Data Exploring and Analysis

Nowadays, massive data is collected daily and distributed over various channels. This requires efficient and flexible data analysis tools. Python's open source Pandas library fills that gap and deals with three different data structures: series, data frames, and panels. A *series* is a one-dimensional data structure such as a dictionary, array, list, tuple, and so on. A *data frame* is a two-dimensional data structure with heterogeneous data types, i.e., tabular data. A *panel* refers to a three-dimensional data structure such as a three-dimensional array. It should be clear that the higher-dimensional data structure is a container of its lower-dimensional data structure. In other words, a panel is a container of a data frame, and a data frame is a container of a series.

Series Data Structures

As mentioned earlier, a series is a sequence of one-dimensional data such as a dictionary, list, array, tuple, and so on.

Creating a Series

Pandas provides a `Series()` method that is used to create a series structure. A serious structure of size `n` should have an index of length `n`. By default Pandas creates indices starting at 0 and ending with `n-1`. A Pandas series can be created using the constructor `pandas.Series (data, index, dtype, copy)` where `data` could be an array, constant, list, etc. The series `index` should be unique and hashable with length `n`, while `dtype` is a data type that could be explicitly declared or inferred from the received data. Listing 6-1 creates a series with a default index and with a set index.

Listing 6-1. Creating a Series

```
In [5]: import pandas as pd
        import numpy as np
        data = np.array(['O','S','S','A'])
        S1 = pd.Series(data) # without adding index
        S2 = pd.Series(data,index=[100,101,102,103]) # with
        adding index print (S1) print ("\n") print (S2)
        0    O
        1    S
        2    S
        3    A
        dtype: object

        100  O
        101  S
        102  S
        103  A
        dtype: object
```

```
In [40]:import pandas as pd
        import numpy as np
        my_series2 = np.random.randn(5, 10)
        print ("\nmy_series2\n", my_series2)
```

This is the output of creating a series of random values of 5 rows and 10 columns.

```
my_series2
 [[ 0.08590877  0.59702919 -1.29330859 -1.42021041 -0.09535271  0.09058623
  -1.14191133 -0.84699991  0.94028641  1.79400706]
 [ 0.50645411 -0.37674882 -1.16751734 -1.24061761  0.03981985  0.13478382
   0.76132521 -0.40671662 -0.7484758   0.30420489]
 [-0.66951224 -1.19373055  1.86446782  1.43047631 -0.06302096  0.49239499
  -0.48208329 -1.9805521  -0.73735706 -1.03152802]
 [-0.79181088  1.02769491 -1.27216885  0.20320462  0.19385809 -0.51614599
  -0.66898612 -0.60962025 -1.43724096 -0.22663712]
 [ 1.14193093 -0.8842498   0.22409272 -0.29599594  1.1917404   1.09016684
   1.87701454  1.08452103 -1.49587483 -0.31887386]]
```

As mentioned earlier, you can create a series from a dictionary; Listing 6-2 demonstrates how to create an index for a data series.

Listing 6-2. Creating an Indexed Series

```
In [6]: import pandas as pd
        import numpy as np
        data = {'X' : 0., 'Y' : 1., 'Z' : 2.}
        SERIES1 = pd.Series(data)
        print (SERIES1)
        X 0.0
        Y 1.0
        Z 2.0
        dtype: float64
In [7]: import pandas as pd
        import numpy as np
        data = {'X' : 0., 'Y' : 1., 'Z' : 2.}
        SERIES1 = pd.Series(data,index=['Y','Z','W','X'])
        print (SERIES1)
        Y 1.0
```

```
Z 2.0
W NaN
X 0.0
dtype: float64
```

If you can create series data from a scalar value as shown in Listing 6-3, then an index is mandatory, and the scalar value will be repeated to match the length of the given index.

Listing 6-3. Creating a Series Using a Scalar

```
In [9]: # Use sclara to create a series
        import pandas as pd
        import numpy as np
        Series1 = pd.Series(7, index=[0, 1, 2, 3, 4])
        print (Series1)
        0    7
        1    7
        2    7
        3    7
        4    7
        dtype: int64
```

Accessing Data from a Series with a Position

Like lists, you can access a series data via its index value. The examples in Listing 6-4 demonstrate different methods of accessing a series of data. The first example demonstrates retrieving a specific element with index 0. The second example retrieves indices 0, 1, and 2. The third example retrieves the last three elements since the starting index is -3 and moves backward to -2, -1. The fourth and fifth examples retrieve data using the series index labels.

Listing 6-4. Accessing a Data Series

```
In [18]: import pandas as pd
         Series1 = pd.Series([1,2,3,4,5],index =
                             ['a','b','c','d','e'])
         print ("Example 1:Retrieve the first element")
         print (Series1[0] )
         print ("\nExample 2:Retrieve the first three element")
         print (Series1[:3])
         print ("\nExample 3:Retrieve the last three element")
         print(Series1[-3:])
         print ("\nExample 4:Retrieve a single element")
         print (Series1['a'])
         print ("\nExample 5:Retrieve multiple elements")
         print (Series1[['a','c','d']])
```

```
Example 1:Retrieve the first element
1

Example 2:Retrieve the first three element
a    1
b    2
c    3
dtype: int64

Example 3:Retrieve the last three element
c    3
d    4
e    5
dtype: int64

Example 4:Retrieve a single element
1

Example 5:Retrieve multiple elements
a    1
c    3
d    4
dtype: int64
```

Exploring and Analyzing a Series

Numerous statistical methods can be applied directly on a data series. Listing 6-5 demonstrates the calculation of mean, max, min, and standard deviation of a data series. Also, the `.describe()` method can be used to give a data description, including quantiles.

Listing 6-5. Analyzing Series Data

```
In [10]: import pandas as pd
         import numpy as np
         my_series1 = pd.Series([5, 6, 7, 8, 9, 10])
         print ("my_series1\n", my_series1)
         print ("\n Series Analysis\n ")
         print ("Series mean value : ", my_series1.mean()) #
         find mean value in a series
         print ("Series max value : ",my_series1.max()) #
         find max value in a series
         print ("Series min value : ",my_series1.min()) #
         find min value in a series
         print ("Series standard deviation value : ",
         my_series1.std()) # find standard deviation
         my_series1
         0    5
         1    6
         2    7
         3    8
         4    9
         5    10
         dtype: int64
```

```
         Series Analysis

         Series mean value : 7.5
         Series max value : 10
         Series min value : 5
         Series standard deviation value : 1.8708286933869707

In [11]: my_series1.describe()
Out[11]: count     6.000000
         mean      7.500000
         std       1.870829
         min       5.000000
         25%       6.250000
         50%       7.500000
         75%       8.750000
         max      10.000000
         dtype: float64
```

If you copied by reference one series to another, then any changes to the series will adapt to the other one. After copying `my_series1` to `my_series_11`, once you change the indices of `my_series_11`, it reflects back to `my_series1`, as shown in Listing 6-6.

Listing 6-6. Copying a Series to Another with a Reference

```
In [17]: my_series_11 = my_series1
         print (my_series1)
         my_series_11.index = ['A', 'B', 'C', 'D', 'E', 'F']
         print (my_series_11)
         print (my_series1)
         0    5
         1    6
         2    7
         3    8
```

```
4     9
5     10
dtype: int64
A     5
B     6
C     7
D     8
E     9
F     10
dtype: int64
A     5
B     6
C     7
D     8
E     9
F     10
dtype: int64
```

You can use the `.copy()` method to copy the data set without having a reference to the original series. See Listing 6-7.

