TRANSFORMATION TECHNIQUES

Other types of data transformations including cleaning, filtering, deduplication, and others.

1. Performing data deduplication

It is very likely that your dataframe contains duplicate rows. Removing them is essential to enhance the quality of the dataset. This can be done with the following steps:

1. Let's consider a simple dataframe, as follows:

```
frame3 = pd.DataFrame({'column 1': ['Looping'] * 3 + ['Functions'] * 4,
'column 2': [10, 10, 22, 23, 23, 24, 24]})
frame3
```

The preceding code creates a simple dataframe with two columns. You can clearly see from the following screenshot that in both columns, there are some duplicate entries:

	column 1	column 2
0	Looping	10
1	Looping	10
2	Looping	22
3	Functions	23
4	Functions	23
5	Functions	24
6	Functions	24

2. The pandas dataframe comes with a duplicated() method that returns a Boolean series stating which of the rows are duplicates:

```
frame3.duplicated()
```

The output of the preceding code is pretty easy to interpret:

```
0    False
1     True
2    False
3    False
4     True
5    False
6     True
dtype: bool
```

The rows that say True are the ones that contain duplicated data.

3. Now, we can drop these duplicates using the drop_duplicates() method:

```
frame4 = frame3.drop_duplicates()
frame4
```

The output of the preceding code is as follows:

	column 1	column 2
0	Looping	10
2	Looping	22
3	Functions	23
5	Functions	24

Note that rows 1, 4, and 6 are removed. Basically, both the duplicated() and drop_duplicates() methods consider all of the columns for comparison. Instead of all the columns, we could specify any subset of the columns to detect duplicated items.

4. Let's add a new column and try to find duplicated items based on the second column:

```
frame3['column 3'] = range(7)
frame5 = frame3.drop_duplicates(['column 2'])
frame5
```

The output of the preceding snippet is as follows:

	column 1	column 2	column 3
0	Looping	10	0
2	Looping	22	2
3	Functions	23	3
5	Functions	24	5

Note that both the duplicated and drop duplicates methods keep the first observed value during the duplication removal process. If we pass the take_last=True argument, the methods return the last one.

2. Replacing values

Often, it is essential to find and replace some values inside a dataframe. This can be done with the following steps:

1. We can use the replace method in such cases:

```
import numpy as np
replaceFrame = pd.DataFrame({'column 1': [200., 3000., -786., 3000.,
```

```
234., 444., -786., 332., 3332. ], 'column 2': range(9)})
replaceFrame.replace(to_replace =-786, value= np.nan)
```

The output of the preceding code is as follows:

	column 1	column 2
0	200.0	0
1	3000.0	1
2	NaN	2
3	3000.0	3
4	234.0	4
5	444.0	5
6	NaN	6
7	332.0	7
8	3332.0	8

Note that we just replaced one value with the other values. We can also replace multiple values at once.

2. In order to do so, we display them using a list:

```
replaceFrame = pd.DataFrame({'column 1': [200., 3000., -786., 3000.,
234., 444., -786., 332., 3332. ], 'column 2': range(9)})
replaceFrame.replace(to_replace =[-786, 0], value= [np.nan, 2])
```

In the preceding code, there are two replacements. All -786 values will be replaced by NaN and all 0 values will be replaced by 2. That's pretty straightforward, right?

3. Handling missing data

Whenever there are missing values, a NaN value is used, which indicates that there is no value specified for that particular index. There could be several reasons why a value could be NaN:

- It can happen when data is retrieved from an external source and there are some incomplete values in the dataset.
- It can also happen when we join two different datasets and some values are not matched.
- Missing values due to data collection errors.
- When the shape of data changes, there are new additional rows or columns that are not determined.
- Reindexing of data can result in incomplete data.

Let's see how we can work with the missing data:

1. Let's assume we have a dataframe as shown here:

```
data = np.arange(15, 30).reshape(5, 3)
dfx = pd.DataFrame(data, index=['apple', 'banana', 'kiwi', 'grapes',
```

```
'mango'], columns=['store1', 'store2', 'store3'])
dfx
```

And the output of the preceding code is as follows:

	store1	store2	store3
apple	15	16	17
banana	18	19	20
kiwi	21	22	23
grapes	24	25	26
mango	27	28	29

Assume we have a chain of fruit stores all over town. Currently, the dataframe is showing sales of different fruits from different stores. None of the stores are reporting missing values.

2. Let's add some missing values to our dataframe:

```
fx['store4'] = np.nan
dfx.loc['watermelon'] = np.arange(15, 19)
dfx.loc['oranges'] = np.nan
dfx['store5'] = np.nan
dfx['store4']['apple'] = 20.
dfx
```

And the output will now look like the following screenshot:

	store1	store2	store3	store4	store5
apple	15.0	16.0	17.0	20.0	NaN
banana	18.0	19.0	20.0	NaN	NaN
kiwi	21.0	22.0	23.0	NaN	NaN
grapes	24.0	25.0	26.0	NaN	NaN
mango	27.0	28.0	29.0	NaN	NaN
watermelon	15.0	16.0	17.0	18.0	NaN
oranges	NaN	NaN	NaN	NaN	NaN

Note that we've added two more stores, store4 and store5, and two more types of fruits, watermelon and oranges. Assume that we know how many kilos of apples and watermelons were sold from store4, but we have not collected any data from store5. Moreover, none of the stores reported sales of oranges. We are quite a huge fruit dealer, aren't we?

Note the following characteristics of missing values in the preceding dataframe:

- An entire row can contain NaN values.
- An entire column can contain NaN values.
- Some (but not necessarily all) values in both a row and a column can be NaN.

Based on these characteristics, let's examine NaN values in the next section.

