:

```
7 + myobj1
```

However, the preceding code snippet returns a `NotImplemented` error because 7 does not "know" how to add 7 and `myobj1`. The solution involves implementing the magic method `__radd__()` in the `MyObject` class.

As you can probably surmise, similar comments apply to the following pairs of magic methods in Python:

- `__sub__()` and `__rsub__()`
- `__mul__()` and `__rmul__()`
- `__div__()` and `__rdiv__()`

CREATING AN EMPLOYEE CUSTOM CLASS

This section contains an example of defining a custom class to keep track of some employee-related information. In object-oriented programming (OOP), 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 4.8 displays the content of the custom class `employee.py`, which keeps track of an employee's first name, last name, and title.

LISTING 4.8: 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 4.8 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__()` method 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 4.8 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. 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() )
```

As you can undoubtedly guess, 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 class `Employee` that keeps track of three attributes of an employee. This section shows you how to create a custom 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 4.9 displays the content of the Python module `employees.py` that uses a list to keep track of multiple `Employee` objects, each of which represents information about a single employee.

LISTING 4.9: 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 4.9 starts with an `import` statement that imports the definition of the `Employee` class that was defined in the previous section. Next, Listing 4.9 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 4.9.

The output of Listing 4.9 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 populate a list of instances of a Python class in order to manage a set of objects, such as processing a list of 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 enabling 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.

A PYTHON ITERABLE CLASS

The following example defines a simple `Counter` class and uses the `iter()` function to get an iterator of the counter object:

```
class Counter:
    def __init__(self):
        self.__current = 0

counter = Counter()
iterator = iter(counter)
```

The preceding code raises a `TypeError` because the `counter` object is not an iterable:

```
TypeError: 'Counter' object is not iterableCode language:
```

The following code block adds the `__getitem__()` method to the `Counter` class:

```
class Counter:
    def __init__(self):
        self.current = 0

    def __getitem__(self, index):
        if isinstance(index, int):
            self.current += 1
            return self.current
```

The preceding `Counter` class is a sequence because it implements the `__getitem__()` method that returns an element based on an index. Moreover, you can invoke the `iter()` function to obtain the iterator of the counter, as shown here:

```
counter = Counter()
iterator = iter(counter)
print(type(iterator))
```

The output of the preceding code block is here:

```
<class 'iterator'>
```

Notice that Python creates and returns an iterator object. Hence, you can use the iterator object to iterate the `counter`, as shown here:

```
for _ in range(1, 4):
    print(next(iterator))
```

Listing 4.10 contains the `CounterIterator` class as part of the definition of the `Counter` class, and it also implements the `iterable` protocol.

LISTING 4.10: counter1.py

```
class Counter:
    def __init__(self):
        self.current = 0

    def __getitem__(self, index):
        if isinstance(index, int):
            self.current += 1
            return self.current

    def __iter__(self):
        return self.CounterIterator(self)

    class CounterIterator:
        def __init__(self, counter):
            self.__counter = counter

        def __iter__(self):
            return self

        def __next__(self):
            self.__counter.current += 1
            return self.__counter.currentCode
```

The preceding `Counter` class implements the `__iter__()` method, which returns an iterator that is an instance of the `CounterIterator` class. The latter is part of the definition of the `Counter` class. Moreover, the `CounterIterator` class implements both the `__iter__()` method as well as the `__next__()` method, and therefore this class supports the iterator protocol.

One more point to keep in mind: when a Python class contains the `__iter__()` method as well as the `__getitem__()` method, the `iter()` function always uses the `__iter__()` method. The preceding sentence is true about the `Counter` class, which is illustrated by the following code block:

```
counter = Counter()
# return the __iter__() method:
iterator = iter(counter)
print(type(iterator))
```

The output of the preceding code block is shown here:

```
<class '__main__.Counter.CounterIterator'>
1
2
3
```

The preceding output shows that the `iter()` function invokes the `__iter__()` method instead of `__getitem__()` method. In addition, the `__iter__()` method returns an

instance of the `CounterIterator` class, which is why `CounterIterator` is displayed in the output.

WORKING WITH LINKED LISTS

You can use Python 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 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.

CUSTOM CLASSES AND LINKED LISTS

Listing 4.11 displays the content of `LLAndList.py` that illustrates how to create a linked list where the nodes contain the values in a Python list.

LISTING 4.11: 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 4.11 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 4.11 defines the `LinkedList` class that performs some initialization in the `__init__()` method and 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 4.11 is here:

```
List of Items:
d
c
34
b
12
a
```

CUSTOM CLASSES AND DICTIONARIES

Listing 4.12 displays the content of `LLAndDict.py` that illustrates how to create a linked list where each node references a Python dictionary.

LISTING 4.12: 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 4.12 contains code that is very similar to the previous section. The difference involves the following code block that uses `dict` instead of a 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 4.12 is here:

```
List of Keys and Values:
b bbb
c ccc
a aaa
```

CUSTOM CLASSES AND PRIORITY QUEUES

In Chapter 2, you learned about the `Queue` data structure. In this section, you will see how to create and populate a priority queue with objects. Listing 4.13 displays the content of `PriorityQueue.py` that illustrates how to create a priority queue and populate the queue with instances of the custom `Task` class.

LISTING 4.13: 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 4.13 starts by initializing the variable `q`, which is an instance of the `PriorityQueue` class. Next, Listing 4.11 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 4.13 invokes the `addTasks()` method, followed by the `displayTasks()` method. The output from Listing 4.11 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
```

THE BASE CLASSES OF A GIVEN CLASS

There are several ways for finding the "ancestor" classes of a given class. For example, if class `Three` is a subclass of class `Two` and `Two` is a subclass of class `One`, then the base classes of class `Three` are class `Two` and class `One`.

One approach involves importing the `inspect` class, after which you can inspect the base classes of a given class. For example, suppose you have defined a custom class `Three`, and you want to find the base classes of the class `Three`. You can do so by invoking any of the following code snippets:

- `inspect.getmro(Three))`
- `inspect.getclasstree(inspect.getmro(Three))`
- `type.mro(Three)`

In case you are wondering, the acronym "mro" refers to *method resolution order*, which indicates the order in which a method is identified for the purpose of execution.

Listing 4.13 displays the content of `display_base_classes.py` that shows you how to display the "ancestor" classes of a given class.

LISTING 4.14: display_base_classes.py

```
print("Example 1:")
class One(object):
  pass

class Two(One):
  pass

import inspect
print("Class Hierarchy for class Two:")
```

```
print(inspect.getmro(Two))
print("----------------\n")

print("Example 2:")
def classlookup(cls):
  c = list(cls.__bases__)
  for base in c:
    c.extend(classlookup(base))
  return c

class One: pass
class Two(One): pass
class Three(Two): pass

result1 = classlookup(Three)
print("Class Hierarchy for class Three:")
print(result1)
print("----------------\n")

print("Example 3:")
result2 = inspect.getclasstree(inspect.getmro(Three))
print("Class Hierarchy for class Three:")
print(result2)
print("----------------\n")

print("Example 4:")
result3 = inspect.getclasstree(inspect.getmro(Three))
print("Class Hierarchy for class Three:")
print(result3)
print("----------------\n")

print("Example 5:")
class Three(Two, One):
    def __init__(self):
       pass

result5 = Three.__bases__
print("Class Hierarchy for class Three:")
print(result5)
print("----------------\n")

print("Example 6:")
result6 = type.mro(Three)
print("Class Hierarchy for class Three:")
print(result6)
print("----------------\n")
```

Listing 4.13 starts with the definition of a class hierarchy in which class `Three` is a subclass of class `Two`, and class `Two` is a subclass of class `One`. The remaining portion of Listing 4.13 contains several code blocks that display the base classes of the custom class `Three`. You can choose whichever technique you wish to find the base classes of a given custom class.

ENCAPSULATION

One of the main reasons for accessors and mutators in a class is for retrieving and updating, respectively, 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*.

One benefit of encapsulation involves preventing variables or methods from being accessed unintentionally, which in turn can assist in reducing inadvertent errors in your code.

Another important benefit of encapsulation is the ability to change the inner workings of a class without changing the *signature* of the API for each method, which refers to the combination of the return type and the type of the arguments (if any) of a method in a class. 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).

By way of analogy, many components in vehicle engines are sealed, which prevents people from tampering with their inner contents. As a result, sealed components are simply replaced with new working components. In fact, the same is true of laptops: if your memory chips are malfunctioning, they are replaced with new components, which is much faster and cheaper than trying to fix those malfunctioning components.

Listing 4.14 contains a rudimentary example of encapsulation whereby the algorithm search for an item in a list involves invoking private methods in this class.

LISTING 4.14: encapsulation1.py

```
class Encapsulation:
   def __init__(self,my_list):
     self.my_list = my_list
     self.sorted = False

   # invokes a private method:
   def search_list(self,item):
     self.__search_my_list(item)

   def __search_my_list(self,item):
     if(self.sorted == False):
        # perform a linear search:
        self.__linear_search(item)
     else:
        # perform a binary search:
        self.__binary_search(item)

   def __linear_search(self,item):
     # perform a linear search
     pass

   def __binary_search(self,item):
     # perform a binary search
     pass
```

```
if __name__ == '__main__':
    enc1 = Encapsulation([4,1,5,3,8])
    enc1.search_list(5)
```

Listing 4.14 contains the definition of the `Encapsulation` class whose `__init__()` method initializes the list `my_list` and `sorted` variable as `false`. Next, the `search_list()` method invokes the private method `__search_my_list()`, which contains a conditional statement to determine which method to invoke to search for a value.

Once again, remember that designating variables and methods as private is just a convention in Python, which means that those methods can be directly invoked from other objects, thereby bypassing the designated sequence of method invocations.