Listing 6-7. Copying Series Values to Another

```
In [21]: my_series_11 = my_series1.copy()
         print (my_series1)
         my_series_11.index = ['A', 'B', 'C', 'D', 'E', 'F']
         print (my_series_11)
         print (my_series1)
         0     5
         1     6
         2     7
         3     8
```

```
4    9
5    10
dtype: int64
A    5
B    6
C    7
D    8
E    9
F    10
dtype: int64
0    5
1    6
2    7
3    8
4    9
5    10
dtype: int64
```

Operations on a Series

Numerous operations can be implemented on series data. You can check whether an index value is available in a series or not. Also, you can check all series elements against a specific condition, such as if the series value is less than 8 or not. In addition, you can perform math operations on series data directly or via a defined function, as shown in Listing 6-8.

Listing 6-8. Operations on Series

```
In [23]: 'F' in my_series_11
Out[23]: True

In [27]: temp = my_series_11 < 8
         temp
```

```
Out[27]: A    True
         B    True
         C    True
         D    False
         E    False
         F    False
         dtype: bool
         In [35]: len(my_series_11)

Out[35]: 6

In [28]: temp = my_series_11[my_series_11 < 8 ] * 2
         temp

Out[28]: A    10
         B    12
         C    14
         dtype: int64
```

Define a function to add two series and call the function, like this:

```
In [37]: def AddSeries(x,y):
             for i in range (len(x)):
                 print (x[i] + y[i])

In [39]: print ("Add two series\n")
         AddSeries (my_series_11, my_series1)
         Add two series
         10
         12
         14
         16
         18
         20
```

You can visualize data series using the different plotting systems that are covered in Chapter 7. However, Figure 6-1 demonstrates how to get an at-a-glance idea of your series data and graphically explore it via visual plotting diagrams. See Listing 6-9.

Listing 6-9. Visualizing Data Series

```
In [49]: import matplotlib.pyplot as plt
         plt.plot(my_series2)
         plt.ylabel('index')
         plt.show()
```

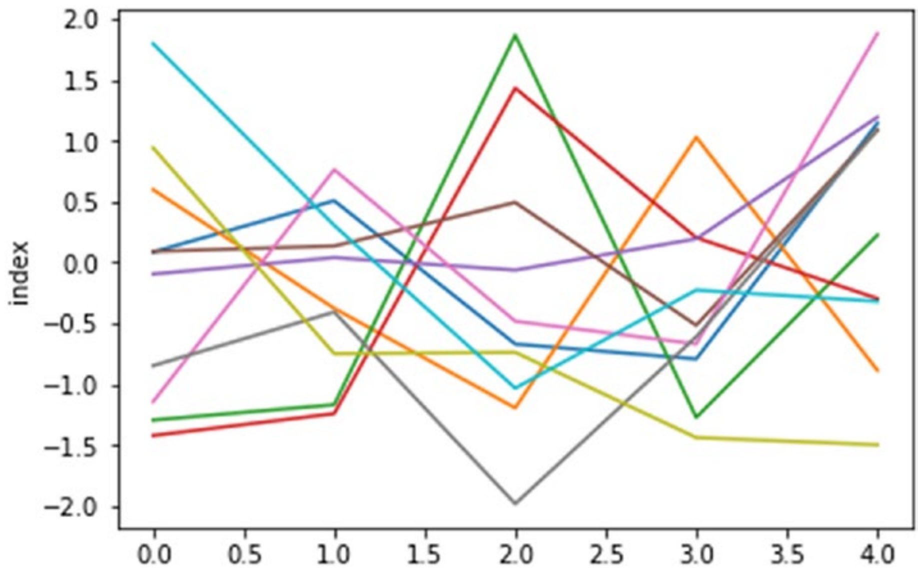

Figure 6-1. *Line visualization*

```
In [54]: from numpy import *
         import math
         import matplotlib.pyplot as plt
         t = linspace(0, 2*math.pi, 400)
```

```
        a = sin(t)
        b = cos(t)
        c = a + b

In [50]: plt.plot(t, a, 'r') # plotting t, a separately
         plt.plot(t, b, 'b') # plotting t, b separately
         plt.plot(t, c, 'g') # plotting t, c separately
         plt.show()
```

We can add multiple plots to the same canvas as shown in Figure 6-2.

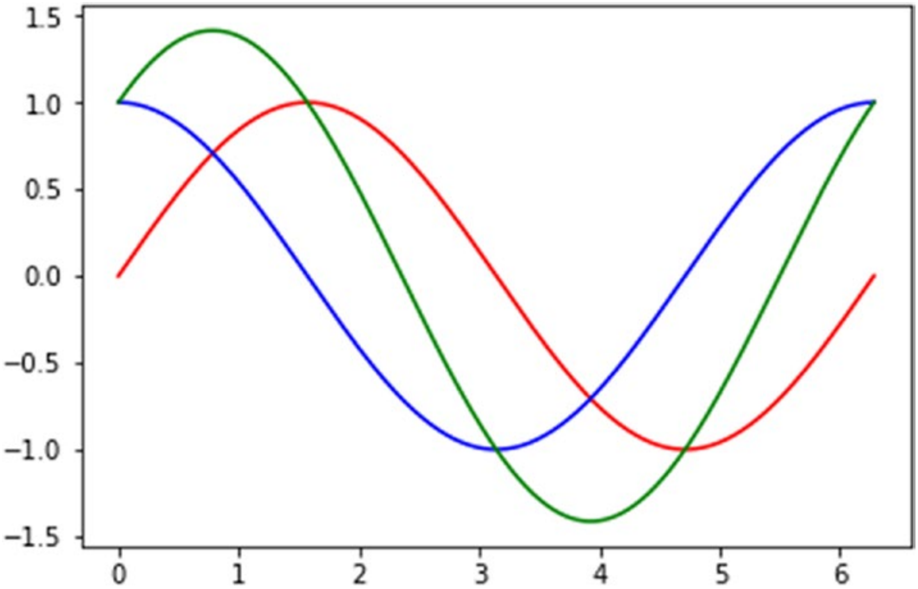

Figure 6-2. *Multiplots on the same canvas*

Data Frame Data Structures

As mentioned earlier, a data frame is a two-dimensional data structure with heterogeneous data types, i.e., tabular data.

Creating a Data Frame

Pandas can create a data frame using the constructor `pandas.DataFrame(data, index, columns, dtype, copy)`. A data frame can be created from lists, series, dictionaries, Numpy arrays, or other data frames. A Pandas data frame not only helps to store tabular data but also performs arithmetic operations on rows and columns of the data frame. Listing 6-10 creates a data frame from a single list and a list of lists.