NaN values in pandas objects

We can use the isnull() function from the pandas library to identify NaN values:

1. Check the following example:

```
dfx.isnull()
```

The output of the preceding code is as follows:

	store1	store2	store3	store4	store5
apple	False	False	False	False	True
banana	False	False	False	True	True
kiwi	False	False	False	True	True
grapes	False	False	False	True	True
mango	False	False	False	True	True
watermelon	False	False	False	False	True
oranges	True	True	True	True	True

Note that the True values indicate the values that are NaN. Alternatively, we can also use the notnull() method to do the same thing. The only difference would be that the function will indicate True for the values which are not null.

2. Check it out in action:

```
dfx.notnull()
```

And the output of this is as follows:

	store1	store2	store3	store4	store5
apple	True	True	True	True	False
banana	True	True	True	False	False
kiwi	True	True	True	False	False
grapes	True	True	True	False	False
mango	True	True	True	False	False
watermelon	True	True	True	True	False
oranges	False	False	False	False	False

Compare these two tables. These two functions, notnull() and isnull(), are the complement to each other.

3. We can use the sum() method to count the number of NaN values in each store. How does this work, you ask? Check the following code:

```
dfx.isnull().sum()
```

And the **output** of the preceding code is as follows:

```
store1 1
store2 1
store3 1
store4 5
store5 7
dtype: int64
```

The fact that *True* is 1 and *False* is 0 is the main logic for summing. The preceding results show that one value was not reported by store1, store2, and store3. Five values were not reported by store4 and seven values were not reported by store5.

4. We can go one level deeper to find the total number of missing values:

```
dfx.isnull().sum().sum()
```

And the **output** of the preceding code is as follows:

```
15
```

This indicates 15 missing values in our stores. We can use an alternative way to find how many values were actually reported.

5. So, instead of counting the number of missing values, we can count the number of reported values:

```
dfx.count()
```

And the **output** of the preceding code is as follows:

```
store1 6
store2 6
store3 6
store4 2
store5 0
dtype: int64
```

4. Dropping missing values

One of the ways to handle missing values is to simply remove them from our dataset. We have seen that we can use the isnull() and notnull() functions from the pandas library to determine null values:

```
dfx.store4[dfx.store4.notnull()]
```

The **output** of the preceding code is as follows:

```
apple 20.0
watermelon 18.0
Name: store4, dtype: float64
```

The output shows that store4 only reported two items of data. Now, we can use the **dropna()** method to remove the rows:

```
dfx.store4.dropna()
```

The **output** of the preceding code is as follows:

```
apple 20.0
watermelon 18.0
Name: store4, dtype: float64
```

Note that the dropna() method just returns a copy of the dataframe by dropping the rows with NaN. The original dataframe is not changed.

If dropna() is applied to the entire dataframe, then it will drop all the rows from the dataframe, because there is at least one NaN value in our dataframe:

```
dfx.dropna()
```

The output of the preceding code is an empty dataframe.

We can also drop rows that have NaN values. To do so, we can use the `how=all` argument to drop only those rows entire values are entirely NaN:

```
dfx.dropna(how='all')
```

The output of the preceding code is as follows:

	store1	store2	store3	store4	store5
apple	15.0	16.0	17.0	20.0	NaN
banana	18.0	19.0	20.0	NaN	NaN
kiwi	21.0	22.0	23.0	NaN	NaN
grapes	24.0	25.0	26.0	NaN	NaN
mango	27.0	28.0	29.0	NaN	NaN
watermelon	15.0	16.0	17.0	18.0	NaN

Note that only the orange rows are removed because those entire rows contained NaN values.

5. Dropping by columns

Furthermore, we can also pass axis=1 to indicate a check for NaN by columns.

Check the following example:

```
dfx.dropna(how='all', axis=1)
```

And the output of the preceding code is as follows:

	store1	store2	store3	store4
apple	15.0	16.0	17.0	20.0
banana	18.0	19.0	20.0	NaN
kiwi	21.0	22.0	23.0	NaN
grapes	24.0	25.0	26.0	NaN
mango	27.0	28.0	29.0	NaN
watermelon	15.0	16.0	17.0	18.0
oranges	NaN	NaN	NaN	NaN

Note that `store5` is dropped from the dataframe. By passing in `axis=1`, we are instructing pandas to drop columns if all the values in the column are `NaN`. Furthermore, we can also pass another argument, `thresh`, to specify a minimum number of NaNs that must exist before the column should be dropped:

```
dfx.dropna(thresh=5, axis=1)
```

And the output of the preceding code is as follows:

	store1	store2	store3
apple	15.0	16.0	17.0
banana	18.0	19.0	20.0
kiwi	21.0	22.0	23.0
grapes	24.0	25.0	26.0
mango	27.0	28.0	29.0
watermelon	15.0	16.0	17.0
oranges	NaN	NaN	NaN

Compared to the preceding, note that even the store4 column is now dropped because it has more than five NaN values

6. Mathematical operations with NaN

The pandas and numpy libraries handle NaN values differently for mathematical operations.

Consider the following example:

```
ar1 = np.array([100, 200, np.nan, 300])
ser1 = pd.Series(ar1)
ar1.mean(), ser1.mean()
```

The **output** of the preceding code is the following:

```
(nan, 200.0)
```

Note the following things:

- When a NumPy function encounters NaN values, it returns NaN.
- Pandas, on the other hand, ignores the NaN values and moves ahead with processing. When performing the sum operation, NaN is treated as 0. If all the values are NaN, the result is also NaN.

Let's compute the total quantity of fruits sold by store4:

```
ser2 = dfx.store4
ser2.sum()
```

The **output** of the preceding code is as follows:

```
38.0
```

Note that store4 has five NaN values. However, during the summing process, these values are treated as 0 and the result is 38.0.