SINGLE INHERITANCE

There are two types of single inheritance that you will encounter in programming languages. One common type is the "classical" *class-based inheritance*, which 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*, which you will encounter in functional languages, such as JavaScript.

Class mechanisms in Python are slightly closer to those in C++, partly because both support multiple inheritance (discussed in the next section). 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 functions (and methods) have side effects, such as modifying the value of global or static variables. In XSLT (Extensible Stylesheet Language Transformations), all variables are treated as read-only variables, which eliminates the problems associated with side effects. However, 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 4.15 displays the content of `SingleInherit1.py` that illustrates inheritance in Python.

LISTING 4.15: 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 4.15 defines `ClassA` with a `print()` statement in the `__init__()` method 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__()` method also contains a `print()` statement.

The final code block in Listing 4.15 instantiates `instanceA` of `ClassA` and `instanceB` of `ClassB`, followed by some `print()` statements. The output from Listing 4.15 is here:

```
Hello from A
Hello from B

instanceA:
Hello again from A
instanceB:
Hello again from A
```

AN EXAMPLE OF INHERITANCE

Listing 4.16 displays the content of `FoodPreferences.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 4.16: FoodPreferences.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 4.16 defines `FoodPrefs` with a `print()` statement in the `__init__()` method simply to show that this method has been executed later in the code.

The next portion of Listing 4.16 defines the `Vegetarian` class as a subclass of `FoodPrefs`, which contains the methods `add()`, `remove()`, `show_list()`, and `blocked_list()` to add, remove, display items, and specify invalid items, respectively, for a vegetarian diet.

The next portion of Listing 4.16 defines the `Carnivore` class as a subclass of `FoodPrefs`, which also contains the methods `add()`, `remove()`, and `show_list()` 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 4.16 instantiates `veggie` of type `Vegetarian` and `carnie` of type `Carnivore`, followed by the invocation of several methods. Launch the code in Listing 4.16, 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. However, 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. As you can probably surmise, *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 ClassA and ClassB *both* contain a method called func() that is *not* defined in ClassC. Now consider the following code snippet, where the variable c is an instance of ClassC:

```
c = ClassC()
c.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 executes 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 an example, Listing 4.17 displays the content of the file `MultipleInherit1.py` that illustrates multiple inheritance.

LISTING 4.17: 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 4.17 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 4.17 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, *polymorphism* allows you to define methods that "accept" instances of different classes (as parameters) and yet perform the intended calculations correctly.

As another example, Listing 4.18 displays the content of `Polymorphism1.py` that defines two custom Python classes and a method that can be invoked with instances of both custom Python classes.

LISTING 4.18: 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 4.18 starts with the definition of the custom 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 4.18 defines `instanceR` and `instanceS` that are instances of the custom classes `Rect` and `Square`, respectively. The `calcPerimeter()` method is invoked with each of these instances, and the correct method is invoked in both cases. The output from Listing 4.18 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 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 at the following site:

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

THE `abc` MODULE

Although Python does not provide interfaces (such as Java) or contracts, the `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 ABC module is available online:

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

SUMMARY

This chapter started with a description of accessibility conventions for variables and methods in Python classes, after which you saw how to create custom classes. You also saw an example of a class with a valid iterator and a custom class with an invalid `__iter__()` method.

Next, you learned how instances of a class are initialized, along with a description of accessors, mutators, and the `@property` decorator. Then, you learned about the magic methods `__str__()` and `__repr()__`, and the circumstances under which they are invoked.

In addition, you learned how to define the code in the `__lt__()` magic method to compare two instances of a custom class, as well as an example of how to create a simple linked list. Then you saw how to work with custom classes in conjunction with linked lists, dictionaries, and priority queues.

Finally, you learned about object-oriented concepts such as encapsulation, single inheritance, multiple inheritance, and polymorphism.

CHAPTER 5

QUEUES AND STACKS

This chapter introduces you to queues and stacks, along with several Python code samples that show you basic functionality involving these two data structures.

The first part of this chapter explains the concept of a queue, along with code samples that show you how to perform various operations on a queue. Some of the code samples also contain built-in functions for queues, such as `isEmpty()`, `isFull()`, `push()`, and `dequeue()`.

The second part of this chapter explains the concept of a stack, along with code samples that show you how to perform various operations on a stack. In addition, you will see code samples for finding the largest and smallest elements in a stack and reversing the contents of a stack.

The final section contains three interesting tasks that illustrate the usefulness of a stack data structure. The first task determines whether a string consists of well-balanced round parentheses, square brackets, and curly braces. The second task parses an arithmetic expression that can perform addition, subtraction, multiplication, or division, as well as any combination of these four arithmetic operations. The third task converts infix notation to postfix notation.

WHAT IS A QUEUE?

A *queue* consists of a collection of objects that uses the FIFO (first-in-first-out) rule for inserting and removing items. By way of analogy, consider a toll booth: the first vehicle that arrives is the first vehicle to pay the necessary toll and also the first vehicle to exit the tool booth. As another analogy, consider customers standing in a line (which is a queue) in a bank: the person at the front of the queue is the first person to approach an available teller. The "back" of the queue is the person at the end of the line (i.e., the last person).

A queue typically has a maximum size `MAX` and a minimum size of 0, along with the following methods:

1. `isEmpty()` returns `True` if the queue is empty
2. `isFull()` returns `True` if the queue is full
3. `queueSize()` returns the number of elements in the stack
4. `add(item)` adds an element to the back of the queue if the queue is not full
5. `dequeue()` removes the front element of the queue if the queue is not empty

To ensure that we do not exceed the capacity of the queue, we must invoke the `isFull()` method before "pushing" an item onto the queue and invoke the `isEmpty()` method before "popping" an element from the queue.

Types of Queues

The following section contains various types of queues that can be created, most of which are extensions of a generic queue, followed by a brief description:

- Queue
- Circular Queue
- Dequeue
- Priority Queue

A *queue* functions like a linear list that supports deletion from one end and insertion at the other end. The term *enqueue* refers to adding an element to a queue, whereas *dequeue* refers to removing an element from a queue.

A *circular queue* is a linear list with the following constraint: the last element in the queue "points" to the first element in the queue. A circular queue is also called a *ring buffer*. By way of analogy, a conga line is a queue: if the person at the front of the queue is "connected" to the last person in the conga line, that is called a *circular queue*.

A *dequeue* is a linear list that is also a double-ended queue in which insertions and deletions can be performed at *both* ends of the queue. In addition, there are two types of dequeues:

- *Input restricted* means that insertions occur only at one end.
- *Output restricted* means that deletions occur only at one end.

A *priority queue* is queue that allows for removing and inserting items in any position of the queue. For example, the scheduler of the operating system of your desktop and laptop uses a priority queue to schedule programs for execution. Consequently, a higher priority task is executed before a lower priority task.

Moreover, after a priority queue is created, it is possible for a higher priority task to arrive: in this scenario, that new and higher priority task is inserted into the appropriate location in the queue for task execution. In fact, Unix has a `nice` command line utility to lower the execution priority of tasks.

Now let's turn our attention to creating a basic queue along with some simple enhancements, which is the topic of the next several sections.

CREATING A QUEUE USING A PYTHON LIST

Listing 5.1 displays the contents of the file `myqueue.py` that illustrates how to use a `List` class to define functions to perform various operations on a queue.

LISTING 5.1: myqueue.py

```
import numpy as np

MAX = 4 # 100
```

```
myqueue = list()

def isEmpty():
 return len(myqueue) == 0

def isFull():
  return len(myqueue) == MAX

def dequeue():
  if myqueue:
    front = myqueue.pop(0)
    print("returning front:",front)
    return front
  else:
    print("* myqueue is empty *")

def push(item):
  if isFull() == False:
    myqueue.append(item)
  else:
    print("* myqueue is full *")

print("pushing values onto myqueue:")
push(10)
print("myqueue:",myqueue)
push(20)
print("myqueue:",myqueue)
push(200)
print("myqueue:",myqueue)
push(50)
print("myqueue:",myqueue)
push(-123)
print("myqueue:",myqueue)
print("myqueue:",myqueue)
print()

print("dequeue values from myqueue:")
dequeue()
print("myqueue:",myqueue)
dequeue()
print("myqueue:",myqueue)
dequeue()
print("myqueue:",myqueue)
dequeue()
print("myqueue:",myqueue)
dequeue()
print("myqueue:",myqueue)
```

Listing 5.1 starts by initializing the variable `myqueue` as an empty list and assigning the value 4 to the variable `MAX`, which is the maximum number of elements that the queue can contain (obviously, you can change this value).

The next portion of Listing 5.1 defines several functions: the `isEmpty` function that returns `true` if the length of `myqueue` is 0 (and false otherwise), followed by the function `isFull()` that returns `true` if the length of `myqueue` is `MAX` (and `False` otherwise).

The next portion of Listing 5.1 defines the function `dequeue()` that invokes the `pop()` method to remove the front element of `myqueue`, provided that `myqueue` is not empty. Next, the function `push()` invokes the `append()` method to add a new element to the end of `myqueue`, provided that `myqueue` is not full.

The final portion of Listing 5.1 invokes the `push()` function to append various numbers to `myqueue`, followed by multiple invocations of the `dequeue()` method to remove elements from the front of the queue. Launch the code in Listing 5.1, and you will see the following output:

```
pushing values onto myqueue:
myqueue: [10]
myqueue: [10, 20]
myqueue: [10, 20, 200]
myqueue: [10, 20, 200, 50]
* myqueue is full *
myqueue: [10, 20, 200, 50]
myqueue: [10, 20, 200, 50]

dequeue values from myqueue:
returning front: 10
myqueue: [20, 200, 50]
returning front: 20
myqueue: [200, 50]
returning front: 200
myqueue: [50]
returning front: 50
myqueue: []
* myqueue is empty *
myqueue: []
```

Listing 5.2 displays the content of the file `myqueue2.py` that illustrates how to define a queue and perform various operations on the queue.

LISTING 5.2: myqueue2.py

