```
2
4
6
8
```

4.6 Multiply Lists and Arrays

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

Listing 4.5: multiply1.py

```
import numpy as np

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

Listing 4.5 contains a Python list called list and a `NumPy` array called `arr1`. The `print()` statements display the contents of list and arr1 as well as the result of doubling `list1` and `arr1`. Recall that "doubling" a Python list is different from doubling a Python array, which you can see in the output from launching Listing 4.5:

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

4.7 Doubling the Elements in a List

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

Listing 4.6: double_list1.py

```
import numpy as np

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

for e in list1:
```

```
  list2.append(2*e)

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

Listing 4.6 contains a Python list called `list1` and an empty `NumPy` list called `list2`. The next code snippet iterates through the elements of `list1` and appends them to the variable `list2`. The pair of `print()` statements display the contents of `list1` and `list2` to show you that they are the same. The output from launching Listing 4.6 is here:

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

4.8 Lists and Exponents

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

Listing 4.7: exponent_list1.py

```
import numpy as np

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

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

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

Listing 4.7 contains a Python list called `list1` and an empty `NumPy` list called `list2`. The next code snippet iterates through the elements of `list1` and appends the square of each element to the variable `list2`. The pair of `print()` statements display the contents of `list1` and `list2`. The output from launching Listing 4.7 is here:

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

4.9 Arrays and Exponents

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

Listing 4.8: exponent_array1.py

```
import numpy as np

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

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

Listing 4.8 contains a `NumPy` array called `arr1` followed by two `NumPy` arrays called `arr2` and `arr3`. Notice the compact manner in which the `NumPy` `arr2` is initialized with the square of the elements in in `arr1`, followed by the initialization of the `NumPy` array `arr3` with the cube of the elements in `arr1`. The three `print()` statements display the contents of `arr1`, `arr2`, and `arr3`. The output from launching Listing 4.8 is here:

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

4.10 Math Operations and Arrays

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

Listing 4.9: mathops_array1.py

```
import numpy as np

arr1 = np.array([1,2,3])
sqrt = np.sqrt(arr1)
log1 = np.log(arr1)
exp1 = np.exp(arr1)

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

Listing 4.9 contains a `NumPy` array called `arr1` followed by three `NumPy` arrays called `sqrt`, `log1`, and `exp1` that are initialized with the square root, the log, and the exponential value of the elements in `arr1`, respectively. The three `print()` statements display the contents of `sqrt`, `log1`, and `exp1`. The output from launching Listing 4.9 is here:

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

4.11 Working with "-1" Subranges with Vectors

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

Listing 4.10: npsubarray2.py

```
import numpy as np

# _1 => "all except the last element in …" (row or col)

arr1  = np.array([1,2,3,4,5])
print('arr1:',arr1)
print('arr1[0:_1]:',arr1[0:_1])
print('arr1[1:_1]:',arr1[1:_1])
print('arr1[::_1]:', arr1[::_1]) # reverse!
```

Listing 4.10 contains a `NumPy` array called `arr1` followed by four `print` statements, each of which displays a different subrange of values in `arr1`. The output from launching Listing 4.10 is here:

```
('arr1:',       array([1, 2, 3, 4, 5]))
('arr1[0:_1]:', array([1, 2, 3, 4]))
('arr1[1:_1]:', array([2, 3, 4]))
('arr1[::_1]:', array([5, 4, 3, 2, 1]))
```

4.12 Working with "-1" Subranges with Arrays

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

Listing 4.11: np2darray2.py

```
import numpy as np

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

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

Listing 4.11 contains a `NumPy` array called `arr1` followed by four `print` statements, each of which displays a different subrange of values in `arr1`. The output from launching Listing 4.11 is here:

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

4.13 Other Useful `NumPy` Methods

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

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

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

The `reshape()` method and the `linspace()` method are very useful for changing the dimensions of an array and generating a list of numeric values, respectively. The `reshape()` method often appears in `TensorFlow` code, and the `linspace()` method is useful for generating a set of numbers in linear regression (discussed in Chapter 4). The `mean()` and `std()` methods are useful for calculating the mean and the standard deviation of a set of numbers. For example, you can use these two methods in order to resize the values in a Gaussian distribution so that their mean is 0 and the standard deviation is 1. This process is called *standardizing* a Gaussian distribution.