Listing 6-10. Creating a Data Frame from a List

```
In [19]: import pandas as pd
         data = [10,20,30,40,50]
         DF1 = pd.DataFrame(data)
         print (DF1)
         0    10
         1    20
         2    30
         3    40
         4    50

In [22]: import pandas as pd
         data = [['Ossama',25],['Ali',43],['Ziad',32]]
         DF1 = pd.DataFrame(data,columns=['Name','Age'])
         print (DF1)
              Name       Age
         0    Ossama     25
         1    Ali        43
         2    Ziad       32

In [21]: import pandas as pd
         data = [['Ossama',25],['Ali',43],['Ziad',32]]
         DF1 = pd.DataFrame(data,columns=['Name','Age'],
         dtype=float) print (DF1)
```

```
        Name     Age
0       Ossama   25.0
1       Ali      43.0
2       Ziad     32.0
```

You can create a data frame from dictionaries or arrays, as shown in Listing 6-11. Also, you can set the data frame indices. However, if you don't set the indices, then the data frame starts with 0 and goes up to `n-1`, where `n` is the length of the list. Column names are taken by default from the dictionary keys. However, it's possible to set labels for columns as well. The first data frame's `df1` columns are labeled with the dictionary key names; that's why you don't see NaN cases except for the missing value of the project in dictionary 1. While in the second data frame, named `df2`, you change the column name from `Test1` to `Test_1`, and you get NaNs for all the records. This is because of the absence of `Test_1` in the dictionary key of `data`.

Listing 6-11. Creating a DataFrame from a Dictionary

```
In [13]: import pandas as pd
         data = [{'Test1': 10, 'Test2': 20},{'Test1': 30,
                 'Test2': 20, 'Project': 20}]
         # With three column indices, values same as dictionary
         keys
         df1 = pd.DataFrame(data, index=['First', 'Second'],
         columns=['Test2', 'Project' , 'Test1'])

         #With two column indices with one index with another
         name
         df2 = pd.DataFrame(data, index=['First', 'Second'],
         columns=['Project', 'Test_1','Test2 ')]
         print (df1)
         print ("\n")
         print (df2)
```

```
        Test2    Project    Test1
First   20       NaN        10
Second  20       20.0       30

        Project  Test_1     Test2
First   NaN      NaN        20
Second  20.0     NaN        20
```

Pandas allows you to create a data frame from a dictionary of series where you get the union of all series indices passed. As shown in Listing 6-12 with the student `Salwa`, no `Test1` value is given. That's why NaN is set automatically.

Listing 6-12. Creating a Data Frame from a Series

```
In [16]: import pandas as pd
         data = {'Test1' : pd.Series([70, 55, 89],
                 index=['Ahmed', 'Omar', 'Ali']),
                 'Test2' : pd.Series([56, 82, 77, 65],
                 index=['Ahmed', 'Omar', 'Ali', 'Salwa'])}

         df1 = pd.DataFrame(data)
         print (df1)

         Test1    Test2
Ahmed    70.0     56
Ali      89.0     77
Omar     55.0     82
Salwa    NaN      65
```

Updating and Accessing a Data Frame's Column Selection

You can select a specific column using the column labels. For example, `df1['Test2']` is used to select only the column labeled `Test2` in the data frame, while `df1[:]` is used to display all the columns and all the rows, as shown in Listing 6-13.

Listing 6-13. Data Frame Column Selection

```
In [51]: import pandas as pd
         data = {'Test1' : pd.Series([70, 55, 89],
                 index=['Ahmed', 'Omar', 'Ali']),
                 'Test2' : pd.Series([56, 82, 77, 65],
                 index=['Ahmed', 'Omar', 'Ali', 'Salwa'])}

         df1 = pd.DataFrame(data)
         print (df1['Test2']) # Column selection
         print("\n")
         print (df1[:]) # Column selection

         Ahmed     56
         Ali       77
         Omar      82
         Salwa     65
         Name: Test2, dtype: int64

                   Test1       Test2
         Ahmed     70.0        56
         Ali       89.0        77
         Omar      55.0        82
         Salwa     NaN         65
```

You can select columns by using the column labels or the column index. `df1.iloc[:, [1,0]]` is used to display all rows for columns 1 and 0 starting with column 1, which refers to the column named `Test2`. In addition, `df1[0:4:1]` is used to display all the rows starting from row 0 up to row 3 incremented by 1, which gives all rows from 0 up to 3. See Listing 6-14.

Listing 6-14. Data Frame Column and Row Selection

```
In [46]: df1.iloc[:, [1,0 ]]

Out[46]:            Test2   Test1
         Ahmed      56      70.0
         Ali        77      89.0
         Omar       82      55.0
         Salwa      65      NaN

In [39]: df1[0:4:1]

Out[39]:            Test1   Test2
         Ahmed      70.0    56
         Ali        89.0    77
         Omar       55.0    82
         Salwa      NaN     65
```

Column Addition

You can simply add a new column and add its values directly using a series. In addition, you can create a new column by processing the other columns, as shown in Listing 6-15.

Listing 6-15. Adding a New Column to a Data Frame

```
In [66]: # add a new Column
         import pandas as pd
         data = {'Test1' : pd.Series([70, 55, 89],
                 index=['Ahmed', 'Omar', 'Ali']),
                 'Test2' : pd.Series([56, 82, 77, 65],
                 index=['Ahmed', 'Omar', 'Ali', 'Salwa'])}
              df1 = pd.DataFrame(data)
              print (df1)
              df1['Project'] = pd.Series([90,83,67, 87],
              index=['Ali','Omar','Salwa', 'Ahmed'])
              print ("\n")
         df1['Average'] = round((df1['Test1']+df1['Test2']+
         df1['Project'])/3, 2)
         print (df1)

                Test1   Test2
         Ahmed  70.0    56
         Ali    89.0    77
         Omar   55.0    82
         Salwa  NaN     65

                Test1   Test2   Project   Average
         Ahmed  70.0    56      87        71.00
         Ali    89.0    77      90        85.33
         Omar   55.0    82      83        73.33
         Salwa  NaN     65      67        NaN
```

Column Deletion

You can delete any column using the `del` method. For example, `del df2['Test2']` deletes the `Test2` column from the data set. In addition, you can use the pop method to delete a column. For example, `df2.pop('Project')` is used to delete the column `Project`. However, you should be careful when you use the `del` or `pop` method since a reference might exist. In this case, it deletes not only from the executed data frame but also from the referenced data frame. Listing 6-16 creates the data frame `df1` and copies `df1` to `df2`.

Listing 6-16. Creating and Copying a Data Frame

```
In [70]: import pandas as pd
         data = {'Test1' : pd.Series([70, 55, 89],
                 index=['Ahmed', 'Omar', 'Ali']),
                 'Test2' : pd.Series([56, 82, 77, 65],
                 index=['Ahmed', 'Omar', 'Ali', 'Salwa'])}
         print (df1)
         df2 = df1
         print ("\n")
         print (df2)
                  Test1    Test2    Project    Average
         Ahmed    70.0     56       87         71.00
         Ali      89.0     77       90         85.33
         Omar     55.0     82       83         73.33
         Salwa    NaN      65       67         NaN
                  Test1    Test2    Project    Average
Ahmed             70.0     56       87         71.00
Ali               89.0     77       90         85.33
Omar              55.0     82       83         73.33
Salwa             NaN      65       6          7  NaN
```

In the previous Python script, you saw how to create `df2` and assign it `df1`. In Listing 6-17, you are deleting the `Test2` and `Project` variables using the `del` and `pop` methods sequentially. As shown, both variables are deleted from both data frames `df1` and `df2` because of the reference existing between these two data frames as a result of using the assign (=) operator.