```
import numpy as np

MAX = 4 # 100
myqueue = list()

def isEmpty():
 return len(myqueue) == 0

def isFull():
  return len(myqueue) == MAX

def dequeue():
  if myqueue:
    front = myqueue.pop(0)
    print("returning front:",front)
    return front
  else:
    print("* myqueue is empty *")
```

```
def push(item):
  if isFull() == False:
    myqueue.append(item)
  else:
    print("* myqueue is full *")

arr1 = np.array([10,20,200,50,-123])

print("pushing values onto myqueue:")
for num in range(0,len(arr1)):
  push(num)
  print("myqueue:",myqueue)

print("dequeue values from myqueue:")
while(len(myqueue) > 0):
  dequeue()
  print("myqueue:",myqueue)
```

Listing 5.2 starts by initializing `myqueue` as an empty list and assigning the value 4 to the variable `MAX`, which is the maximum number of elements that the queue can contain (obviously, you can specify a much larger value).

The next portion of Listing 5.2 defines several functions: the `isEmpty` function that returns `True` if the length of `myqueue` is 0 (and `False` otherwise), followed by the function `isFull` that returns `True` if the length of `myqueue` is `MAX` (and `False` otherwise).

The next portion of Listing 5.2 defines the function `dequeue()` that invokes the `pop()` method to remove the front element of `myqueue`, provided that `myqueue` is not empty. Next, the function `push()` invokes the `append()` method to add a new element to the back of `myqueue`, provided that `myqueue` is not full.

The final portion of Listing 5.2 invokes the `push()` function to append various numbers to `myqueue`, followed by multiple invocations of the `dequeue()` method to remove elements from the front of the queue. Launch the code in Listing 5.2, and you will see the same output at Listing 5.1.

CREATING A ROLLING QUEUE

Listing 5.3 appends and deletes elements from a queue, but we can make the code even simpler by combining a push and delete operation in the same function. Listing 5.3 displays the content of the file `rolling_queue.py` that illustrates how to make sure that it is always possible to insert an element as the first element in a queue.

LISTING 5.3: rolling_queue.py

```
import numpy as np

MAX = 5 # maximum queue size
myqueue = list()

def isEmpty():
 return len(myqueue) == 0
```

```
def isFull():
  return len(myqueue) == MAX

def dequeue():
  if myqueue:
    front = myqueue.pop(0)
    print("returning front:",front)
    return front
  else:
    print("* myqueue is empty *")

def push(item):
  if isFull() == True:
    # remove last item:
    last_item = myqueue.pop()
    print("removed last item: ",last_item)

  # add new front item:
  myqueue.insert(0,item)
  print("new first item: ",item," queue: ",myqueue)

max = 100 # the number of elements for the queue
arr1 = [i for i in range(0,max)]

print("pushing values onto myqueue:")
for num in range(0,len(arr1)):
  push(num)
  #print("myqueue:",myqueue)

print("dequeue values from myqueue:")
while(isEmpty() == False):
  dequeue()
  print("myqueue:",myqueue)
```

Listing 5.3 is similar with Listing 5.2, along with a simple modification: if the queue is full, the `push()` method removes the final element of the queue and then inserts an element as the new first element of the queue. If need be, you can compare the code shown in bold in Listing 5.3 with the corresponding code in Listing 5.2. Launch the code in Listing 5.3, and you will see the following output:

```
=> pushing values onto myqueue:
new first item:  0  queue:  [0]
new first item:  1  queue:  [1, 0]
new first item:  2  queue:  [2, 1, 0]
new first item:  3  queue:  [3, 2, 1, 0]
new first item:  4  queue:  [4, 3, 2, 1, 0]
removed last item:  0
new first item:  5  queue:  [5, 4, 3, 2, 1]
removed last item:  1
new first item:  6  queue:  [6, 5, 4, 3, 2]
removed last item:  2
```

```
new first item:  7  queue:  [7, 6, 5, 4, 3]
removed last item:  3
new first item:  8  queue:  [8, 7, 6, 5, 4]
removed last item:  4
new first item:  9  queue:  [9, 8, 7, 6, 5]
removed last item:  5
new first item:  10  queue:  [10, 9, 8, 7, 6]

// details omitted for brevity

new first item:  99  queue:  [99, 98, 97, 96, 95]
=> dequeue values from myqueue:
returning front: 99
myqueue: [98, 97, 96, 95]
returning front: 98
myqueue: [97, 96, 95]
returning front: 97
myqueue: [96, 95]
returning front: 96
myqueue: [95]
returning front: 95
myqueue: []
```

CREATING A SHIFTING QUEUE USING A LIST

Listing 5.4 displays the content of the file `shifting_queue_list.py` that illustrates how to use a Python `list` class to define a shifting queue.

LISTING 5.4: shifting_queue_list.py

```
import numpy as np

MAX = 6 # 100
myqueue = [None] * MAX
print("myqueue:",myqueue)
print()
lpos = 2
rpos = 4

myqueue[2] = 222
myqueue[3] = 333
print("manually inserted two values:")
print("myqueue:",myqueue)

def isEmpty():
 return lpos == rpos

def isFull():
  return rpos >= MAX

def dequeue():
  global lpos,rpos
```

```
    if (lpos < rpos):
      front = myqueue[lpos]
      print("dequeued value:",front)
      myqueue[lpos] = None
      lpos += 1
      return front
    else:
      print("* myqueue is empty *")

def shift_left(myqueue):
  global lpos, rpos

  for i in range(0,rpos-lpos):
    myqueue[i] = myqueue[lpos+i]

  # replace right-most element with None:
  for i in range(rpos-lpos,rpos):
    #print("updating pos:",i)
    myqueue[i] = None

  print("Completed myqueue shift:",myqueue)
  rpos -= lpos
  lpos = 0
  return myqueue

def push(myqueue, item):
  global lpos, rpos

  if isFull() == False:
    print("rpos=",rpos,"pushing item onto myqueue:",item)
    myqueue[rpos] = item
    rpos += 1
  else:
    if(lpos == 0):
      print("*** myqueue is full: cannot push item:",item)
      print()
    else:
      print()
      print("Call shift_left to shift myqueue")
      print("before shift:",myqueue)
      print("left shift count:",lpos)
      myqueue = shift_left(myqueue)
      print("rpos=",rpos,"pushing item:",item)

      # now push the current item:
      print("rpos=",rpos,"Second try: pushing item onto myqueue:",item)
      myqueue[rpos] = item
      rpos += 1
  return myqueue

arr1 = np.array([1000,2000,8000,5000,-1000])

print("=> Ready to push the following values onto myqueue:")
print(arr1)
print()
```

```
for i in range(0,len(arr1)):
  myqueue = push(myqueue,arr1[i])
  if isFull() == False:
    print("appended",arr1[i],"to myqueue:",myqueue)

print("=> Ready to dequeue values from myqueue:")
while(lpos < rpos):
  dequeue()
  print("lpos:",lpos,"rpos:",rpos)
  print("popped myqueue:",myqueue)
```

Listing 5.4 starts by initializing the variables `MAX` (for the maximum size of the queue) and `myqueue` (which is an array-based queue), along with the integers `lpos` and `rpos` that are the index positions of the first element and the last element, respectively, of the queue.

The next portion of Listing 5.4 defines the familiar functions `isEmpty()` and `isFull()` that you have seen in previous code samples. However, the `dequeue()` function has been modified to handle cases in which elements are popped from `myqueue`: each time this happens, the variable `lpos` is incremented by 1. Note that this code block is executed only when `lpos` is less than `rpos`: otherwise, the queue is empty.

The function `shift_left()` is invoked when `lpos` is greater than 0 and `rpos` equals `MAX`: this scenario occurs when there are open "slots" at the front of the queue and the right-most element is occupied. This function shifts all the elements toward the front of the queue, thereby freeing up space so that more elements can be appended to `myqueue`. Every element in the array is occupied when `lpos` equals 0 and `rpos` equals `MAX`, at which point the only operation that we can perform is to remove an element from the front of the queue.

The final portion of Listing 5.4 initializes the NumPy array `arr1` with a set of integers, followed by a loop that iterates through the elements of `arr1` and invokes the `push()` function to append those elements to `myqueue`. When this loop finishes execution, another loop invokes the `dequeue()` function to remove elements from the front of the queue.

Change the value of `MAX` so that its value is less than, equal to, or greater than the number of elements in the array `arr1`. Doing so will exhibit different execution paths in the code. Note that numerous `print()` statements are included in Listing 5.4 to show you the execution flow of the code (later you can "comment out" those statements). Launch the code in Listing 5.4, and you will see the following output:

```
myqueue: [None, None, None, None, None, None]

manually inserted two values:
myqueue: [None, None, 222, 333, None, None]
=> Ready to push the following values onto myqueue:
[ 1000  2000  8000  5000 -1000]

rpos= 4 pushing item onto myqueue: 1000
appended 1000 to myqueue: [None, None, 222, 333, 1000, None]
rpos= 5 pushing item onto myqueue: 2000

Call shift_left to shift myqueue
before shift: [None, None, 222, 333, 1000, 2000]
```

```
left shift count: 2
Completed myqueue shift: [222, 333, 1000, 2000, None, None]
rpos= 4 pushing item: 8000
rpos= 4 Second try: pushing item onto myqueue: 8000
appended 8000 to myqueue: [222, 333, 1000, 2000, 8000, None]
rpos= 5 pushing item onto myqueue: 5000
*** myqueue is full: cannot push item: -1000