4.14 Arrays and Vector Operations

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

Listing 4.12: array_vector.py

```
import numpy as np

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

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

Listing 4.12 contains two `NumPy` arrays called `a` and `b` followed by eight `print` statements, each of which displays the result of "applying" a different arithmetic operation to the `NumPy` arrays `a` and `b`. The output from launching Listing 4.12 is here:

```
('a    :   ', array([[1, 2], [3, 4]]))
('b    :   ', array([[5, 6], [7, 8]]))
('a + b:   ', array([[ 6,  8], [10, 12]]))
('a _ b:   ', array([[_4, _4], [_4, _4]]))
('a * b:   ', array([[ 5, 12], [21, 32]]))
('a / b:   ', array([[0, 0], [0, 0]]))
('b / a:   ', array([[5, 3], [2, 2]]))
('a.dot(b):', array([[19, 22], [43, 50]]))
```

4.15 NumPy and Dot Products (1)

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

Listing 4.13: dotproduct1.py

```
import numpy as np

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

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

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

Listing 4.13 contains two `NumPy` arrays called `a` and `b` followed by a simple loop that computes the dot product of a and b. The next section contains five `print` statements that display the contents of `a` and `b`, their inner product that's calculated in three different ways. The output from launching Listing 4.13 is here:

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

4.16 NumPy and Dot Products (2)

`NumPy` arrays support a "dot" method for calculating the inner product of an array of numbers, which uses the same formula that you use for calculating the inner product of a pair of vectors. Listing 4.14 displays the contents of `dotproduct2.py` that illustrates how to calculate the dot product of two `NumPy` arrays.

Listing 4.14: dotproduct2.py

```
import numpy as np
```

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

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

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

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

4.17 NumPy and the "Norm" of Vectors

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

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

Listing 4.15: array_norm.py

```
import numpy as np

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

print('a:      ',a)
print('asquare:',asquare)
```

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

Listing 4.15 contains an initial `NumPy` array called `a`, followed by the `NumPy` array `asquare` and the numeric values `asqsum`, `anorm1`, `anorm2`, and `anorm3`. The `NumPy` array `asquare` contains the square of the elements in the `NumPy` array `a`, and the numeric value `asqsum` contains the sum of the elements in the `NumPy` array `asquare`. Next, the numeric value `anorm1` equals the square root of the sum of the square of the elements in a. The numeric value `anorm2` is the same as `anorm1`, computed in a slightly different fashion. Finally, the numeric value `anorm3` is equal to `anorm2`, but as you can see, `anorm3` is calculated via a single `NumPy` method, whereas `anorm2` requires a succession of `NumPy` methods.

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

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

4.18 NumPy and Other Operations

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

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

Listing 4.16: otherops.py

```
import numpy as np
```

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

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

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

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

4.19 NumPy and the reshape() Method

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

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

Listing 4.17: numpy_reshape.py

```
import numpy as np

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

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

x = x.reshape((_1))
print(x.shape) # (6,)
print('x2:',x)
```

```
x = x.reshape((6, _1))
print(x.shape) # (6, 1)
print('x3:',x)

x = x.reshape((_1, 6))
print(x.shape) # (1, 6)
print('x4:',x)
```

Listing 4.17 contains a `NumPy` array called `x` whose dimensions are 3x2, followed by a set of invocations of the `reshape()` method that reshape the contents of `x`. The first invocation of the `reshape()` method changes the shape of x from 3x2 to 2x3. The second invocation changes the shape of `x` from 2x3 to 6x1. The third invocation changes the shape of `x` from 1x6 to 6x1. The final invocation changes the shape of `x` from 6x1 to 1x6 again.

Each invocation of the `reshape()` method is followed by a `print()` statement so that you can see the effect of the invocation. The output from launching Listing 4.17 is here:

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

4.20 Calculating the Mean and Standard Deviation

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

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

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

np.std()