Listing 6-17. Deleting Columns from a Data Frame

```
In [71]: # Delete a column in data frame using del function
         print ("Deleting the first column using DEL function:")
         del df2['Test2']
         print (df2)
         # Delete a column in data frame using pop function
         print ("\nDeleting another column using POP function:")
         df2.pop('Project')
         print (df2)

         Deleting the first column using DEL function:

                      Test1   Project    Average
         Ahmed        70.0    87         71.00
         Ali          89.0    90         85.33
         Omar         55.0    83         73.33
         Salwa        NaN     67         NaN

         Deleting another column using POP function:
                      Test1    Average
         Ahmed        70.0     71.00
         Ali          89.0     85.33
         Omar         55.0     73.33
         Salwa        NaN      NaN

In [72]: print (df1)
                      Test1    Average
         Ahmed        70.0     71.00
         Ali          89.0     85.33
         Omar         55.0     73.33
         Salwa        NaN      NaN
```

```
In [73]: print (df2)
                    Test1    Average
         Ahmed      70.0     71.00
         Ali        89.0     85.33
         Omar       55.0     73.33
         Salwa      NaN      NaN
```

To solve this problem, you can use the `df. copy()` method instead of the assign operator (=). Listing 6-18 shows that you deleted the variables `Test2` and `Project` using the `del()` and `pop()` methods sequentially, but only `df2` has been affected, while `df1` remains unchanged.

Listing 6-18. Using the Copy Method to Delete Columns from a Data Frame

```
In [83]: # add a new Column
         import pandas as pd
         data = {'Test1' : pd.Series([70, 55, 89],
                 index=['Ahmed', 'Omar', 'Ali']),
                 'Test2' : pd.Series([56, 82, 77, 65],
                 index=['Ahmed', 'Omar', 'Ali', 'Salwa'])}
         df1 = pd.DataFrame(data)
         df1['Project'] = pd.Series([90,83,67, 87],
         index=['Ali','Omar','Salwa', 'Ahmed'])
         print ("\n")
         df1['Average'] = round((df1['Test1']+df1['Test2']+df1
         ['Project'])/3, 2)
         print (df1)
         print ("\n")
         df2= df1.copy() # copy df1 into df2 using copy() method
         print (df2)
         #delete columns using del and pop methods
         del df2['Test2']
```

```
df2.pop('Project')
print ("\n")
print (df1)
print ("\n")
print (df2)
```

```
       Test1  Test2  Project  Average
Ahmed   70.0     56       87    71.00
Ali     89.0     77       90    85.33
Omar    55.0     82       83    73.33
Salwa    NaN     65       67      NaN


       Test1  Test2  Project  Average
Ahmed   70.0     56       87    71.00
Ali     89.0     77       90    85.33
Omar    55.0     82       83    73.33
Salwa    NaN     65       67      NaN


       Test1  Test2  Project  Average
Ahmed   70.0     56       87    71.00
Ali     89.0     77       90    85.33
Omar    55.0     82       83    73.33
Salwa    NaN     65       67      NaN


       Test1  Average
Ahmed   70.0    71.00
Ali     89.0    85.33
Omar    55.0    73.33
Salwa    NaN      NaN
```

Row Selection

In Listing 6-19, you are selecting the second row for student `Omar`. Also, you use the slicing methods to retrieve rows 2 and 3.

Listing 6-19. Retrieving Specific Rows

```
In [106]: # add a new Column
          import pandas as pd
          data = {'Test1' : pd.Series([70, 55, 89],
                  index=['Ahmed', 'Omar', 'Ali']),
                  'Test2' : pd.Series([56, 82, 77, 65],
                  index=['Ahmed', 'Omar', 'Ali', 'Salwa'])}
          df1 = pd.DataFrame(data)
          df1['Project'] = pd.Series([90,83,67, 87],index=
          ['Ali','Omar','Salwa', 'Ahmed'])
         print ("\n")
         df1['Average'] = round((df1['Test1']+df1['Test2']+df1
         ['Project'])/3, 2)
         print (df1)
         print ("\nselect iloc function to retrieve row number 2")
         print (df1.iloc[2])
         print ("\nslice rows")
         print (df1[2:4] )
```

```
       Test1  Test2  Project  Average
Ahmed   70.0     56       87    71.00
Ali     89.0     77       90    85.33
Omar    55.0     82       83    73.33
Salwa    NaN     65       67      NaN

select  iloc function to retrieve  row number 2
Test1      55.00
Test2      82.00
Project    83.00
Average    73.33
Name: Omar, dtype: float64

slice rows
       Test1  Test2  Project  Average
Omar    55.0     82       83    73.33
Salwa    NaN     65       67      NaN
```

Row Addition

Listing 6-20 demonstrates how to add rows to an existing data frame.

Listing 6-20. Adding New Rows to the Data Frame

```
In [134 ]: import pandas as pd
           data = {'Test1' : pd.Series([70, 55, 89],
                   index=['Ahmed', 'Omar', 'Ali']),
                   'Test2' : pd.Series([56, 82, 77, 65],
                   index=['Ahmed', 'Omar', 'Ali', 'Salwa']),
                'Project' : pd.Series([87, 83, 90, 67],
                index=['Ahmed', 'Omar', 'Ali', 'Salwa']),
                'Average' : pd.Series([71, 73.33, 85.33, 66],
                index=['Ahmed', 'Omar', 'Ali', 'Salw
           data = pd.DataFrame(data)
           print (data)
           print("\n")
           df2 = pd.DataFrame([[80, 70, 90, 80]], columns
           = ['Test1','Test2','Project','Average'],
           index=['Khalid'])
           datadata.append(df2)
           print (data)
```

```
       Average  Project  Test1  Test2
Ahmed    71.00       87   70.0     56
Ali      85.33       90   89.0     77
Omar     73.33       83   55.0     82
Salwa    66.00       67    NaN     65


        Average  Project  Test1  Test2
Ahmed     71.00       87   70.0     56
Ali       85.33       90   89.0     77
Omar      73.33       83   55.0     82
Salwa     66.00       67    NaN     65
Khalid    80.00       90   80.0     70
```

Row Deletion

Pandas provides the `df.drop()` method to delete rows using the label index, as shown in Listing 6-21.

Listing 6-21. Deleting Rows from a Data Frame

```
In [138]: print (data)
      print ('\n')
      data = data.drop('Omar')
      print (data)
```

```
        Average  Project  Test1  Test2
Ahmed     71.00       87   70.0     56
Ali       85.33       90   89.0     77
Omar      73.33       83   55.0     82
Salwa     66.00       67    NaN     65
Khalid    80.00       90   80.0     70

        Average  Project  Test1  Test2
Ahmed     71.00       87   70.0     56
Ali       85.33       90   89.0     77
Salwa     66.00       67    NaN     65
Khalid    80.00       90   80.0     70
```

Exploring and Analyzing a Data Frame

Pandas provides various methods for analyzing data in a data frame. The `.describe()` method is used to generate descriptive statistics that summarize the central tendency, dispersion, and shape of a data set's distribution, excluding NaN values.

```
DataFrame.describe(percentiles=None,include=None, exclude=None)
[source]
```

`DataFrame.describe()` analyzes both numeric and object series, as well as data frame column sets of mixed data types. The output will vary depending on what is provided. Listing 6-22 analyzes the `Age`, `Salary`, Height, and Weight attributes in a data frame. It also shows the mean, max, min, standard deviation, and quantiles of all attributes. However, Salwa's Age is missing; you get the full description of Age attributes excluding Salwa's data.