=> Ready to dequeue values from myqueue:
dequeued value: 222
lpos: 1 rpos: 6
popped myqueue: [None, 333, 1000, 2000, 8000, 5000]
dequeued value: 333
lpos: 2 rpos: 6
popped myqueue: [None, None, 1000, 2000, 8000, 5000]
dequeued value: 1000
lpos: 3 rpos: 6
popped myqueue: [None, None, None, 2000, 8000, 5000]
dequeued value: 2000
lpos: 4 rpos: 6
popped myqueue: [None, None, None, None, 8000, 5000]
dequeued value: 8000
lpos: 5 rpos: 6
popped myqueue: [None, None, None, None, None, 5000]
dequeued value: 5000
lpos: 6 rpos: 6
popped myqueue: [None, None, None, None, None, None]
```

CREATING AN ITERABLE QUEUE

In Chapter 2, you learned about iterators and how to define custom classes that are iterable. Listing 5.5 displays the content of the file `iterable_queue.py` that shows you how to define an iterable queue, which involves defining the two methods that are shown in bold.

LISTING 5.5: iterable_queue.py

```
# Implement the __iter__() and __next__() methods in
# order to make a custom Queue class that is iterable.

class Queue:
    def __init__(self):
        self.items = []

    def enqueue(self, item):
        self.items.append(item)

    def dequeue(self):
        if not self.is_empty():
            return self.items.pop(0)

    def front(self):
        if not self.is_empty():
            return self.items[0]
```

```
    def is_empty(self):
        return len(self.items) == 0

    def size(self):
        return len(self.items)

    def __iter__(self):
        self.index = 0
        return self

    def __next__(self):
        if self.index < len(self.items):
            item = self.items[self.index] self.index += 1
            return item
        else:
            raise StopIteration

# Example usage
my_queue = Queue()
my_queue.enqueue(1)
my_queue.enqueue(2)
my_queue.enqueue(3)

for item in my_queue:
    print(item)
```

Listing 5.5 contains the `__iter__()` method initializing an `index` variable to 0, and the `__next__()` method returns the elements one by one, starting from the front of the queue, while incrementing the index. When the index reaches the size of the queue, it raises a `StopIteration` exception, which indicates the end of the iteration. Launch the code in Listing 5.5, and you will see the following output:

```
1
2
3
```

This concludes the portion of the chapter pertaining to queues. The remainder of this chapter discusses the stack data structure, which is based on a LIFO structure instead of a FIFO structure.

WHAT IS A STACK?

In general terms, a *stack* consists of a collection of objects that use the LIFO (last-in-first-out) principle. By contrast, a queue follows the FIFO (first-in-first-out) principle, which you learned earlier in this chapter.

As a simple example, consider an elevator that has one entrance: the last person who enters the elevator is the first person who exits the elevator. Thus, the order in which people exit an elevator is the reverse of the order in which people enter an elevator.

Another analogy that might help you understand the concept of a stack is the stack of plates in a cafeteria:

1. a plate can be added to the top of the stack if the stack is not full
2. a plate can be removed from the stack if the stack is not empty

Based on the preceding observations, a stack has a maximum size `MAX` and a minimum size of 0.

Use Cases for Stacks

The following list contains applications and use cases for stack-based data structures:

- Recursion
- Keeping track of function calls
- Evaluation of expressions
- Reversing characters
- Servicing hardware interrupts
- Solving combinatorial problems using backtracking

Operations with Stacks

Earlier in this chapter, you saw Python functions to perform operations on queues; in an analogous fashion, we can define a stack in terms of the following methods:

- `isEmpty()` returns `true` if the stack is empty
- `isFull()` returns `true` if the stack is full
- `stackSize()` returns the number of elements in the stack
- `push(item)` adds an element to the "top" of the stack if the stack is not full
- `pop()` removes the top-most element of the stack if the stack is not empty

To ensure that we do not exceed the capacity of the stack, we must invoke the `isFull()` method before "pushing" an item onto the stack, and invoke the `isEmpty()` method before "popping" an element from the stack.

WORKING WITH STACKS

Listing 5.6 displays the content of the file `mystack.py` that illustrates how to define a stack and perform various operations on the stack.

LISTING 5.6: mystack.py

```
import numpy as np

MAX = 3 # 100
mystack = list()

def isEmpty():
 return len(mystack) == 0
```

```
def isFull():
  return len(mystack) == MAX

def pop():
  if len(mystack) > 0:
    top = mystack.pop()
    #print("returning top:",top)
    return top
  else:
    print("* mystack is empty *")

def push(item):
  if isFull() == False:
    mystack.append(item)
  else:
    print("* mystack is full *")

print("pushing values onto mystack:")
push(10)
print("mystack:",mystack)
push(20)
print("mystack:",mystack)
push(200)
print("mystack:",mystack)
push(-123)
print("mystack:",mystack)
push(50)
print("mystack:",mystack)
print()

print("popping values from mystack:")
pop()
print("mystack:",mystack)
pop()
print("mystack:",mystack)
pop()
print("mystack:",mystack)
pop()
print("mystack:",mystack)
pop()
print("mystack:",mystack)
pop()
print("mystack:",mystack)
```

Listing 5.6 is very similar to Listing 5.1, except that we are working with a stack instead of a queue. In particular, Listing 5.6 starts by initializing `mystack` as an empty list and assigning the value 3 to the variable `MAX`, which is the maximum number of elements that the stack can contain (obviously, you can specify a much larger value).

The next portion of Listing 5.6 defines several functions: the `isEmpty` function that returns `True` if the length of `mystack` is 0 (and `False` otherwise), followed by the function `isFull` that returns `True` if the length of `mystack` is `MAX` (and `false` otherwise).

The next portion of Listing 5.6 defines the function `dequeue`, which invokes the `pop()` method to remove the front element of `mystack`, provided that `mystack` is not empty. Next, the function `push()` invokes the `append()` method to add a new element to the top of `mystack`, provided that `myqueue` is not full.

The final portion of Listing 5.6 invokes the `push()` function to append various numbers to `mystack`, followed by multiple invocations of the `dequeue()` method to remove elements from the top of `mystack`. Launch the code in Listing 5.6, and you will see the following output:

```
pushing values onto mystack:
mystack: [10]
mystack: [10, 20]
mystack: [10, 20, -123]
* mystack is full *
mystack: [10, 20, -123]
* mystack is full *
mystack: [10, 20, -123]

popping values from mystack:
mystack: [10, 20]
mystack: [10]
mystack: []
* mystack is empty *
mystack: []
* mystack is empty *
mystack: []
```

Listing 5.7 displays the content of the file `mystack2.py` that illustrates how to define a stack and perform various operations on the stack.

LISTING 5.7: mystack2.py

```
import numpy as np

MAX = 3 # 100
mystack = list()

def isEmpty():
 return len(mystack) == 0

def isFull():
  return len(mystack) == MAX

def pop():
  if len(mystack) > 0:
    top = mystack.pop()
    #print("returning top:",top)
    return top
  else:
    print("* mystack is empty *")

def push(item):
  if isFull() == False:
    mystack.append(item)
  else:
    print("* mystack is full *")
arr1 = np.array([10,20,-123,200,50])
```

```
print("pushing values onto mystack:")
for num in range(0,len(arr1)):
  push(num)
  print("mystack:",mystack)
print()

print("popping values from mystack:")
for num in range(0,len(arr1)):
  pop()
  print("mystack:",mystack)
print("mystack:",mystack)
```

Listing 5.7 is straightforward because it is a direct counterpart to Listing 5.2: the latter involves a queue whereas the former involves a stack. Launch the code in Listing 5.7, and you will see the following output:

```
pushing values onto mystack:
mystack: [0]
mystack: [0, 1]
mystack: [0, 1, 2]
* mystack is full *
mystack: [0, 1, 2]
* mystack is full *
mystack: [0, 1, 2]

popping values from mystack:
mystack: [0, 1]
mystack: [0]
mystack: []
* mystack is empty *
mystack: []
* mystack is empty *
mystack: []
```

CREATING AN ITERABLE STACK

In Chapter 2, you learned about iterators and how to define custom classes that are iterable. Listing 5.8 displays the content of the file `iterable_stack.py` that shows you how to define an iterable stack, which involves defining the two methods that are shown in bold.

LISTING 5.8: iterable_stack.py

```
# Implement the __iter__() and __next__() methods in
# order to make a custom Stack class that is iterable.

class Stack:
    def __init__(self):
        self.items = []

    def push(self, item):
        self.items.append(item)
    def pop(self):
```

```
        if not self.is_empty():
            return self.items.pop()

    def peek(self):
        if not self.is_empty():
            return self.items[-1]

    def is_empty(self):
        return len(self.items) == 0

    def size(self):
        return len(self.items)

    def __iter__(self):
        self.index = len(self.items)
        return self

    def __next__(self):
        if self.index > 0:
            self.index -= 1
            return self.items[self.index]
        else:
            raise StopIteration


# Example usage
my_stack = Stack()
my_stack.push(1)
my_stack.push(2)
my_stack.push(3)

for item in my_stack:
    print(item)
```

Listing 5.8 contains two new magic methods, the `__iter__()` method and `__next__()` method (both are discussed in Chapter 2 and Chapter 4), that make this an iterable Python class. Launch the code in Listing 5.8, and you will see the following output:

```
3
2
1
```

TASK: REVERSE AND PRINT STACK VALUES

Listing 5.9 displays the content of the file `reverse_stack.py` that illustrates how to define a stack and print its contents in reverse order. This code sample uses a "regular" stack data structure.

LISTING 5.9: reverse_stack.py