Similarly, we can compute averages as shown here:

```
ser2.mean()
```

The **output** of the code is the following: 19.0

Note that NaNs are treated as 0s. It is the same for cumulative summing:

```
ser2.cumsum()
```

And the **output** of the preceding code is as follows:

```
apple 20.0
banana NaN
kiwi NaN
grapes NaN
mango NaN
watermelon 38.0
oranges NaN
Name: store4, dtype: float64
```

Note that only actual values are affected in computing the cumulative sum.

7. Filling missing values

We can use the fillna() method to replace NaN values with any particular values.

Check the following example:

```
filledDf = dfx.fillna(0)
filledDf
```

The **output** of the preceding code is shown in the following screenshot:

	store1	store2	store3	store4	store5
apple	15.0	16.0	17.0	20.0	0.0
banana	18.0	19.0	20.0	0.0	0.0
kiwi	21.0	22.0	23.0	0.0	0.0
grapes	24.0	25.0	26.0	0.0	0.0
mango	27.0	28.0	29.0	0.0	0.0
watermelon	15.0	16.0	17.0	18.0	0.0
oranges	0.0	0.0	0.0	0.0	0.0

Note that in the preceding dataframe, all the `NaN` values are replaced by `0`. Replacing the values with `0` will affect several statistics including mean, sum, and median.

Check the difference in the following two examples:

```
dfx.mean()
```

And the output of the preceding code is as follows:

```
store1 20.0
store2 21.0
store3 22.0
store4 19.0
store5 NaN
dtype: float64
```

Now, let's compute the mean from the filled dataframe with the following command:

```
filledDf.mean()
```

And the **output** we get is as follows:

```
store1 17.142857
store2 18.000000
store3 18.857143
store4 5.428571
store5 0.000000
dtype: float64
```

Note that there are slightly different values. Hence, filling with `0` might not be the optimal solution.

8. **Backward and forward filling**

NaN values can be filled based on the last known values. To understand this, let's consider taking our store dataframe as an example.

We want to fill store4 using the forward-filling technique:

```
dfx.store4.fillna(method='ffill')
```

And the **output** of the preceding code is the following:

```
apple 20.0
banana 20.0
kiwi 20.0
grapes 20.0
mango 20.0
watermelon 18.0
oranges 18.0
Name: store4, dtype: float64
```

Here, from the forward-filling technique, the last known value is 20 and hence the rest of the NaN values are replaced by it.

The direction of the fill can be changed by changing method='bfill'. Check the following example:

```
dfx.store4.fillna(method='bfill')
```

And the **output** of the preceding code is as follows:

```
apple 20.0
banana 18.0
kiwi 18.0
grapes 18.0
mango 18.0
watermelon 18.0
oranges NaN
Name: store4, dtype: float64
```

Note here that the NaN values are replaced by 18.0.

9. Interpolating missing values

The pandas library provides the interpolate() function both for the series and the dataframe. By default, it performs a linear interpolation of our missing values. Check the following example:

```
ser3 = pd.Series([100, np.nan, np.nan, np.nan, 292])
ser3.interpolate()
```

And the output of the preceding code is the following:

```
0 100.0
1 148.0
2 196.0
3 244.0
4 292.0
dtype: float64
```

The first value before and after any sequence of the NaN values. In the preceding series, ser3, the first and the last values are 100 and 292 respectively. Hence, it calculates the next value as *(292-100)/(5-1) = 48*. So, the next value after 100 is *100 + 48 = 148*.

10. Renaming axis indexes

Consider the example from the *Reshaping and pivoting* section. Say you want to transform the index terms to capital letters:

```
dframe1.index = dframe1.index.map(str.upper)
dframe1
```

The **output** of the preceding code is as follows:

	Bergen	Oslo	Trondheim	Stavanger	Kristiansand
RAINFALL	0	1	2	3	4
HUMIDITY	5	6	7	8	9
WIND	10	11	12	13	14

Note that the indexes have been capitalized. If we want to create a transformed version of the dataframe, then we can use the `rename()` method. This method is handy when we do not want to modify the original data. Check the following example:

```
dframe1.rename(index=str.title, columns=str.upper)
```

And the **output** of the code is as follows:

	BERGEN	OSLO	TRONDHEIM	STAVANGER	KRISTIANSAND
Rainfall	0	1	2	3	4
Humidity	5	6	7	8	9
Wind	10	11	12	13	14

The rename method does not make a copy of the dataframe.

11. Discretization and binning

Often when working with continuous datasets, we need to convert them into discrete or interval forms. Each interval is referred to as a bin, and hence the name *binning* comes into play:

1. Let's say we have data on the heights of a group of students as follows:

```
height = [120, 122, 125, 127, 121, 123, 137, 131, 161, 145, 141, 132]
```

And we want to convert that dataset into intervals of 118 to 125, 126 to 135, 136 to 160, and finally 160 and higher.

2. To convert the preceding dataset into intervals, we can use the cut() method provided by the pandas library:

```
bins = [118, 125, 135, 160, 200]
category = pd.cut(height, bins)
category
```

The **output** of the preceding code is as follows:

```
[(118, 125], (118, 125], (118, 125], (125, 135], (118, 125], ..., (125, 135], (160, 200], (135, 160], (135, 160], (125, 135]] Length: 12 Categories (4, interval[int64]): [(118, 125] < (125, 135] < (135, 160] < (160, 200]]
```

If you look closely at the output, you'll see that there are mathematical notations for intervals. Do you recall what these parentheses mean from your elementary mathematics class? If not, here is a quick recap:

- A parenthesis indicates that the side is open.
- A square bracket means that it is closed or inclusive.