Listing 6-22. Creating a Data Frame with Five Attributes

```
In [61]: print (df1)
data = {'Age' : pd.Series([30, 25, 44, ],
index=['Ahmed', 'Omar', 'Ali']),
'Salary' : pd.Series([25000, 17000, 30000, 12000],
index=['Ahmed', 'Omar', 'Ali',
'Height' : pd.Series([160, 154, 175, 165],
index=['Ahmed', 'Omar', 'Ali', 'Salwa'
'Weight' : pd.Series([85, 70, 92, 65], index=['Ahmed', 'Omar',
'Ali', 'Salwa']),
'Gender' : pd.Series(['Male', 'Male', 'Male', 'Female'],
index=['Ahmed', 'Omar',

data = pd.DataFrame(data)
print (data)
print("\n")
df2 = pd.DataFrame([[42, 31000, 170, 80, 'Female']], columns
=['Age','Salary','Height'
                       , index=['Mona'])

data = data.append(df2)
print (data)
```

```
        Age  Gender  Height  Salary  Weight
Ahmed  30.0    Male     160   25000      85
Ali    44.0    Male     175   30000      92
Omar   25.0    Male     154   17000      70
Salwa   NaN  Female     165   12000      65
```

```
        Age  Gender  Height  Salary  Weight
Ahmed  30.0    Male     160   25000      85
Ali    44.0    Male     175   30000      92
Omar   25.0    Male     154   17000      70
Salwa   NaN  Female     165   12000      65
Mona   42.0  Female     170   31000      80
```

Applying the `data.describe()` method, you get the full description of all attributes except the `Gender` attribute because of its string data type. You can enforce implementation of all attributes by using the `include='all'` method attribute. Also, you can apply the analysis to a specific pattern, for example, to the `Salary` pattern only, which finds the mean, min, max, std, and quantiles of all employees' salaries. See Listing 6-23.

Listing 6-23. Analyzing a Data Frame

```
In [63]: data.describe()
```

Out[63]:

	Age	Height	Salary	Weight
count	4.000000	5.000000	5.000000	5.000000
mean	35.250000	144.800000	23000.000000	78.400000
std	9.215024	42.517055	8276.472679	10.968136
min	25.000000	70.000000	12000.000000	65.000000
25%	28.750000	154.000000	17000.000000	70.000000
50%	36.000000	160.000000	25000.000000	80.000000
75%	42.500000	165.000000	30000.000000	85.000000
max	44.000000	175.000000	31000.000000	92.000000

```
In [64]: data.describe(include='all')
```

Out[64]:

	Age	Gender	Height	Salary	Weight
count	4.000000	5	5.000000	5.000000	5.000000
unique	NaN	2	NaN	NaN	NaN
top	NaN	Male	NaN	NaN	NaN
freq	NaN	3	NaN	NaN	NaN
mean	35.250000	NaN	144.800000	23000.000000	78.400000
std	9.215024	NaN	42.517055	8276.472679	10.968136
min	25.000000	NaN	70.000000	12000.000000	65.000000
25%	28.750000	NaN	154.000000	17000.000000	70.000000
50%	36.000000	NaN	160.000000	25000.000000	80.000000
75%	42.500000	NaN	165.000000	30000.000000	85.000000
max	44.000000	NaN	175.000000	31000.000000	92.000000

```
In [66]: data.Salary.describe()
```

```
Out[66]: count        5.000000
         mean     23000.000000
         std       8276.472679
         min      12000.000000
         25%      17000.000000
         50%      25000.000000
         75%      30000.000000
         max      31000.000000
         Name: Salary, dtype: float64
```

Listing 6-24 includes only the numeric columns in a data frame's description.

Listing 6-24. Analyzing Only Numerical Patterns

```
In [67]: data.describe(include=[np.number])
```

Out[67]:

	Age	Height	Salary	Weight
count	4.000000	5.000000	5.000000	5.000000
mean	35.250000	144.800000	23000.000000	78.400000
std	9.215024	42.517055	8276.472679	10.968136
min	25.000000	70.000000	12000.000000	65.000000
25%	28.750000	154.000000	17000.000000	70.000000
50%	36.000000	160.000000	25000.000000	80.000000
75%	42.500000	165.000000	30000.000000	85.000000
max	44.000000	175.000000	31000.000000	92.000000

Listing 6-25 includes only string columns in a data frame's description.

Listing 6-25. Analyzing String Patterns Only (Gender)

```
In [68]: data.describe(include=[np.object])
```

Out[68]:

	Gender
count	5
unique	2
top	Male
freq	3

```
In [70]: data.describe(exclude=[np.number])
```

Out[70]:

	Gender
count	5
unique	2
top	Male
freq	3

You can measure overweight employee by calculating the optimal weight and comparing this with their recorded weight, as shown in Listing 6-26.

Listing 6-26. Checking the Weight Optimality

```
In [71]: data
```

Out[71]:

	Age	Gender	Height	Salary	Weight
Ahmed	30.0	Male	160	25000	85
Ali	44.0	Male	175	30000	92
Omar	25.0	Male	154	17000	70
Salwa	NaN	Female	165	12000	65
Mona	42.0	Female	70	31000	80

```
In [75]: OptimalWeight = data['Height']- 100
         OptimalWeight
```

```
Out[75]: Ahmed     60
         Ali       75
         Omar      54
         Salwa     65
         Mona      70
         Name: Height, dtype: int64
```

```
In [93]:unOptimalCases = data['Weight'] <= OptimalWeight
unOptimalCases
```

```
Out[93]: Ahmed     False
         Ali       False
         Omar      False
         Salwa      True
         Mona      False
         dtype: bool
```

Panel Data Structures

As mentioned earlier, a *panel* is a three-dimensional data structure like a three-dimensional array.

Creating a Panel

Pandas creates a panel using the constructor `pandas.Panel(data, items, major_axis, minor_axis, dtype, copy)`. The panel can be created from a dictionary of data frames and narrays. The data can take various forms, such as ndarray, series, map, lists, dictionaries, constants, and also another data frames.

The following Python script creates an empty panel:

```
#creating an empty panel
import pandas as pd
p = pd.Panel ()
```

Listing 6-27 creates a panel with three dimensions.

Listing 6-27. Creating a Panel with Three Dimensions

```
In [143]: # creating an empty panel
          import pandas as pd
          import numpy as np

          data = np.random.rand(2,4,5)
          Paneldf = pd.Panel(data)
          print (Paneldf)
```

```
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 4 (major_axis) x 5 (minor_axis)
Items axis: 0 to 1
Major_axis axis: 0 to 3
Minor_axis axis: 0 to 4
```

Accessing Data from a Panel with a Position

Listing 6-28 creates a panel and fills it with random data, where the first item in the panel is a 4x3 array and the second item is a 4x2 array of random values. For the `Item2` column, two values are NaN since its dimension is 4x2. You can also access data from a panel using item labels, as shown in Listing 6-28.