```
import numpy as np

MAX = 8 # 100
mystack = list()

def isEmpty():
 return len(mystack) == 0

def isFull():
  return len(mystack) == MAX

def pop():
  #print("len(mystack) =", len(mystack))
  if len(mystack) > 0:
    top = mystack.pop()
    #print("returning top:",top)
    return top
  else:
    print("* mystack is empty *")
    return None

def push(item):
  if isFull() == False:
    mystack.append(item)
  else:
    print("* mystack is full *")

arr1 = np.array([10,20,-123,200,50])

print("pushing values onto mystack:")
for i in range(0,len(arr1)):
  push(arr1[i])
  #print("mystack:",mystack)
print("mystack:",mystack)
print()

reversed = []
print("popping values from mystack:")
for num in range(0,len(arr1)):
  top = pop()
  reversed.append(top)
  #print("reversed:",reversed)
print("reversed:",reversed)
```

Listing 5.9 contains the code in Listing 5.6, along with a loop that invokes the `push()` function to insert the elements of the NumPy array `arr1` (which contains integers) in the variable `mystack`.

After the preceding loop finishes execution, another loop iterates through the elements of `mystack` by invoking the `pop()` method, and in turn appends each element to the array

reversed. As a result, the elements in the array reversed are the reverse order of the elements in `mystack`. Launch the code in Listing 5.9, and you will see the following output:

```
pushing values onto mystack:
mystack: [10, 20, -123, 200, 50]

popping values from mystack:
reversed: [50, 200, -123, 20, 10]
```

TASK: DISPLAY THE MIN AND MAX STACK VALUES

Listing 5.10 displays the content of the file `stack_min_max.py` that illustrates how to define a stack and perform various operations on the stack. This code sample uses a "regular" stack data structure.

LISTING 5.10: stack_min_max.py

```
import numpy as np

MAX = 6 # 100
mystack = [None] * MAX
lindex = 0
rindex = 0
min_val = np.Infinity
max_val = -np.Infinity

def isEmpty():
 return lindex == rindex

def isFull():
  return rindex >= MAX

def pop():
  #print("len(mystack) =", len(mystack))
  if len(mystack) > 0:
    top = mystack.pop()
    #print("returning top:",top)
    return top
  else:
    print("* mystack is empty *")

def update_min_max_values(item):
    global min_val, max_val

    if(min_val > item):
       min_val = item

    if(max_val < item):
       max_val = item
```

```
def min():
  return min_val
def max():
  return max_val

def push(mystack, item):
  global rindex

  if isFull() == False:
    #print("1rindex=",rindex,"pushing item onto mystack:",item)
    #mystack[rindex] = item
    mystack.append(item)
    rindex += 1
    #print("push 5rindex:",rindex)
    update_min_max_values(item)
  else:
    print("* mystack is full *")
    print("Cannot push item:",item)
    print("push 6rindex:",rindex)
  return mystack

arr1 = np.array([1000,2000,8000,5000,-1000])

print("pushing values onto mystack:")
for i in range(0,len(arr1)):
  mystack = push(mystack,arr1[i])
  print("mystack:",mystack)
print()

print("min value:",min_val)
print("max value:",max_val)
```

Listing 5.10 contains the familiar functions `isEmpty()`, `isFull()`, and `pop()` that have been discussed in previous code samples. Notice that the function `pop()` invokes the function `update_min_max_values()` each time that an element is removed from the stack. The latter method updates the variables `min_val` and `max_val` to keep track of the smallest and largest elements, respectively, in the stack. Launch the code in Listing 5.10, and you will see the following output:

```
=> Pushing list of values onto mystack:
[ 1000  2000  8000  5000 -1000]

min value: -1000
max value: 8000
```

CREATING TWO STACKS USING AN ARRAY

Listing 5.11 displays the contents of the file `stack_array2.py` that illustrates how to use an array to define two adjacent stacks and perform various operations on the stack.

LISTING 5.11: stack_array2.py

```
import numpy as np

MAX = 6 # 100
mystack = [None] * MAX * 2
lindex1 = 0
rindex1 = 0
lindex2 = int(MAX/2)
rindex2 = int(MAX/2)

def isEmpty(num):
 if(num == 1):
   return lindex1 == rindex1
 else:
   return lindex2 == rindex2

def isFull(num):
  if(num == 1):
    return rindex1 >= int(MAX/2)
  else:
    return rindex2 >= int(MAX)

def pop(num):
  global lindex1,rindex1, lindex2,rindex2

  if(num == 1):
    if (lindex1 <= rindex1):
      print("pop position:",rindex1)
      rear = mystack[rindex1]
      print("popped value:",rear)
      mystack[rindex1] = None
      rindex1 -= 1
      return rear
    else:
      print("* mystack is empty *")
      return None
  else:
    if (lindex2 <= rindex2):
      print("pop position:",rindex2)
      rear = mystack[rindex2]
      print("popped value:",rear)
      mystack[rindex2] = None
      rindex2 -= 1
      return rear
    else:
      print("* mystack is empty *")
      return None

def push(mystack, num, item):
  global rindex1,rindex2

  if(num == 1):
    if isFull(num) == False:
```

```
      print("1rindex1=",rindex1,"pushing item onto mystack:",item)
      mystack[rindex1] = item
      rindex1 += 1
      print("push 5rindex1:",rindex1)
    else:
      print("* mystack is full *")
      print("Cannot push item:",item)
      print("push 6rindex1:",rindex1)
    return mystack
  else:
    if isFull(num) == False:
      print("1rindex2=",rindex2,"pushing item onto mystack:",item)
      mystack[rindex2] = item
      rindex2 += 1
      print("push 5rindex2:",rindex2)
    else:
      print("* mystack is full *")
      print("Cannot push item:",item)
      print("push 6rindex2:",rindex2)
    return mystack

print("=> Pushing list of values onto mystack:")
arr1 = np.array([1000,2000,8000,5000,-1000])
print(arr1)
print()

#print("Alindex1:",lindex1,"rindex1:",rindex1)
#print("Alindex2:",lindex2,"rindex2:",rindex2)

for i in range(0,len(arr1)):
  rand = np.random.rand()
  if(rand > 0.5):
    num = 1
  else:
    num = 2

  print("=> selected stack:",num)
  mystack = push(mystack,num, arr1[i])

#print("Blindex1:",lindex1,"rindex1:",rindex1)
#print("Blindex2:",lindex2,"rindex2:",rindex2)

print("-------------------")
print("left stack:")
for idx in range(lindex1,rindex1):
  print(mystack[idx])
print()

print("right stack:")
for idx in range(lindex2,rindex2):
  print(mystack[idx])
print("-------------------")
```

Listing 5.11 defines two stacks inside the stack `mystack` (which is an array): the left stack occupies the left half of `mystack` and the right stack occupies the right half of `mystack`. In addition, the variables `lindex1` and `rindex1` are the left-most and right-most index positions for the left stack, whereas the variables `lindex2` and `rindex2` are the left-most and right-most index positions for the right stack.

Notice that the usual functions `isEmpty()`, `isFull()`, and `push()` perform their respective operations based on the currently "active" stack, which is based on the value of the variable `num`: the values 1 and 2 correspond to the left stack and right stack, respectively.

One more difference is the loop that generates random numbers and then populates the two stacks based on whether each generated random number is greater than 0.5. Launch the code in Listing 5.11, and you will see the following output:

```
=> Pushing list of values onto mystack:
[ 1000  2000  8000  5000 -1000]

=> selected stack: 2
1rindex2= 3 pushing item onto mystack: 1000
push 5rindex2: 4
=> selected stack: 1
1rindex1= 0 pushing item onto mystack: 2000
push 5rindex1: 1
=> selected stack: 1
1rindex1= 1 pushing item onto mystack: 8000
push 5rindex1: 2
=> selected stack: 2
1rindex2= 4 pushing item onto mystack: 5000
push 5rindex2: 5
=> selected stack: 1
1rindex1= 2 pushing item onto mystack: -1000
push 5rindex1: 3
--------------------
left stack:
2000
8000
-1000

right stack:
1000
5000
--------------------
```

TASK: REVERSE A STRING USING A STACK

Listing 5.12 displays the content of the file `reverse_string.py` that illustrates how to use a stack to reverse a string.

LISTING 5.12: reverse_string.py

```
import numpy as np

MAX = 20 # 100
```

```
mystack = list()
def isEmpty():
 return len(mystack) == 0

def isFull():
  return len(mystack) == MAX

def pop():
  #print("len(mystack) =", len(mystack))
  if len(mystack) > 0:
    top = mystack.pop()
    #print("returning top:",top)
    return top
  else:
    print("* mystack is empty *")
    return None

def push(item):
  if isFull() == False:
    mystack.append(item)
  else:
    print("* mystack is full *")

my_str = "abcdxyz"

#print("pushing values onto mystack:")
for i in range(0,len(my_str)):
  push(my_str[I])

#print("mystack:",mystack)
#print()

reversed = ""
#print("popping values from mystack:")
for num in range(0,len(my_str)):
  top = pop()
  reversed += top
  #print("reversed:",reversed)

print("string:  ",my_str)
print("reversed:",reversed)
```

Listing 5.12 starts by initializing `mystack` as an empty list, followed by the usual functions `isEmpty()`, `isFull()`, and `pop()` that perform their respective operations. The next portion of Listing 5.12 initializes the variable `my_str` as a string of characters, and then pushes each character onto `mystack`. Next, a loop removes each element from `mystack` and then appends each element to the string reversed (which initialized as an empty string). Launch the code in Listing 5.12, and you will see the following output:

```
string:   abcdxyz
reversed: zyxdcba
```

TASK: BALANCED PARENTHESES (1)

Listing 5.13 displays the content of the file `scalar_balanced_parens1.py` that illustrates how to use a NumPy array to determine whether a string contains balanced round parentheses.

LISTING 5.13: scalar_balanced_parens1.py