From the preceding code block, `(118, 125]` means the left-hand side is open and the right-hand side is closed. This is mathematically denoted as follows:

	Bergen	Oslo	Trondheim	Stavanger	Kristiansand
RAINFALL	0	1	2	3	4
HUMIDITY	5	6	7	8	9
WIND	10	11	12	13	14

Hence, `118` is not included, but anything greater than `118` is included, while `125` is included in the interval.

3. We can set a right=False argument to change the form of interval:

```
category2 = pd.cut(height, [118, 126, 136, 161, 200], right=False)
category2
```

And the **output** of the preceding code is as follows:

[[118, 126), [118, 126), [118, 126), [126, 136), [118, 126), ..., [126, 136), [161, 200), [136, 161), [136, 161), [126, 136)] Length: 12 Categories (4, interval[int64]): [[118, 126) < [126, 136) < [136, 161) < [161, 200)]

Note that the output form of closeness has been changed. Now, the results are in the form of *right-closed, left-open*.

4. We can check the number of values in each bin by using the pd.value_counts() method:

```
pd.value_counts(category)
```

And the **output** is as follows:

```
(118, 125] 5
(135, 160] 3
(125, 135] 3
(160, 200] 1
dtype: int64
```

The output shows that there are five values in the interval `[118-125)`.

5. We can also indicate the bin names by passing a list of labels:

```
bin_names = ['Short Height', 'Average height', 'Good Height', 'Taller']
pd.cut(height, bins, labels=bin_names)
```

And the **output** is as follows:

```
[Short Height, Short Height, Short Height, Average height, Short Height, ..., Average height, Taller, Good Height, Good Height, Average height]
Length: 12
Categories (4, object): [Short Height < Average height < Good Height < Taller]
```

Note that we have passed at least two arguments, the data that needs to be discretized and the required number of bins. Furthermore, we have used a `right=False` argument to change the form of interval.

6. Now, it is essential to note that if we pass just an integer for our bins, it will compute equal-length bins based on the minimum and maximum values in the data. Okay, let's verify what we mentioned here:

```
import numpy as np
pd.cut(np.random.rand(40), 5, precision=2)
```

In the preceding code, we have just passed `5` as the number of required bins, and the **output** of the preceding code is as follows:

```
[(0.81, 0.99], (0.094, 0.27], (0.81, 0.99], (0.45, 0.63], (0.63, 0.81], ..., (0.81, 0.99], (0.45, 0.63], (0.45, 0.63], (0.81, 0.99], (0.81, 0.99]] Length: 40
Categories (5, interval[float64]): [(0.094, 0.27] < (0.27, 0.45] < (0.45, 0.63] < (0.63, 0.81] < (0.81, 0.99]]
```

Pandas provides a `qcut` method that forms the bins based on sample quantiles. Let's check this with an example:

randomNumbers = np.random.rand(2000)
category3 = pd.qcut(randomNumbers, 4) # cut into quartiles
category3

And the **output** of the preceding code is as follows:

[(0.77, 0.999], (0.261, 0.52], (0.261, 0.52], (-0.000565, 0.261], (-0.000565, 0.261], ..., (0.77, 0.999], (0.77, 0.999], (0.261, 0.52], (-0.000565, 0.261], (0.261, 0.52]]
Length: 2000
Categories (4, interval[float64]): [(-0.000565, 0.261] < (0.261, 0.52] < (0.52, 0.77] < (0.77, 0.999]]

Note that based on the number of bins, which we set to 4, it converted our data into four different categories. If we count the number of values in each category, we should get equal-sized bins as per our definition. Let's verify that with the following command:

pd.value_counts(category3)

And the **output** of the command is as follows:

0.77, 0.999] 500
(0.52, 0.77] 500
(0.261, 0.52] 500
(-0.000565, 0.261] 500
dtype: int64

Our claim is hence verified. Each category contains an equal size of 500 values. Note that, similar to cut, we can also pass our own bins:

pd.qcut(randomNumbers, [0, 0.3, 0.5, 0.7, 1.0])

And the **output** of the preceding code is as follows:

[(0.722, 0.999], (-0.000565, 0.309], (0.309, 0.52], (-0.000565, 0.309], (-0.000565, 0.309], ..., (0.722, 0.999], (0.722, 0.999], (0.309, 0.52], (-0.000565, 0.309], (0.309, 0.52]] Length: 2000
Categories (4, interval[float64]): [(-0.000565, 0.309] < (0.309, 0.52] < (0.52, 0.722] < (0.722, 0.999]]

Note that it created four different categories based on our code. Congratulations! We successfully learned how to convert continuous datasets into discrete datasets.

12. Outlier detection and filtering

Outliers are data points that diverge from other observations for several reasons. During the EDA phase, one of our common tasks is to detect and filter these outliers. The main reason for this detection and filtering of outliers is that the presence of such outliers can cause serious issues in statistical analysis. In this section, we are going to perform simple outlier detection and filtering. Let's get started:

1. Load the dataset that is available from the GitHub link as follows:

   ```
   df = pd.read_csv('https://raw.githubusercontent.com/PacktPublishing/hands-on-exploratory-data-analysis-with-python/master/Chapter%204/sales.csv')
   df.head(10)
   ```

The dataset was synthesized manually by creating a script. If you are interested in looking at how we created the dataset, the script can be found inside the folder named Chapter 4 in the GitHub repository shared with this book.