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

Listing 4.18: sample_mean_std.py

```
import numpy as np

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

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

Listing 4.18 contains a NumPy array x2 that consists of the first eight integers. Next, the mean() and std() that are "associated" with x2 are invoked in order to calculate the mean and standard deviation, respectively, of the elements of x2. The output from launching Listing 4.18 is here:

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

4.21 Calculating Mean and Standard Deviation: Another Example

The code sample in this section extends the code sample in the previous section with additional statistical values, and the code in Listing 4.19 can be used for any data distribution. Keep in mind that the code sample uses random numbers simply for the purposes of illustration: after you have launched the code sample, replace those numbers with values from a CSV file or some other dataset containing meaningful values.

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

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

Listing 4.19 displays the contents of stat_summary.py that illustrates how to display various statistical values from a NumPy array of random numbers.

Listing 4.19: stat_values.py

```
import numpy as np

from numpy import percentile
from numpy.random import rand

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

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

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

The data sample (shown in bold) in Listing 4.19 is from a uniform distribution between 0 and 1. The `NumPy` `percentile()` function calculates a linear interpolation (average) between observations, which is needed to calculate the median on a sample with an even number of values. As you can surmise, the `NumPy` functions `min()` and `max()` calculate the smallest and largest values in the data sample. The output from launching Listing 4.19 is here:

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

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

4.22 What is `Pandas`?

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

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

```
pip3 install pandas
```

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

Pandas DataFrames support a variety of input types, such as ndarrays, lists, dicts, or Series. Pandas also provides another data type called Pandas Series (not discussed in this chapter), this data structure provides another mechanism for managing data (search online for more details).

4.22.1 Pandas Dataframes

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

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

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

4.22.2 Dataframes and Data Cleaning Tasks

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

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

4.23 A Labeled Pandas Dataframe

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

Listing 4.20: pandas_labeled_df.py

```
import numpy
import pandas

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

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

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

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

Listing 4.20 contains two important statements followed by the variable `myarray`, which is a 3x3 `NumPy` array of numbers. The variables `rownames` and `colnames` provide names for the rows and columns, respectively, of the data in `myarray`. Next, the variable `mydf` is initialized as a `Pandas DataFrame` with the specified datasource (i.e., `myarray`).

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

```
           January      February         March
apples          10            30            20
oranges         50            40            60
beer          1000          2000          3000
           January      February         March
count     3.000000      3.000000      3.000000
mean    353.333333    690.000000   1026.666667
std     560.386771   1134.504297   1709.073823
min      10.000000     30.000000     20.000000
25%      30.000000     35.000000     40.000000
50%      50.000000     40.000000     60.000000
75%     525.000000   1020.000000   1530.000000
max    1000.000000   2000.000000   3000.000000
```

4.24 Pandas Numeric DataFrames

Listing 4.21 displays the contents of `pandas_numeric_df.py` that illustrates how to define a `Pandas DataFrame` whose rows and columns are numbers (but the column labels are characters).

Listing 4.21: pandas_numeric_df.py

```
import pandas as pd

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

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

```
       A        B        C       D
0 0.0457 _0.0141  1.3809    NaN
1_0.9554 _1.5010  0.0372    NaN
2_0.6627  1.5348 _0.8597    NaN
3_2.4529  1.2373 _0.1337    NaN
4 1.4145  1.9517 _2.3204    NaN
5_0.4949 _1.6497 _1.0846    NaN
6_1.0476 _0.7486 _0.8055    NaN
7    NaN      NaN      NaN    NaN
8    NaN      NaN      NaN    NaN
9    NaN      NaN      NaN    NaN
```

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

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

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

The output of the preceding code block is here:

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

4.25 Pandas Boolean DataFrames

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

Listing 4.22: pandas_boolean_df.py

```
import pandas as pd
```

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

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

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

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

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

```
df1 & df2:
      a      b
0  False  False
1  False  True
2  True   False
df1 | df2:
      a      b
0  True   True
1  True   True
2  True   True
df1 ^ df2:
      a      b
0  True   True
1  True   False
2  False  True
```

4.25.1 Transposing a `Pandas` Dataframe

The `T` attribute (as well as the transpose function) enables you to generate the transpose of a `Pandas DataFrame`, similar to a `NumpPy ndarray`.