```
#####################################################
# 1) counter = 0
# 2) while input string is non-empty:
# 3)   if "(" appears in input string, counter += 1
# 4)   if a ")" appears in input string, counter -= 1
# 5) expression is balanced if counter = 0
# 6) expression is not balanced if counter != 0
#####################################################

def check_balanced(myexpr):
  counter = 0
  for ch in myexpr:
    if(ch == "("):
      counter += 1
    elif (ch == ")"):
      counter += 1

  print("Expression",myexpr,end="")
  if(counter == 0):
    print(" IS BALANCED")
  else:
    print(" IS IMBALANCED")
  print()

expr_list = ["()","()(","()()()","()))"]
for term in expr_list:
  check_balanced(term)
```

The code in Listing 5.13 is the implementation of the comment block at the beginning of Listing 5.13. The nice aspect of this simple code sample is that the function `check_balanced()` uses an integer variable to determine whether an expression contains balanced round parentheses. Thus, this iterative solution does not have the overhead of a recursive solution and the overhead of using a data structure. Launch the code in Listing 5.13, and you will see the following output:

```
Expression () IS BALANCED

Expression ()( IS IMBALANCED

Expression ()()() IS BALANCED

Expression ()) IS IMBALANCED
```

TASK: BALANCED PARENTHESES (2)

Listing 5.14 displays the content of the file `array_balanced_parens1.py` that illustrates how to use a NumPy array to determine whether a string contains balanced parentheses.

LISTING 5.14: array_balanced_parens1.py

```
import numpy as np

def check_balanced(my_expr):
  left_chars = "([{"
  right_chars = ")]}"
  balan_pairs = np.array(["()","[]", "{}"])
  my_stack = np.array([])

  for idx in range(0,len(my_expr)):
    char = my_expr[idx]
    #print("char:",char)
    if char in left_chars:
      my_stack = np.append(my_stack,char)
      #print("appended to my_stack:",my_stack)
    elif char in right_chars:
      if(my_stack.size > 0):
        top = my_stack[len(my_stack)-1]
        two_chars = top + char
        #print("=> two_chars:",two_chars)
        if(two_chars in balan_pairs):
          #print("old stack:",my_stack)
          #remove right-most element:
          my_stack = my_stack[:-1]
          if(my_stack.size == 0):
            my_stack = np.array([])
          #print("new stack:",my_stack)
          continue
        else:
          #print("non-match:",char)
          break
      else:
        print("empty stack: invalid string",my_expr)
        return False
    else:
      print("invalid character:",char)
      return False

  return (my_stack.size == 0)

# main code starts here:
expr_list = np.array(["[()]{}{[()()]()}",
                      "[()]{}{[()()]()]",
                      "(()]{}{[()()]()]",
                      "(())(()()()()()]"])
```

```
for expr in expr_list:
  if( check_balanced(expr) == True):
    print("balanced string:",expr)
  else:
    print("unbalanced string:",expr)
```

Listing 5.14 is the longest code sample in this chapter that also reveals the usefulness of combining a stack with recursion to solve the task at hand, which is to determine which strings comprise balanced parentheses.

Listing 5.14 starts with the function `check_balanced()` that takes a string called `my_expr` as its lone parameter. Notice the way that the following variables are initialized:

```
left_chars = "([{"
right_chars = ")]}"
balan_pairs = np.array(["()","[]", "{}"])
```

The variable `left_chars` and `right_chars` contain the left-side parentheses and right-side parentheses, respectively, that are permissible in a well-balanced string. Next, the variable `balan_pairs` is an array of three strings that represent a balanced pair of round parentheses, square parentheses, and curly parentheses, respectively.

The key idea for this code sample involves two actions and a logical comparison, as listed here:

1. Whenever a *left* parenthesis is encountered in the current string, this parenthesis is pushed onto a stack.
2. Whenever a *right* parenthesis is encountered in the current string, we check the top of the stack to see if it equals the corresponding left parenthesis.
3. If the comparison in 2) is true, we pop the top element of the stack.
4. If the comparison in 2) is false, the string is unbalanced.

We repeat the preceding sequence of steps until we reach the end of the string: if the stack is empty, the expression is balanced; otherwise, the expression is unbalanced.

For example, suppose the string `my_expr` is initialized as "()." The first character is "(," which is a left parenthesis: Step #1 above tells us to push "(" onto our (initially empty) stack called `mystack`. The next character in `my_expr` is ")," which is a right parenthesis: Step #2 above tells us to compare ")" with the element in the top of the stack, which is "(." Since "(" and ")" constitute a balanced pair of parentheses, we pop the element "(" from the stack. We have also reached the end of `my_expr`, and since the stack is empty, we conclude that "()" is a balanced expression (which we knew already).

Now suppose the string `my_expr` is initialized as "((." The first character is "(," which is a left parenthesis, and Step #1 above tells us to push "(" onto our (initially empty) stack called `mystack`. The next character in `my_expr` is "(," which is a left parenthesis: Step #1 above tells us to push "(" onto the stack. We have reached the end of `my_expr` and since the stack is non-empty, we have determined that `my_expr` is unbalanced.

As a third example, suppose the string `my_expr` is initialized as "(()." The first character is "(," which is a left parenthesis: Step #1 above tells us to push "(" onto our (initially empty) stack called `mystack`. The next character in `my_expr` is "(," which is a left parenthesis: Step #1 above

tells us push another "(" onto the stack. The next character in `my_expr` is "),"and Step #2 tells us to compare ")" with the element in the top of the stack, which is "(." Since "(" and ")" constitute a balanced pair of parentheses, we pop the top-most element of the stack. At this point, we have reached the end of `my_expr`, and since the stack is non-empty, we know that `my_expr` is unbalanced.

Try tracing through the code with additional strings consisting of a sequence of parentheses. After doing so, the code details in Listing 5.14 will become much simpler to understand.

The next code block in Listing 5.14 initializes the variable `my_expr` as an array of strings, each of which consists of various parentheses (round, square, and curly). The next portion is a loop that iterates through the elements of `my_expr` and in turn invokes the function `check_balanced` to determine which ones (if any) comprise balanced parentheses. Launch the code in Listing 5.14, and you will see the following output:

```
balanced string: [()]{}{[()()]()}
unbalanced string: [()]{}{[()()]()]
unbalanced string: (()]{}{[()()]()]
unbalanced string: (())(()()()()()]
```

Consider enhancing Listing 5.14 so that the invalid character is displayed for strings that consists of unbalanced parentheses.

TASK: TOKENIZE ARITHMETIC EXPRESSIONS

The code sample in this section is a prelude to the task in the next section that involves parsing arithmetic expressions. In fact, the code in Listing 5.15 is included in the code in Listing 5.16 in a subsequent section. The rationale for the inclusion of a separate code sample is to enable you to tokenize expressions that might not be arithmetic expressions.

Listing 5.15 displays the content of `tokenize_expr.py` that illustrates how to tokenize an arithmetic expression and remove white spaces and tab characters.

LISTING 5.15: tokenize_expr.py

```
# be sure to install shlex:
# pip3 install shlex
import shlex

try:
  from StringIO import StringIO
except ImportError:
  from io import StringIO

expr_list = ["          2789  *  3+7-8-  9",
             "4 /2  +  1",
             "2  -  3  +  4     "]

for expr in expr_list:
  input = StringIO(expr)
  result = list(shlex.shlex(input))
  print("string:",expr)
  print("result:",result)
```

Listing 5.15 starts with a `try/except` block that imports `StringIO` from the appropriate location, followed by the variable `expr_list` containing a list of arithmetic expressions that include extra white spaces (including non-visible tab characters).

The next portion of Listing 5.15 contains a loop that iterates through each element of `expr_list` and then invokes the `shlex.shlex` function that tokenizes each element. Launch the code in Listing 5.15, and you will see the following output:

```
string:          2789  *  3+7-8-  9
result: ['2789', '*', '3', '+', '7', '-', '8', '-', '9']
string: 4/2  +  1
result: ['4', '/', '2', '+', '1']
string: 2  -  3  +  4
result: ['2', '-', '3', '+', '4']
```

TASK: EVALUATE ARITHMETIC EXPRESSIONS

Evaluating arithmetic expressions is an interesting task because there are various ways to approach this problem. First, some people would argue that the "real" way to solve this task involves levers and parsers. However, the purpose of this task is to familiarize you with parsing strings, after which you will be better equipped to do so with non-arithmetic expressions.

Second, perhaps the simplest way involves the `eval()` function, which is a one-line solution and therefore the simplest solution. However, this solution does not familiarize you with parsing expressions.

After you have finished reading the code sample, you can enhance the code in several ways. For example, the operators "*" and "/" have equal priority, both of which have higher priority than "+" and "-" (and the latter pair have equal priority). The current code sample does not take into account this priority, which means that "2+3*4" is evaluated as 20 (which is incorrect) instead of 14 (which is the correct answer). So, one variation involves adding the priority constraint for arithmetic operators.

The current code sample does not support round parentheses, square brackets, curly braces, or exponentiation (try adding these additional features after you finish reading this section).

Listing 5.16 displays the content of the file `parse_expr.py` that illustrates how to parse and evaluate an arithmetic expression using a stack.

LISTING 5.16: parse_expr.py

```
import numpy as np
import re

try:
  from StringIO import StringIO
except ImportError:
  from io import StringIO
import shlex

# performs "num1 oper num2"
# and returns the result
def reduction(num1,num2,oper):
  num1 = float(num1)
```

```
  num2 = float(num2)
  reduced = 0.0

  #print("RED string:",num1,oper,num2)
  if( oper == "*"):
    reduced = num1*num2
  elif( oper == "/"):
    reduced = num1/num2
  elif( oper == "+"):
    reduced = num1+num2
  elif( oper == "-"):
    reduced = num1-num2
  else:
    print("Binvalid operator:",oper)
  #print("returning reduced value:",reduced)
  return reduced