The output of the preceding df.head(10) command is shown in the following screenshot:

	Account	Company	Order	SKU	Country	Year	Quantity	UnitPrice	transactionComplete
0	123456779	Kulas Inc	99985	s9-supercomputer	Aruba	1981	5148	545	False
1	123456784	GitHub	99986	s4-supercomputer	Brazil	2001	3262	383	False
2	123456782	Kulas Inc	99990	s10-supercomputer	Montserrat	1973	9119	407	True
3	123456783	My SQ Man	99999	s1-supercomputer	El Salvador	2015	3097	615	False
4	123456787	ABC Dogma	99996	s6-supercomputer	Poland	1970	3356	91	True
5	123456778	Super Sexy Dingo	99996	s9-supercomputer	Costa Rica	2004	2474	136	True
6	123456783	ABC Dogma	99981	s11-supercomputer	Spain	2006	4081	195	False
7	123456785	ABC Dogma	99998	s9-supercomputer	Belarus	2015	6576	603	False
8	123456778	Loolo INC	99997	s8-supercomputer	Mauritius	1999	2460	36	False
9	123456775	Kulas Inc	99997	s7-supercomputer	French Guiana	2004	1831	664	True

2. Now, suppose we want to calculate the total price based on the quantity sold and the unit price. We can simply add a new column, as shown here:

   ```
   df['TotalPrice'] = df['UnitPrice'] * df['Quantity']
   df
   ```

This should add a new column called TotalPrice, as shown in the following screenshot:

	Account	Company	Order	SKU	Country	Year	Quantity	UnitPrice	transactionComplete	TotalPrice
0	123456779	Kulas Inc	99985	s9-supercomputer	Aruba	1981	5148	545	False	2805660
1	123456784	GitHub	99986	s4-supercomputer	Brazil	2001	3262	383	False	1249346
2	123456782	Kulas Inc	99990	s10-supercomputer	Montserrat	1973	9119	407	True	3711433
3	123456783	My SQ Man	99999	s1-supercomputer	El Salvador	2015	3097	615	False	1904655
4	123456787	ABC Dogma	99996	s6-supercomputer	Poland	1970	3356	91	True	305396
5	123456778	Super Sexy Dingo	99996	s9-supercomputer	Costa Rica	2004	2474	136	True	336464
6	123456783	ABC Dogma	99981	s11-supercomputer	Spain	2006	4081	195	False	795795
7	123456785	ABC Dogma	99998	s9-supercomputer	Belarus	2015	6576	603	False	3965328
8	123456778	Loolo INC	99997	s8-supercomputer	Mauritius	1999	2460	36	False	88560
9	123456775	Kulas Inc	99997	s7-supercomputer	French Guiana	2004	1831	664	True	1215784

Now, let's answer some questions based on the preceding table.

Let's find the transaction that exceeded 3,000,000:

```
TotalTransaction = df["TotalPrice"]
TotalTransaction[np.abs(TotalTransaction) > 3000000]
```

The **output** of the preceding code is as follows:

```
2 3711433
7 3965328
13 4758900
15 5189372
17 3989325
    ...
9977 3475824
9984 5251134
9987 5670420
9991 5735513
9996 3018490
Name: TotalPrice, Length: 2094, dtype: int64
```

Note that, in the preceding example, we have assumed that any price greater than 3,000,000 is an outlier.

Display all the columns and rows from the preceding table if TotalPrice is greater than 6741112, as follows:

```
df[np.abs(TotalTransaction) > 6741112]
```

The **output** of the preceding code is the following:

	Account	Company	Order	SKU	Country	Year	Quantity	UnitPrice	transactionComplete	TotalPrice
818	123456781	Gen Power	99991	s1-supercomputer	Burkina Faso	1985	9693	696	False	6746328
1402	123456778	Will LLC	99985	s11-supercomputer	Austria	1990	9844	695	True	6841580
2242	123456770	Name IT	99997	s9-supercomputer	Myanmar	1979	9804	692	False	6784368
2876	123456772	Gen Power	99992	s10-supercomputer	Mali	2007	9935	679	False	6745865
3210	123456782	Loolo INC	99991	s8-supercomputer	Kuwait	2006	9886	692	False	6841112
3629	123456779	My SQ Man	99980	s3-supercomputer	Hong Kong	1994	9694	700	False	6785800
7674	123456781	Loolo INC	99989	s6-supercomputer	Sri Lanka	1994	9882	691	False	6828462
8645	123456789	Gen Power	99996	s11-supercomputer	Suriname	2005	9742	699	False	6809658
8684	123456785	Gen Power	99989	s2-supercomputer	Kenya	2013	9805	694	False	6804670

Note that in the output, all the TotalPrice values are greater than 6741112. We can use any sort of conditions, either row-wise or column-wise, to detect and filter outliers.

13. Permutation and random sampling

Well, now we have some more mathematical terms to learn: *permutation* and *random sampling*. Let's examine how we can perform permutation and random sampling using the pandas library:

1. With NumPy's numpy.random.permutation() function, we can randomly select or permute a series of rows in a dataframe. Let's understand this with an example:

```
dat = np.arange(80).reshape(10,8)
df = pd.DataFrame(dat)
df
```

And the output of the preceding code is as follows:

	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	8	9	10	11	12	13	14	15
2	16	17	18	19	20	21	22	23
3	24	25	26	27	28	29	30	31
4	32	33	34	35	36	37	38	39
5	40	41	42	43	44	45	46	47
6	48	49	50	51	52	53	54	55
7	56	57	58	59	60	61	62	63
8	64	65	66	67	68	69	70	71
9	72	73	74	75	76	77	78	79

2. Next, we call the np.random.permutation() method. This method takes an argument – the length of the axis we require to be permuted – and gives an array of integers indicating the new ordering:

```
sampler = np.random.permutation(10)
sampler
```

The **output** of the preceding code is as follows:

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

3. The preceding output array is used in ix-based indexing for the take() function from the pandas library. Check the following example for clarification:

```
df.take(sampler)
```

The **output** of the preceding code is as follows:

	0	1	2	3	4	5	6	7
1	8	9	10	11	12	13	14	15
5	40	41	42	43	44	45	46	47
3	24	25	26	27	28	29	30	31
6	48	49	50	51	52	53	54	55
2	16	17	18	19	20	21	22	23
4	32	33	34	35	36	37	38	39
9	72	73	74	75	76	77	78	79
0	0	1	2	3	4	5	6	7
7	56	57	58	59	60	61	62	63
8	64	65	66	67	68	69	70	71

It is essential that you understand the output. Note that our sampler array contains array([1, 5, 3, 6, 2, 4, 9, 0, 7, 8]). Each of these array items represents the rows of the original dataframe. So, from the original dataframe, it pulls the first row, then the fifth row, then the third row, and so on. Compare this with the original dataframe output and it will make more sense.