For example, the following code snippet defines a `Pandas dataFrame` `df1` and then displays the transpose of `df1`:

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

```
print("df1.T:")
print(df1.T)
```

The output is here:

```
df1.T:
   0  1  2
a  1  0  1
b  0  1  1
```

The following code snippet defines Pandas dataFrames df1 and df2 and then displays their sum:

```
df1 = pd.DataFrame({'a' : [1, 0, 1], 'b' : [0, 1, 1] },
dtype=int)
df2 = pd.DataFrame({'a' : [3, 3, 3], 'b' : [5, 5, 5] },
dtype=int)

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

The output is here:

```
df1 + df2:
   a  b
0  4  5
1  3  6
2  4  6
```

4.26 Pandas Dataframes and Random Numbers

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

Listing 4.23: pandas_random_df.py

```
import pandas as pd
import numpy as np

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

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

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

```
         a    b
0      1.0  2.0
1      1.0  1.0
2      4.0  3.0
3      3.0  1.0
4      1.0  2.0
sum   10.0  9.0
mean   2.0  1.8
```

4.27 Combining Pandas DataFrames (1)

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

Listing 4.24: pandas_combine_df.py

```
import pandas as pd
import numpy as np

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

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

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

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

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

```
contents of df:
       foo1      foo2
0  0.274680 _0.848669
1 _0.399771 _0.814679
2  0.454443 _0.363392
3  0.473753  0.550849
```

```
4 _0.211783 _0.015014
contents of foo1:
0    0.256773
1    1.204322
2    1.040515
3   _0.518414
4    0.634141
Name: foo1, dtype: float64
contents of foo2:
0   _2.506550
1   _0.896516
2   _0.222923
3    0.934574
4    0.527033
Name: foo2, dtype: float64
```

4.28 Combining Pandas DataFrames (2)

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

Listing 4.25: concat_frames.py

```
import pandas as pd

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

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

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

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

```
0  Vancouver        40            72
1    Toronto        20            65
2   Montreal        25            50
0         SF        30            60
1    Chicago        15            40
2         LA        55            85
```

4.29 Data Manipulation with `Pandas Dataframes` (1)

As a simple example, suppose that we have a two-person company that keeps track of income and expenses on a quarterly basis, and we want to calculate the profit/loss for each quarter, and also the overall profit/loss.

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

Listing 4.26: pandas_quarterly_df1.py

```
import pandas as pd

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

df = pd.DataFrame(summary)

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

Listing 4.26 defines the variable `summary` that contains hard-coded quarterly information about cost and revenue for our two-person company. In general these hard-coded values would be replaced by data from another source (such as a CSV file), so think of this code sample as a simple way to illustrate some of the functionality that is available in `Pandas DataFrames`.

The variable df is a Pandas DataFrame based on the data in the summary variable. The three print statements display the quarters, the cost per quarter, and the revenue per quarter.

The output from Listing 4.26 is here:

```
Entire Dataset:
    Cost    Quarter  Revenue
0  23500      Q1      40000
1  34000      Q2      60000
2  57000      Q3      50000
3  32000      Q4      30000
Quarter:
0    Q1
1    Q2
2    Q3
3    Q4
Name: Quarter, dtype: object
Cost:
0    23500
1    34000
2    57000
3    32000
Name: Cost, dtype: int64
Revenue:
0    40000
1    60000
2    50000
3    30000
Name: Revenue, dtype: int64
```

4.30 Data Manipulation with Pandas DataFrames (2)

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

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

Listing 4.27: pandas_quarterly_df2.py

```
import pandas as pd
```

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

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

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

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

```
First Dataset:
   Cost     Quarter  Revenue
0 _23500      Q1      40000
1 _34000      Q2      60000
2 _57000      Q3      50000
3 _32000      Q4      30000
Second Dataset:
   Cost     Quarter  Revenue  Total
0 _23500      Q1      40000   16500
1 _34000      Q2      60000   26000
2 _57000      Q3      50000   _7000
3 _32000      Q4      30000   _2000
```

4.31 Data Manipulation with `Pandas Dataframes` (3)

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

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

Listing 4.28: pandas_quarterly_df3.py

```
import pandas as pd
```

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

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

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

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

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

```
First Dataset:
   Cost     Quarter  Revenue
0 _23500      Q1      40000
1 _34000      Q2      60000
2 _57000      Q3      50000
3 _32000      Q4      30000
Second Dataset:
      Cost    Quarter   Revenue  Total
0   _23500     Q1       40000    16500
1   _34000     Q2       60000    26000
2   _57000     Q3       50000    _7000
3   _32000     Q4       30000    _2000
Sum _146500  Q1Q2Q3Q4  180000    33500
```

4.32 Pandas DataFrames and CSV Files

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