Listing 6-28. Selecting and Displaying Panel Items

```
In [147]: # creating an empty panel

import pandas as pd
import numpy as np
data = {'Item1' : pd.DataFrame(np.random.randn(4, 3)),
        'Item2' : pd.DataFrame(np.random.randn(4, 2))}
Paneldf = pd.Panel(data)
print (Paneldf['Item1'])
print ("\n")
print (Paneldf['Item2'])
```

```
          0         1         2
0 -1.069595  0.835842  0.950269
1  1.063784  0.520086  1.342309
2 -2.236069  0.229717  0.752612
3  1.014550  0.903234  2.011993


          0         1   2
0 -1.126333  1.528085 NaN
1 -1.255712  0.076873 NaN
2  1.593704 -0.648342 NaN
3  0.287446  1.591275 NaN
```

Python displays the panel items in a data frame with two dimensions, as shown previously. Data can be accessed using the method `panel.major_axis(index)` and also using the method `panel.minor_axis(index)`. See Listing 6-29.

Listing 6-29. Selecting and Displaying a Panel with Major and Minor Dimensions

```
In [149]: print (Paneldf.major_xs(1))
```

```
      Item1     Item2
0  1.063784 -1.255712
1  0.520086  0.076873
2  1.342309       NaN
```

```
In [150]: print (Paneldf.minor_xs(1))
```

```
      Item1     Item2
0  0.835842  1.528085
1  0.520086  0.076873
2  0.229717 -0.648342
3  0.903234  1.591275
```

Exploring and Analyzing a Panel

Once you have a panel, you can make statistical analysis on the maintained data. In Listing 6-30, you can see two groups of employees, each of which has five attributes maintained in a panel called `P`. You implement the `.describe()` method for `Group1`, as well as for the `Salary` attribute in this group.

Listing 6-30. Panel Analysis

```
In [104]: import pandas as pd
data1 = {'Age' : pd.Series([30, 25, 44, ], index=['Ahmed',
'Omar', 'Ali']),
'Salary' : pd.Series([25000, 17000, 30000, 12000],
index=['Ahmed', 'Omar', 'Ali', 'Salwa']),
'Height' : pd.Series([160, 154, 175, 165], index=['Ahmed',
'Omar', 'Ali', 'Salwa']),
```

```
'Weight' : pd.Series([85, 70, 92, 65], index=['Ahmed', 'Omar',
'Ali', 'Salwa']),
'Gender' : pd.Series(['Male', 'Male', 'Male', 'Female'],
index=['Ahmed', 'Omar', 'Ali', 'Salwa'])}

data2 = {'Age' : pd.Series([24, 19, 33,25  ], index=['Ziad',
'Majid', 'Ayman', 'Ahlam']),
'Salary' : pd.Series([17000, 7000, 22000, 21000],
index=['Ziad', 'Majid', 'Ayman', 'Ahlam']),
'Height' : pd.Series([170, 175, 162, 177], index=['Ziad',
'Majid', 'Ayman', 'Ahlam']),
'Weight' : pd.Series([77, 84, 74, 90], index=['Ziad', 'Majid',
'Ayman', 'Ahlam']),
'Gender' : pd.Series(['Male', 'Male', 'Male', 'Female'],
index=['Ziad', 'Majid', 'Ayman', 'Ahlam'])}

data = {'Group1': data1, 'Group2': data2}
p = pd.Panel(data)

In [106]: p['Group1'].describe()
```

Out[106]:

	Age	Gender	Height	Salary	Weight
count	3.0	4	4.0	4.0	4.0
unique	3.0	2	4.0	4.0	4.0
top	30.0	Male	175.0	30000.0	70.0
freq	1.0	3	1.0	1.0	1.0

```
In [107]: p['Group1']['Salary'].describe()
```

```
Out[107]: count          4.0
          unique         4.0
          top        30000.0
          freq           1.0
          Name: Salary, dtype: float64
```

Data Analysis

As indicated earlier, Pandas provides numerous methods for data analysis. The objective in this section is to get familiar with the data and summarize its main characteristics. Also, you can define your own methods for specific statistical analyses.

Statistical Analysis

Most of the following statistical methods were covered earlier with practical examples of the three main data collections: series, data frames, and panels.

- `df.describe()`: Summary statistics for numerical columns
- `df.mean()`: Returns the mean of all columns
- `df.corr()`: Returns the correlation between columns in a data frame
- `df.count()`: Returns the number of non-null values in each data frame column
- `df.max()`: Returns the highest value in each column
- `df.min()`: Returns the lowest value in each column
- `df.median()`: Returns the median of each column
- `df.std()`: Returns the standard deviation of each column

Listing 6-31 creates a data frame with six columns and ten rows.

Listing 6-31. Creating a Data Frame

```
In [11]: import pandas as pd
import numpy as np
```

```
Number = [1,2,3,4,5,6,7,8,9,10]
Names = ['Ali Ahmed','Mohamed Ziad','Majid Salim','Salwa
Ahmed', 'Ahlam Mohamed', 'Omar Ali', 'Amna Mohammed','Khalid
Yousif', 'Safa Humaid', 'Amjad Tayel']
City = ['Fujairah','Dubai','Sharjah','AbuDhabi','Fujairah','Dub
ai', 'Sharja ', 'AbuDhabi','Sharjah','Fujairah']
columns = ['Number', 'Name', 'City' ]
dataset= pd.DataFrame({'Number': Number , 'Name': Names,
'City': City}, columns = columns )
Gender= pd.DataFrame({'Gender':['Male','Male','Male','Female',
'Female', 'Male', 'Female', 'Male','Female', 'Male']})
Height = pd.DataFrame(np.random.randint(120,175, size=(12, 1)))
Weight = pd.DataFrame(np.random.randint(50,110, size=(12, 1)))
dataset['Gender']= Gender
dataset['Height']= Height
dataset['Weight']= Weight
dataset.set_index('Number')
```

Out[166]:

	Name	City	Gender	Height	Weight
Number					
1	Ali Ahmed	Fujairah	Male	131	71
2	Mohamed Ziad	Dubai	Male	153	74
3	Majid Salim	Sharjah	Male	145	104
4	Salwa Ahmed	AbuDhabi	Female	173	86
5	Ahlam Mohamed	Fujairah	Female	158	82
6	Omar Ali	Dubai	Male	134	89
7	Amna Mohammed	Sharjah	Female	136	93
8	Khalid Yousif	AbuDhabi	Male	128	98
9	Safa Humaid	Sharjah	Female	162	81
10	Amjad Tayel	Fujairah	Male	160	77

The Python script and examples in Listing 6-32 show the summary of height and weight variables, the mean values of height and weight, the correlation between the numerical variables, and the count of all records in the data set. The correlation coefficient is a measure that determines the degree to which two variables' movements are associated. The most common correlation coefficient, generated by the Pearson correlation, may be used to measure the linear relationship between two variables. However, in a nonlinear relationship, this correlation coefficient may not always be a suitable measure of dependence. The range of values for the correlation coefficient is -1.0 to 1.0. In other words, the values cannot exceed 1.0 or be less than -1.0, whereby a correlation of -1.0 indicates a perfect negative correlation, and a correlation of 1.0 indicates a perfect positive correlation. The correlation coefficient is denoted as `r`. If its value greater than zero, it's a positive relationship; while if the value is less than zero, it's a negative relationship. A value of zero indicates that there is no relationship between the two variables.