Random sampling without replacement

To compute random sampling without replacement, follow these steps:

1. To perform random sampling without replacement, we first create a permutation array.
2. Next, we slice off the first *n* elements of the array where *n* is the desired size of the subset you want to sample.
3. Then we use the df.take() method to obtain actual samples:

```
df.take(np.random.permutation(len(df))[:3])
```

The **output** of the preceding code is as follows:

	0	1	2	3	4	5	6	7
9	72	73	74	75	76	77	78	79
2	16	17	18	19	20	21	22	23
0	0	1	2	3	4	5	6	7

Note that in the preceding code, we only specified a sample of size 3. Hence, we only get three rows in the random sample.

Random sampling with replacement

To generate random sampling with replacement, follow the given steps:

1. We can generate a random sample with replacement using the numpy.random.randint() method and drawing random integers:

```
sack = np.array([4, 8, -2, 7, 5])
sampler = np.random.randint(0, len(sack), size = 10)
sampler
```

We created the sampler using the np.random.randint() method. The **output** of the preceding code is as follows:

```
array([3, 3, 0, 4, 0, 0, 1, 2, 1, 4])
```

2. And now, we can draw the required samples:

```
draw = sack.take(sampler)
draw
```

The **output** of the preceding code is as follows:

```
array([ 7,  7,  4,  5,  4,  4,  8, -2,  8,  5])
```

Compare the index of the sampler and then compare it with the original dataframe. The results are pretty obvious in this case.

14. Computing indicators/dummy variables

Often, we need to convert a categorical variable into some dummy matrix. Especially for statistical modeling or machine learning model development, it is essential to create dummy variables. Let's get started:

1. Let's say we have a dataframe with data on gender and votes, as shown here:

```
df = pd.DataFrame({'gender': ['female', 'female', 'male', 'unknown', 'male',
'female'], 'votes': range(6, 12, 1)})
df
```

The output of the preceding code is as follows:

	gender	votes
0	female	6
1	female	7
2	male	8
3	unknown	9
4	male	10
5	female	11

So far, nothing too complicated. Sometimes, however, we need to encode these values in a matrix form with 1 and 0 values.

2. We can do that using the pd.get_dummies() function:

```
pd.get_dummies(df['gender'])
```

And the **output** of the preceding code is as follows:

	female	male	unknown
0	1	0	0
1	1	0	0
2	0	1	0
3	0	0	1
4	0	1	0
5	1	0	0

Note the pattern. There are five values in the original dataframe with three unique values of male, female, and unknown. Each unique value is transformed into a column and each original value into a row. For example, in the original dataframe, the first value is female, hence it is added as a row with 1 in the female value and the rest of them are 0 values, and so on.

3. Sometimes, we want to add a prefix to the columns. We can do that by adding the prefix argument, as shown here:

```
dummies = pd.get_dummies(df['gender'], prefix='gender')
dummies
```

The **output** of the preceding code is as follows:

	gender_female	gender_male	gender_unknown
0	1	0	0
1	1	0	0
2	0	1	0
3	0	0	1
4	0	1	0
5	1	0	0

Note the gender prefix added to each of the column names. Not that difficult, right? Great work so far.

Benefits of data transformation

- Data transformation promotes interoperability between several applications. The main reason for creating a similar format and structure in the dataset is that it becomes compatible with other systems.
- Comprehensibility for both humans and computers is improved when using better-organized data compared to messier data.
- Data transformation ensures a higher degree of data quality and protects applications from several computational challenges such as null values, unexpected duplicates, and incorrect indexings, as well as incompatible structures or formats.
- Data transformation ensures higher performance and scalability for modern analytical databases and dataframes.

Challenges

- It requires a qualified team of experts and state-of-the-art infrastructure. The cost of attaining such experts and infrastructure can increase the **cost of the operation**.
- Data transformation requires data cleaning before data transformation and data migration. This process of cleansing can be expensively **time-consuming**.
- Generally, the activities of data transformations involve batch processing. This means that sometimes, we might have to wait for a day before the next batch of data is ready for cleansing. This can be very **slow**.

GROUPING DATASETS

- During data analysis, it is often essential to cluster or group data together based on certain criteria. For example, an e-commerce store might want to group all the sales that were done during the Christmas period or the orders that were received on Black Friday.
- These grouping concepts occur in several parts of data analysis.
- Different groupby() mechanics that will accumulate our dataset into various classes that we can perform aggregation on.

In this chapter, we will cover the following topics:

- Understanding groupby()
- Groupby mechanics
- Data aggregation
- Pivot tables and cross-tabulations

Technical requirements

- The code for this chapter can be found in this book's GitHub repository, https://github.com/PacktPublishing/hands-on-exploratory-data-analysis-with-python
- The dataset we'll be using in this chapter is available under open access through Kaggle. It can be downloaded from https://www.kaggle.com/toramky/automobile-dataset.

Understanding groupby()

- During the data analysis phase, categorizing a dataset into multiple categories or groups is often essential. We can do such categorization using the pandas library.
- The pandas groupby function is one of the most efficient and time-saving features for doing this. Groupby provides functionalities that allow us to split-apply-combine throughout the dataframe;
- that is, this function can be used for splitting, applying, and combining dataframes.
- Similar to the **Structured Query Language** (**SQL**), we can use pandas and Python to execute more complex group operations by using any built-in functions that accept the pandas object or the numpy array.