# a function that finds num1 and oper
# and evaluates "num1 oper num2" and
# then appends the result to the stack
def reduce_stack(my_stack):
  num2 = my_stack[len(my_stack)-1]
  oper = my_stack[len(my_stack)-2]
  num1 = my_stack[len(my_stack)-3]
  #print("Anum1:",num1,"oper:",oper,"num2:",num2)

  # remove the right-most three elements:
  my_stack = my_stack[:-3]

  reduced = reduction(num1,num2,oper)
  my_stack = np.append(my_stack,reduced)
  #print("Creturning my_stack:",my_stack)
  return my_stack

# a function to place tokens on the stack
# and determine when to reduce the stack
def reduce_expr(my_expr):
  math_symbols = ["*","/","+","-"]
  #digits = [i for i in range(0,10)]
  digits = [str(i) for i in range(0,10)]
  my_stack = np.array([])
  oper = ""

  my_expr = strip_white_spaces(my_expr)
  for idx in range(0,len(my_expr)):
    token = my_expr[idx]
    if token in math_symbols:
      my_stack = np.append(my_stack,token)
      oper = token
    else:
      floatnum  = float(token)
      #print("found number in expr:",token)
      my_stack = np.append(my_stack,floatnum)
      if(oper != ""):
```

```
        my_stack = reduce_stack(my_stack)
      oper = ""

  return my_stack

# strip white spaces and tokenize symbols:
def strip_white_spaces(my_expr):
  expr2 = re.split(r'[ ]',my_expr)
  my_expr  = [token for token in expr2 if token != '']
  my_expr  = "".join(my_expr)

  # tokenize string with symbols and no spaces (ex: '3+7-8-')
  input = StringIO(my_expr)
  new_expr = list(shlex.shlex(input))
  #print("string:",my_expr)
  #print("result:",new_expr)
  return new_expr

expr_list = ["4 /2  +  1",
             "2  -  3  +  4  ",
             "         125  *  3+7-8-  9"]

for expr in expr_list:
  print("=> string:",expr)
  result = reduce_expr(expr)
  print("=> result:",result)
  print()
```

Listing 5.16 starts with a `try/except` block to import the `shlex` library, which will handle the "heavy lifting" in this code sample.

The next portion of Listing 5.16 contains the function `reduction`, which takes three parameters, where the first two are strings containing numbers and the third parameter is the arithmetic parameter to invoke on the first two parameters. After converting `num1` and `num2` to floating point numbers, an `if/elif` code block determines the value of `oper`, and then applies it to the other two parameters. For example, if `num1`, `num2`, and `oper` have the values 3, 4, and "*," the result is 3*4 = 12, which is returned to the calling function.

The next portion of Listing 5.16 contains the function `reduce_stack()`, which takes a single parameter, that is, our current stack. This function pops the top three values from the stack and assigns them to `num2`, `oper`, and `num1`, respectively. Next, this function invokes the function `reduction()` to determine the result operating on the two numeric values, as shown here:

```
reduced = reduction(num1,num2,oper)
my_stack = np.append(my_stack,reduced)
```

As you can see, the purpose of this function is to perform a stack reduction operation. The next portion of Listing 5.16 contains the function `reduce_expr()` that starts by initializing the following variables:

1. `math_symbols` consists of the four standard arithmetic operators
2. `digits` is assigned the digits in the range of 0 to 9
3. `my_stack` is initialized as an empty NumPy array
4. `oper` is an empty string (and assigned something in `math_symbols` later)

The next portion of the function `reduce_expr()` initializes `my_expr` as the result of invoking the function `strip_white_spaces()`, which is where the "heavy lifting" is performed in this code sample.

The next section in `reduce_expr()` contains a loop that iterates through each character called `token` in the string `my_expr`, and performs the following logic:

1. If `token` is a math symbol, append it to `my_stack` and set `oper` equal to `token`.
2. Otherwise, append the floating point version of `token` to the stack, and if `oper` is not `null`, invoke the `reduce_stack()` function (described earlier).

When the preceding loop finishes execution, the function returns the updated contents of `my_stack` to the calling function.

The important function `strip_white_spaces()` removes redundant white spaces from `my_expr` and assigns the result to `expr2`, after which `expr2` is tokenized and then used to reinitialize the contents of `my_expr`. Then, the `join()` operations concatenates all the elements of `my_expr`. At this point, we invoke `shlex()`, which returns a perfectly parsed arithmetic expression.

The final portion of Listing 5.16 initializes the variable `expr_list` as an array of arithmetic expressions, followed by a loop that invokes the function `reduce_expr` with each element of `expr_list`, and then prints the evaluated expression. Launch the code in Listing 5.16, and you will see the following output:

```
=> string: 4    /2  +  1
=> result: ['3.0']

=> string: 2  -  3  +  4
=> result: ['3.0']

=> string:             125  *  3+7-8-  9
=> result: ['365.0']
```

INFIX, PREFIX, AND POSTFIX NOTATIONS

There are three well-known and useful techniques for representing arithmetic expressions.

Infix notation involves specifying operators *between* their operands, which is the typical way we write arithmetic expressions (example: 3+4*5).

Prefix notation (also called *Polish notation*) involves specifying operators *before* their operands, an example of which is here:

3+4*5 becomes + 3 * 4 5
3+4 becomes + 3 4

Postfix notation (also called *Reverse Polish Notation*) involves specifying operators *after* their operands, an example of which is here:

3+4*5 becomes 345*+

The following table contains additional examples of expressions using infix, prefix, and postfix notation.

```
Infix      Prefix      Postfix
x+y        +xy         xy+
x-y        -xy         xy-
x/y        /xy         xy/
x*y        *xy         xy*
x^y        ^yx         yx^

(x+y)*z    *(x+y)z     (x+y)z*
(x+y)*z    *(+xy)z     (xy+)z*
```

Let's look at the following slightly more complex infix expression (note the "/" shown in bold):

```
[[x+(y/z)-d]^2]/(x+y)
```

We will perform an iterative sequence of steps to convert this infix expression to a prefix expression by applying the definition of infix notation to the top-level operator. In this example, the top-level operator is the "/" symbol (shown in bold). We need to place this "/" symbol in the left-most position, as shown here (and notice the "^" symbol, shown in bold):

```
/[[x+y/z-d]^2](x+y)
```

We need to place this "^" symbol immediately to the left of the second left square bracket, as shown here (and notice the "/," shown in bold):

```
/[^[x+(y/z)-d]2](+xy)
```

We need to place this "/" symbol immediately to the left of the variable `y`, as shown here (and notice the "+," shown in bold):

```
/[^[x+(/yz)-d]2](+xy)
```

We need to place this "+" symbol immediately to the left of the variable `x`, as shown here (and notice the "/," shown in bold):

```
/[^[+x(/yz)-d]2](+xy)
```

Now we need to place this "/" symbol immediately to the left of the variable `x`, as shown here, which is now an infix expression:

```
/[^[-(+(/yz))d]2](+xy)
```

The relative priority of arithmetic operators can be specified as follows:

```
recedence={'^':5,'*':4,'/':4,'+':3,'-':3,'(':2,')':1}
```

SUMMARY

This chapter started with an introduction to queues, which are FIFO data structures, along with real world examples of queues. Next, you learned about functions that are associated with a queue, such as `isEmpty()`, `isFull()`, `push()`, and `dequeue()`.

Then you learned about stacks, which are LIFO data structures, along with some code samples that show you how to perform various operations on stacks. Some examples include reversing the contents of a stack and determining whether the contents of a stack form a palindrome.

In the final portion of this chapter, you learned how to determine whether a string consists of well-balanced round parentheses, square brackets, and curly braces; how to parse an arithmetic expression; and how to convert infix notation to postfix notation.

CHAPTER 6

RECURSION AND COMBINATORICS

This chapter introduces you to recursion, which is illustrated in various Python code samples, followed by an introduction to basic concepts in combinatorics, such as combinations and permutations of objects.

The first part of this chapter shows you how to calculate the sum of an arithmetic series and the sum of a geometric series using iterative algorithms as well as recursive algorithms. These examples provide an introduction to recursion if you are new to this topic. Next, you will learn about calculating factorial values of positive integers as well as Fibonacci numbers.

The second part of this chapter discusses concepts in combinatorics, such as permutations and combinations. Note that a thorough coverage of combinatorics can fill an entire undergraduate course in mathematics, whereas this chapter contains only some rudimentary concepts.

If you are new to recursion, be prepared to read the material more than once and also practice working with the code samples, which will lead to a better understanding of recursion.

WHAT IS RECURSION?

In simplified terms, a function or method is recursive if it invokes itself. Recursion-based algorithms can provide 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 = S + 2
S = S + 3
. . .
S = S + 10
```

If we denote S(n) as the sum of the first n positive integers, then we have the following relationship:

```
S(1) = 1
S(n) = S(n-1) + n for n > 1
```

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 portion shows you how to calculate a closed form solution, followed by an iterative algorithm and then a recursion-based algorithm.