As shown, there is a weak negative correlation (-0.301503) between the height and width of all members in the data set. Also, the initial stats show that the height has the highest deviation; in addition, the 75th quantile of the height is equal to 159.

Listing 6-32. Summary and Statistics of Variables

```
In [186]: # Summary statistics for numerical columns
print ( dataset.describe())
```

	Number	Height	Weight
count	10.00000	10.00000	10.000000
mean	5.50000	148.00000	85.500000
std	3.02765	15.37675	10.617072
min	1.00000	128.00000	71.000000
25%	3.25000	134.50000	78.000000
50%	5.50000	149.00000	84.000000
75%	7.75000	159.50000	92.000000
max	10.00000	173.00000	104.000000

```
In [187]: print (dataset.mean()) # Returns the mean of all
columns

Number      5.5
Height    148.0
Weight     85.5
dtype: float64


In [188]: # Returns the correlation between columns in a
DataFrame
print (dataset.corr())

          Number    Height    Weight
Number  1.000000  0.124105  0.174557
Height  0.124105  1.000000 -0.301503
Weight  0.174557 -0.301503  1.000000


In [189]: # Returns the number of non-null values in each
DataFrame       column
print (dataset.count())

Number    10
Name      10
City      10
Gender    10
Height    10
Weight    10
dtype: int64
```

```
In [190]: # Returns the highest value in each column
print (dataset.max())
```

```
Number              10
Name       Salwa Ahmed
City           Sharjah
Gender            Male
Height             173
Weight             104
dtype: object
```

```
In [191]: # Returns the lowest value in each column
print (dataset.min())
```

```
Number                1
Name      Ahlam Mohamed
City           AbuDhabi
Gender           Female
Height              128
Weight               71
dtype: object
```

```
In [192]: # Returns the median of each column
print (dataset.median())
```

```
Number      5.5
Height    149.0
Weight     84.0
dtype: float64
```

```
In [193]: # Returns the standard deviation of each column
print (dataset.std())
```

```
Number     3.027650
Height    15.376750
Weight    10.617072
dtype: float64
```

Data Grouping

You can split data into groups to perform more specific analysis over the data set. Once you perform data grouping, you can compute summary statistics (aggregation), perform specific group operations (transformation), and discard data with some conditions (filtration). In Listing 6-33, you group data using `City` and find the count of genders per city. In addition, you group the data set by city and display the results, where for example rows 1 and 5 are people from Dubai. You can use multiple grouping attributes. You can group the data set using `City` and `Gender.` The retrieved data shows that, for instance, Fujairah has females (row 4) and males (rows 0 and 9).

Listing 6-33. Data Grouping

```
In [3]: dataset.groupby('City')['Gender'].count()
```

The following output shows that we have 2 students from Abu dhabi, 2 from Dubai, 3 from Fujairah and 3 from Sharjah groupped by gender.

```
Out[3]: City
        AbuDhabi     2
        Dubai        2
        Fujairah     3
        Sharjah      3
        Name: Gender, dtype: int64
```

```
In [4]: print (dataset.groupby('City').groups)
```

```
{'AbuDhabi': Int64Index([3, 7], dtype='int64'), 'Dubai': Int64Index([1, 5], dtype='int64'), 'Fujairah': Int64I
ndex([0, 4, 9], dtype='int64'), 'Sharjah': Int64Index([2, 6, 8], dtype='int64')}
```

```
In [5]: print (dataset.groupby(['City','Gender']).groups)
```

```
{('AbuDhabi', 'Female'): Int64Index([3], dtype='int64'), ('AbuDhabi', 'Male'): Int64Index([7], dtype='int64'),
('Dubai', 'Male'): Int64Index([1, 5], dtype='int64'), ('Fujairah', 'Female'): Int64Index([4], dtype='int64'),
('Fujairah', 'Male'): Int64Index([0, 9], dtype='int64'), ('Sharjah', 'Female'): Int64Index([6, 8], dtype='int6
4'), ('Sharjah', 'Male'): Int64Index([2], dtype='int64')}
```

Iterating Through Groups

You can iterate through a specific group, as shown in Listing 6-34. When you iterate through the gender, it should be clear that by default the groupby object has the same name as the group name.

Listing 6-34. Iterating Through Grouped Data

```
In [7]: grouped = dataset.groupby('Gender')
        for name,group in grouped:
            print (name)
            print (group)
            print ("\n")
```

```
Female
   Number          Name      City  Gender  Height  Weight
3       4   Salwa Ahmed  AbuDhabi  Female     125      57
4       5 Ahlam Mohamed  Fujairah  Female     170      99
6       7 Amna Mohammed   Sharjah  Female     160      97
8       9   Safa Humaid   Sharjah  Female     138      70


Male
   Number          Name      City Gender  Height  Weight
0       1    Ali Ahmed  Fujairah   Male     130      72
1       2  Mohamed Ziad    Dubai   Male     129      61
2       3   Majid Salim  Sharjah   Male     153      51
5       6      Omar Ali    Dubai   Male     135      97
7       8 Khalid Yousif AbuDhabi   Male     170      55
9      10   Amjad Tayel  Fujairah   Male     163      88
```

You can also select a specific group using the get_group() method, as shown in Listing 6-35 where you group data by gender and then select only females.

Listing 6-35. Selecting a Single Group

```
In [9]: grouped = dataset.groupby('Gender')
        print (grouped.get_group('Female'))
```

	Number	Name	City	Gender	Height	Weight
3	4	Salwa Ahmed	AbuDhabi	Female	125	57
4	5	Ahlam Mohamed	Fujairah	Female	170	99
6	7	Amna Mohammed	Sharjah	Female	160	97
8	9	Safa Humaid	Sharjah	Female	138	70

Aggregations

Aggregation functions return a single aggregated value for each group. Once the groupby object is created, you can implement various functions on the grouped data. In Listing 6-36, you calculate the mean and size of height and weight for both males and females. In addition, you calculate the summation and standard deviations for both patterns of males and females.

Listing 6-36. Data Aggregation

```
In [18]: # Aggregation
         grouped = dataset.groupby('Gender')
         print (grouped['Height'].agg(np.mean))
         print ("\n")
         print (grouped['Weight'].agg(np.mean))
         print ("\n")
         print (grouped.agg(np.size))
         print ("\n")
         print (grouped['Height'].agg([np.sum, np.mean,
         np.std]))
```

```
Gender
Female    145.250000
Male      159.333333
Name: Height, dtype: float64


Gender
Female    88.750000
Male      83.666667
Name: Weight, dtype: float64


        Number  Name  City  Height  Weight
Gender
Female       4     4     4       4       4
Male         6     6     6       6       6


        sum        mean       std
Gender
Female  581  145.250000  7.274384
Male    956  159.333333  8.891944
```

Transformations

Transformation on a group or a column returns an object that is indexed the same size as the one being grouped. Thus, the transform should return a result that is the same size as that of a group chunk. See Listing 6-37.