Groupby mechanics

While working with the pandas dataframes, our analysis may require us to split our data by certain criteria. Groupby mechanics amass our dataset into various classes in which we can perform exercises and make changes, such as the following:

- Grouping by features, hierarchically
- Aggregating a dataset by groups
- Applying custom aggregation functions to groups
- Transforming a dataset groupwise

The pandas groupby method performs two essential functions:

- It splits the data into groups based on some criteria.
- It applies a function to each group independently.

To work with groupby functionalities, we need a dataset that has multiple numerical as well as categorical records in it so that we can group by different categories and ranges.

1. Let's start by importing the required Python libraries and datasets:

```
import pandas as pd
df = pd.read_csv("/content/automobileEDA.csv")
df.head()
```

The **output** of the preceding code is as follows:

	symboling	normalized-losses	make	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	length	width	height	curb-weight	engine-type	num-of-cylinders	engine-size
0	3	122	alfa-romero	std	two	convertible	rwd	front	88.6	0.811148	0.890278	48.8	2548	dohc	four	130
1	3	122	alfa-romero	std	two	convertible	rwd	front	88.6	0.811148	0.890278	48.8	2548	dohc	four	130
2	1	122	alfa-romero	std	two	hatchback	rwd	front	94.5	0.822681	0.909722	52.4	2823	ohcv	six	152
3	2	164	audi	std	four	sedan	fwd	front	99.8	0.848630	0.919444	54.3	2337	ohc	four	109
4	2	164	audi	std	four	sedan	4wd	front	99.4	0.848630	0.922222	54.3	2824	ohc	five	136

As you can see, there are multiple columns with categorical variables.

2. Using the groupby() function lets us group this dataset on the basis of the body-style column:

```
df.groupby('body-style').groups.keys()
```

The **output** of the preceding code is as follows:

```
dict_keys(['convertible', 'hardtop', 'hatchback', 'sedan', 'wagon'])
```

From the preceding output, we know that the body-style column has five unique values, including convertible, hardtop, hatchback, sedan, and wagon.

3. Now, we can group the data based on the body-style column. Next, let's print the values contained in that group that have the body-style value of convertible. This can be done using the following code:

```
# Group the dataset by the column body-style
style = df.groupby('body-style')

# Get values items from group with value convertible
style.get_group("convertible")
```

The **output** of the preceding code is as follows:

	symboling	normalized-losses	make	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	length	width	height	curb-weight	engine-type	num-of-cylinders	engine-size
0	3	122	alfa-romero	std	two	convertible	rwd	front	88.6	0.811148	0.890278	48.8	2548	dohc	four	130
1	3	122	alfa-romero	std	two	convertible	rwd	front	88.6	0.811148	0.890278	48.8	2548	dohc	four	130
69	3	142	mercedes-benz	std	two	convertible	rwd	front	96.6	0.866410	0.979167	50.8	3685	ohcv	eight	234
125	3	122	porsche	std	two	convertible	rwd	rear	89.5	0.811629	0.902778	51.6	2800	ohcf	six	194
168	2	134	toyota	std	two	convertible	rwd	front	98.4	0.846708	0.911111	53.0	2975	ohc	four	146
185	3	122	volkswagen	std	two	convertible	fwd	front	94.5	0.765497	0.891667	55.6	2254	ohc	four	109

In the preceding example, we have grouped by using a single body-style column.

Selecting a subset of columns

To form groups based on multiple categories, we can simply specify the column names in the groupby() function. Grouping will be done simultaneously with the first category, the second category, and so on.

Let's groupby using two categories, body-style and drive wheels, as follows:

```
double_grouping = df.groupby(["body-style","drive-wheels"])
double_grouping.first()
```

The **output** of the preceding code is as follows:

body-style	drive-wheels	symboling	normalized-losses	make	aspiration	num-of-doors	engine-location	wheel-base	length	width	height	curb-weight	engine-type	num-of-cylinders	engine-size	fue syst
convertible	fwd	3	122	volkswagen	std	two	front	94.5	0.765497	0.891667	55.6	2254	ohc	four	109	m
	rwd	3	122	alfa-romero	std	two	front	88.6	0.811148	0.890278	48.8	2548	dohc	four	130	m
hardtop	fwd	2	168	nissan	std	two	front	95.1	0.780394	0.886111	53.3	2008	ohc	four	97	2b
	rwd	0	93	mercedes-benz	turbo	two	front	106.7	0.901009	0.976389	54.9	3495	ohc	five	183	
hatchback	4wd	2	83	subaru	std	two	front	93.3	0.755887	0.886111	55.7	2240	ohcf	four	108	2b
	fwd	2	121	chevrolet	std	two	front	88.4	0.678039	0.837500	53.2	1488	l	three	61	2b
	rwd	1	122	alfa-romero	std	two	front	94.5	0.822681	0.909722	52.4	2823	ohcv	six	152	m
sedan	4wd	2	164	audi	std	four	front	99.4	0.848630	0.922222	54.3	2824	ohc	five	136	m
	fwd	2	164	audi	std	four	front	99.8	0.848630	0.919444	54.3	2337	ohc	four	109	m
	rwd	2	192	bmw	std	two	front	101.2	0.849592	0.900000	54.3	2395	ohc	four	108	m
wagon	4wd	0	85	subaru	std	four	front	96.9	0.834214	0.908333	54.9	2420	ohcf	four	108	2b
	fwd	1	122	audi	std	four	front	105.8	0.925997	0.991667	55.7	2954	ohc	five	136	m
	rwd	-1	93	mercedes-benz	turbo	four	front	110.0	0.917347	0.976389	58.7	3750	ohc	five	183	

Not only can we group the dataset with specific criteria, but we can also perform arithmetic operations directly on the whole group at the same time and print the output as a series or dataframe.