Here is the derivation of the closed form sum of the integers from 1 to n inclusive, which we will denote as S. As you can see below, there are two ways to calculate S:

```
S = 1 + 2     + 3     + . . . + (n-1) + n
S = n + (n-1) + (n-2) + . . . + 2     + 1
```

Since there are n columns on the right side of the preceding pair of equations, it is clear that each column has the sum equal to (n+1). 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)
```

Now divide both sides by 2 to 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 calculations manually: that student was Karl F. Gauss (in third grade).

Calculating an Arithmetic Series (Iterative)

Listing 6.1 displays the content of 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 an Arithmetic Series (Recursive)

Listing 6.2 displays the content of `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 a Partial Arithmetic Series

Listing 6.3 displays the content of `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_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 from the corresponding term in the first row to obtain the following result:

```
S - r*S = 1 - r^(n+1)
```

Factor S from both terms on the left side of the preceding equation to obtain 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 content of 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(), which 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 a Geometric Series (Recursive)

Listing 6.5 displays the content of `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. Note that the following code sample uses *tail recursion*.

LISTING 6.5: geom_sum_recursive.py

```
def geom_sum(n,ratio,term,sum):
  if(n == 1):
    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 illustrates tail recursion, which might be more intuitive because it contains each intermediate result as part of the recursive invocation of the function.

The first portion of Listing 6.5 is the function `geom_sum()` that takes four arguments: `n` (the current value of the upper range), `ratio` (which is the exponent 2 in this code sample), `term` (which is the current intermediate term of the sum), and `sum` (the target sum).

The code returns the value 1 when `n` equals 1; otherwise, the values of `term` and `sum` are updated, and the function `geom_sum()` is invoked whose *only* difference is to decrement `n` by 1.

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 sample 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 is reasonably intuitive to infer the following relationship for factorial numbers:

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

The next section uses the preceding formula to calculate the factorial value of various numbers.

Calculating Factorial Values (Iterative)

Listing 6.6 displays the content of `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()`, which 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 content of `Factorial2.py` that illustrates how to calculate factorial values using 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()`, which is the same function that you saw in Listing 6.6. Notice that the second portion of Listing 6.7 is the same as 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 content of `Factorial3.py` that illustrates how to calculate factorial values using tail recursion.

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 is 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 content of `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 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 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 iterate from 0 to `max`. 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 content of `Fibonacci2.py` that illustrates how to calculate Fibonacci numbers using an iterative approach.

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 also calculates the values of Fibonacci numbers; however, this code sample stores intermediate values in an array. Despite the overhead of an array, this code is much more efficient than the code in Listing 6.9. Launch the code in Listing 6.10, and you will see the same output as the previous section.

TASK: REVERSE A STRING VIA RECURSION

Listing 6.11 displays the content of the file `reverser.py` that illustrates how to use recursion to reverse a string.

LISTING 6.11: reverser.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 omitting 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 (ITERATIVE)

Checking for balanced parentheses involves determining whether or not a left parenthesis has a corresponding "matching" right parenthesis. This task can be performed easily using an iterative solution. The examples below contain balanced and unbalanced strings of parentheses in case you need more clarification. Moreover, this task is restricted to round parentheses: later you will see an example of checking for balanced parentheses that can include square brackets and curly braces. Here are some examples of strings that contain round parentheses:

```
S1 = "()()()"
S2 = "(()()())"
S3 = "()("
S4 = "(())"
S5 = "()()("
```

As you can see, the strings `S`, `S3`, and `S4` have balanced parentheses, whereas the strings `S2` and `S5` have unbalanced parentheses.

Listing 6.12 displays the content of the file `balanced_parens.py` that illustrates how to determine whether a string consists of balanced parentheses.

LISTING 6.12: balanced_parens.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 contents of the current character 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 parenthesis ")." 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 content of the 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 function `count_digits()` that recursively invokes itself with the term `int(num/10)`, where `num` is the input parameter. 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 the variable `result` is returned.

If the logic of this code is not clear to you, try tracing through the code with the numbers 5, 25, and 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 (ITERATIVE)

Listing 6.14 displays the content of the file `check_prime.py` that illustrates how to determine whether a positive integer is a prime number.

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 function `is_prime()` that contains a loop that checks whether 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 (ITERATIVE)

Listing 6.15 displays the content of the 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 function `is_prime()` from Listing 6.14 that determines whether a positive integer is a prime number. Next, the 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 divisor 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`. 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 3 can be expressed as the sum of two odd prime numbers.

Listing 6.16 displays the content of the 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 starts with the function `prime()` that determines whether the parameter `num` is a prime number. Next, the function `find_prime_factors()` contains a loop whose loop variable `num` 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. 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 content of the 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 function `gcd()`, which takes two parameters and repeatedly subtracts the smaller from the larger, 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. 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 content of `simple_gcd.py`, which 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 6.19 is a more compact implementation of Euclid's algorithm that achieves the same result as Listing 6.18; if the logic is unclear, review the details of Listing 6.18 to convince yourself that the logic in both code samples is the same. Launch the code in Listing 6.19, 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
```

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

Listing 6.19 displays the content of the 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,"=",int(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
The LCM of 36 and 18 = 2
The LCM of 50 and 11 = 550
The LCM of 100 and 64 = 400
The LCM of 200 and 120 = 15
```

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 be displayed on a bookshelf? 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 you can 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, C}` and the set `{B, A, 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 zeros.) 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 zeros 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: 6 choices
position #2: 5 choices
position #3: 4 choices
position #4: 3 choices
position #5: 2 choices
position #6: 1 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

Now let's look at a slightly different question: how many ways can you select three books from those six books? Here is the first approximation:

position #1: 6 choices
position #2: 5 choices
position #3: 4 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 book #1, book #2 ... book #6, and suppose that you select book #1, book #2, and book #3. Here is a list of all the different orderings of those three books:

```
123
132
213
231
312
321
```

The number of different permutations of three books is 3x2x1 = 3! = 6. However, from the standpoint of purely selecting three books, we must treat all 6 orderings as the same. Hence, the 6 orderings are indistinguishable from each other. As a result, we must divide the number of permutations by the number of orderings that are considered the same. As a result, the correct answer is `N` = 6x5x4/[3x2x1] = 120/3! = 120/6 = 20.

Consider what happens when we multiply the numerator and the denominator of the number `N` by 3x2x1:

```
N = 6x5x4/[3x2x1] = 6x5x4x3x2x1/[3x2x1 x 3x2x1] = 6!/[3! x 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!]
```

Now 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)!]
```

Given any positive integer `n`, the following result (details are in the next section) is true:

```
2^n = C(n,0)+C(n,1)+C(n,2)+. . . +C(n,n-1)+C(n,n)
```

Listing 6.20 displays the content of the 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 function `binom_coefficient()` calculates the binomial of two integers whose formula was derived in a previous section.

The third function is `sum_binomials()`, which calculates the sum of a range of binomial values by invoking the function `binom_coefficient()`, where the latter invokes the function `factorial()`. 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 it involves visualizing a row of `n` books. To find every possible subset of those `n` books, we need only consider that there are two actions that you can perform 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. The selection (or not) of a book in a given position is independent of the selection of the books in every other position. Therefore, 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 + 3*x^2*y + 3*x*y^2 + y^3
        = C(3,0)*x^3 + C(3,1)*x^2*y + C(3,2)*x*y^2 + C(3,3)*y^3
```

In general, we have the following formula:

```
             n
(x+y)^n =   SUM C(n,k)*x^k*y^(n-k)
            k=0
```

Set x=y=1 in the preceding formula to obtain 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.

TASK: SUBSETS CONTAINING A VALUE LARGER THAN K

The more complete description of the task for this section is as follows: given a set N of numbers and a number k, find the number of subsets of N that contain at least one number that is larger than k. This *counting* task is an example of a coding task that can easily be solved as a combinatorial problem: you might be surprised to discover that the solution involves a single (and simple) line of code. Let's define the following set of variables:

- N = a set of numbers
- |N| = # of elements in N (= n)
- NS = the non-empty subsets of N
- P(NS) = the number of non-empty subsets of N (= |NS|)
- M = the numbers {n| n < k} where n is an element of N
- |M| = # of elements in M (= m)
- MS = the non-empty subsets of M
- P(MS) = the number of non-empty subsets of M (= |MS|)
- Q = subsets of N that contain at least one number larger than k

Note that the set NS is partitioned into the sets Q and M, and that the union of Q and M is NS. In other words, a non-empty subset of N is either in Q or in M, but not in both. Therefore, the solution to the task can be expressed as |Q| = P(NS) - P(MS).

Moreover, the sets in M do not contain any number that is larger than k, which means that no element (i.e., subset) in M is an element of Q, and conversely, no element of Q is an element of M.

Recall from a previous result in this chapter that if a set contains m elements, then the number of subsets is 2**m, and the number of *non-empty* subsets is 2**m - 1. Hence, the answer for this task is (2**n - 1) - (2**m - 1).

Listing 6.21 displays the content of subarrays_max_k.py that calculates the sum of a set of binomial coefficients.

LISTING 6.21: subarrays_max_k.py

```
################################################
# N = a set with n elements
# M = a set with m elements
#
# N has 2^n - 1 non-empty subsets
# M has 2^m - 1 non-empty subsets
#
# O = subsets of N with at least one element > k
```

```
# P = subsets of N with all numbers <= k
#
# |P| = 2**m-1
# and |O| = |N| - |P| = (2**n-1) - (2**m-1)
#####################################################

# number of subarrays whose maximum element > k
def count_subsets(n, m):
  count = (2**n - 1) - (2**m - 1)
  return count

arr = [1, 2, 3, 4, 5]
print("contents of array: ")
for num in arr:
  print(num,end=" ")
print()

arrk = [1,2,3,4]
for overk in arrk:
  arr_len = len(arr)
  count = count_subsets(arr_len, overk)

  print("overk:   ",overk)
  print("count:   ",count)
  print("---------------")
```

Listing 6.21 contains the code that implements the details that are described at the beginning of this section. Although the set `N` in Listing 6.21 contains a set of consecutive integers from 1 to `n`, the code works correctly for unsorted arrays or arrays that do not contain consecutive integers. In the latter case, you would need a code block to count the number of elements that are less than a given value of `k`.

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.

INDEX