Listing 6-37. Creating the Index

```
In [26]: dataset = dataset.set_index(['Number'])
         print (dataset)
```

```
                Name      City  Gender  Height  Weight
Number
1         Ali Ahmed  Fujairah    Male     155      65
2       Mohamed Ziad     Dubai    Male     165      59
3        Majid Salim   Sharjah    Male     159      82
4        Salwa Ahmed  AbuDhabi  Female     138     106
5      Ahlam Mohamed  Fujairah  Female     152     100
6           Omar Ali     Dubai    Male     145     108
7      Amna Mohammed   Sharjah  Female     151      67
8       Khalid Yousif  AbuDhabi    Male     171      96
9        Safa Humaid   Sharjah  Female     140      82
10       Amjad Tayel  Fujairah    Male     161      92
```

In Listing 6-38, you group data by `Gender`, then implement the function lambda `x: (x - x.mean()) / x.std()*10`, and display results for both height and weight. The lambda operator or lambda function is a way to create a small anonymous function, i.e., a function without a name. This function is throwaway function; in other words, it is just needed where it has been created.

Listing 6-38. Transformation

```
In [28]: grouped = dataset.groupby('Gender')
         score = lambda x: (x - x.mean()) / x.std()*10
         print (grouped.transform(score))
```

```
             Height     Weight
Number
1         -4.873325  -9.911893
2          6.372810 -13.097858
3         -0.374871  -0.884990
4         -9.966479   9.730865
5          9.279136   6.346216
6        -16.119460  12.920860
7          7.904449 -12.269352
8         13.120491   6.548929
9         -7.217106  -3.807730
10         1.874356   4.424952
```

Filtration

Python provides direct filtering for data. In Listing 6-39, you applied filtering by city, and the return cities appear more than three times in the data set.

Listing 6-39. Filtration

```
In [30]: print (dataset.groupby('City').filter(lambda x: len(x)
>= 3))
```

```
                 Name      City  Gender  Height  Weight
Number
1           Ali Ahmed  Fujairah    Male     155      65
3         Majid Salim   Sharjah    Male     159      82
5       Ahlam Mohamed  Fujairah  Female     152     100
7       Amna Mohammed   Sharjah  Female     151      67
9         Safa Humaid   Sharjah  Female     140      82
10        Amjad Tayel  Fujairah    Male     161      92
```

Summary

This chapter covered how to explore and analyze data in different collection structures. Here is a list of what you just studied in this chapter:

- How to implement Python techniques to explore and analyze a series of data, create a series, access data from series with the position, and apply statistical methods on a series.
- How to explore and analyze data in a data frame, create a data frame, and update and access data. This included column and row selection, addition, and deletion, as well as applying statistical methods on a data frame.
- How to apply statistical methods on a panel to explore and analyze its data.
- How to apply statistical analysis on the derived data from implementing Python data grouping, iterating through groups, aggregations, transformations, and filtration techniques.

The next chapter will cover how to visualize data using numerous plotting packages and much more.

Exercises and Answers

A. Create a data frame called `df` from the following tabular data dictionary that has these index labels: `['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']`.

	Animal	Age	Priority	Visits
a	cat	2.5	yes	1
b	cat	3.0	yes	3
c	snake	0.5	no	2
d	dog	NaN	yes	3
e	dog	5.0	no	2
f	cat	2.0	no	3
g	snake	4.5	no	1
h	cat	NaN	yes	1
i	dog	7.0	no	2
j	dog	3.0	no	1

Answer:

You should import both the Pandas and Numpy libraries.

```
import numpy as np
import pandas as pd
```

You must create a dictionary and list of labels and then call the data frame method and assign the labels list as an index, as shown in Listing 6-40.

Listing 6-40. Creating a Tabular Data Frame

```
In [5]: import numpy as np
        import pandas as pd
        import matplotlib as mpl
```

```
data = { 'Animal': ['cat', 'cat', 'snake', 'dog', 'dog',
                 'cat', 'snake', 'cat', 'dog', 'dog'],
'Age': [2.5, 3, 0.5, np.nan, 5, 2, 4.5, np.nan, 7, 3],
'Visits': [1, 3, 2, 3, 2, 3, 1, 1, 2, 1],
'Priority': ['yes', 'yes', 'no', 'yes', 'no', 'no', 'no',
'yes', 'no', 'no']}

labels = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']

#Create a DataFrame df from this dictionary data which has the
index labels.
df = pd.DataFrame( data, index = labels, columns=['Animal',
'Age', 'Priority', 'Visits'])
print (df)
```

```
  Animal  Age Priority  Visits
a    cat  2.5      yes       1
b    cat  3.0      yes       3
c  snake  0.5       no       2
d    dog  NaN      yes       3
e    dog  5.0       no       2
f    cat  2.0       no       3
g  snake  4.5       no       1
h    cat  NaN      yes       1
i    dog  7.0       no       2
j    dog  3.0       no       1
```

B. Display a summary of the data frame's basic information.

You can use `df.info()` and `df.describe()` to get a full description of your data set, as shown in Listing 6-41.

Listing 6-41. Data Frame Summary

```
In [6]: df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 10 entries, a to j
Data columns (total 4 columns):
Animal      10 non-null object
Age         8 non-null float64
Priority    10 non-null object
Visits      10 non-null int64
dtypes: float64(1), int64(1), object(2)
memory usage: 400.0+ bytes
```

```
In [7]: df.describe()
```

	Age	Visits
count	8.000000	10.000000
mean	3.437500	1.900000
std	2.007797	0.875595
min	0.500000	1.000000
25%	2.375000	1.000000
50%	3.000000	2.000000
75%	4.625000	2.750000
max	7.000000	3.000000

C. Return the first three rows of the data frame `df`.

Listing 6-42 shows the use of `df.iloc[:3]` and `df.head(3)` to retrieve the first `n` rows of the data frame.

Listing 6-42. Selecting a Specific n Rows

```
In [12]: df.head(3)
```

Out[12]:

	Animal	Age	Priority	Visits
a	cat	2.5	yes	1
b	cat	3.0	yes	3
c	snake	0.5	no	2

```
In [13]: df.iloc[:3]
```

Out[13]:

	Animal	Age	Priority	Visits
a	cat	2.5	yes	1
b	cat	3.0	yes	3
c	snake	0.5	no	2

D. Select just the `animal` and age columns from the data frame `df`.

The Python data frame `loc()` method is used to retrieve the specific pattern `df.loc[ : , ['Animal', 'Age']]`. In addition, an array form retrieval can be used too with `df[['Animal', 'Age']]` . See Listing 6-43.

Listing 6-43. Slicing Data Frame

```
In [16]: df.loc[:,['Animal', 'Age']]
         # or
         df [['Animal', 'Age']]
```

Out[16]:

	Animal	Age
a	cat	2.5
b	cat	3.0
c	snake	0.5
d	dog	NaN
e	dog	5.0
f	cat	2.0
g	snake	4.5
h	cat	NaN
i	dog	7.0
j	dog	3.0

E. Count the visit priority per animal.

```
In [8]: df.groupby('Priority')['Animal'].count()
```

F. Find the mean of the animals' ages.

```
In [10]: df.groupby('Animal')['Age'].mean()
```

G. Display a summary of the data set. See Listing 6-44.

Listing 6-44. Data Set Summary

```
In [13]: df.groupby('Animal')['Age'].describe()
```

Out[13]:

	count	mean	std	min	25%	50%	75%	max
Animal								
cat	3.0	2.5	0.500000	2.0	2.25	2.5	2.75	3.0
dog	3.0	5.0	2.000000	3.0	4.00	5.0	6.00	7.0
snake	2.0	2.5	2.828427	0.5	1.50	2.5	3.50	4.5