There are functions such as max(), min(), mean(), first(), and last() that can be directly applied to the GroupBy object in order to obtain summary statistics for each group.

Max and min

Let's compute the maximum and minimum entry for each group. Here, we will find the maximum and minimum for the normalized-losses column:

```
# max() will print the maximum entry of each group
style['normalized-losses'].max()
```

```
# min() will print the minimum entry of each group
style['normalized-losses'].min()
```

The **output** of the preceding code is as follows:

```
body-style
convertible 122
hardtop 93
hatchback 65
sedan 65
wagon 74
Name: normalized-losses, dtype: int64
```

As illustrated in the preceding output, the minimum value for each category is presented.

Mean

We can find the mean values for the numerical column in each group. This can be done using the df.mean() method.

The code for finding the mean is as follows:

```
# mean() will print mean of numerical column in each group
style.mean()
```

The **output** of the preceding code is as follows:

body-style	symboling	normalized-losses	wheel-base	length	width	height	curb-weight	engine-size	bore	stroke	compression-ratio	horsepower
convertible	2.833333	127.333333	92.700000	0.818757	0.910880	51.433333	2801.666667	157.166667	3.491667	3.043333	8.933333	131.666667
hardtop	1.875000	128.625000	98.500000	0.850252	0.925174	52.850000	2810.625000	176.250000	3.608750	3.322500	10.725000	142.250000
hatchback	1.617647	130.897059	95.435294	0.799078	0.904228	52.133824	2322.852941	112.852941	3.236015	3.280312	9.042941	97.768473
sedan	0.329787	120.893617	100.750000	0.855583	0.921070	54.387234	2625.893617	131.691489	3.345106	3.270638	10.965957	103.808511
wagon	-0.160000	98.560000	102.156000	0.871235	0.920222	56.728000	2784.240000	123.840000	3.406400	3.175600	10.316000	98.010246

get the average of each column by specifying a column, as follows:

```
# get mean of each column of specific group
style.get_group("convertible").mean()
```

The output of the preceding code is as follows:

```
symboling                 2.833333
normalized-losses       127.333333
wheel-base               92.700000
length                    0.818757
width                     0.910880
height                   51.433333
curb-weight            2801.666667
engine-size             157.166667
bore                      3.491667
stroke                    3.043333
compression-ratio         8.933333
horsepower              131.666667
peak-rpm               5158.333333
city-mpg                 20.500000
highway-mpg              26.000000
price                 21890.500000
city-L/100km             11.745886
diesel                    0.000000
gas                       1.000000
dtype: float64
```

Next, we can also **count the number of symboling/records** in each group. To do so, use the following code:

```
# get the number of symboling/records in each group
style['symboling'].count()
```

The **output** of the preceding code is as follows:

```
body-style
convertible 6
hardtop 8
hatchback 68
sedan 94
wagon 25
Name: symboling, dtype: int64
```

DATA AGGREGATION

Aggregation is the process of implementing any mathematical operation on a dataset or a subset of it. Aggregation is one of the many techniques in pandas that's used to manipulate the data in the dataframe for data analysis.

The Dataframe.aggregate() function is used to apply aggregation across one or more columns. Some of the most frequently used aggregations are as follows:

- sum: Returns the sum of the values for the requested axis
- min: Returns the minimum of the values for the requested axis
- max: Returns the maximum of the values for the requested axis

We can apply aggregation in a DataFrame, df, as df.aggregate() or df.agg().

Since aggregation only works with numeric type columns, let's take some of the numeric columns from the dataset and apply some aggregation functions to them:

```
# new dataframe that consist length,width,height,curb-weight and price
new_dataset = df.filter(["length","width","height","curb-weight","price"],axis=1)
new_dataset
```

The **output** of the preceding code snippet is as follows:

	length	width	height	curb-weight	price
0	0.811148	0.890278	48.8	2548	13495.0
1	0.811148	0.890278	48.8	2548	16500.0
2	0.822681	0.909722	52.4	2823	16500.0
3	0.848630	0.919444	54.3	2337	13950.0
4	0.848630	0.922222	54.3	2824	17450.0
...	...	...	...	...	...
196	0.907256	0.956944	55.5	2952	16845.0
197	0.907256	0.955556	55.5	3049	19045.0
198	0.907256	0.956944	55.5	3012	21485.0
199	0.907256	0.956944	55.5	3217	22470.0
200	0.907256	0.956944	55.5	3062	22625.0

201 rows × 5 columns

Next, let's apply a single aggregation to get the mean of the columns. To do this, we can use the agg() method, as shown in the following code:

```
# applying single aggregation for mean over the columns
new_dataset.agg("mean", axis="rows")
```

The **output** of the preceding code is as follows:

```
length 0.837102
width 0.915126
height 53.766667
curb-weight 2555.666667
price 13207.129353
dtype: float64
```

We can aggregate more than one function together. For example, we can find the sum and the minimum of all the columns at once by using the following code:

```
# applying aggregation sum and minimum across all the columns
new_dataset.agg(['sum', 'min'])
```

The **output** of the preceding code is as follows:

	length	width	height	curb-weight	price
sum	168.257568	183.940278	10807.1	513689	2654633.0
min	0.678039	0.837500	47.8	1488	5118.0

The output is a dataframe with rows containing the result of the respective aggregation that was applied to the columns. To apply aggregation functions across different columns, you can pass a dictionary with a key containing the column names and values containing the list of aggregation functions for any specific column:

```
# find aggregation for these columns
new_dataset.aggregate({"length":['sum', 'min'],
          "width":['max', 'min'],
          "height":['min', 'sum'],
          "curb-weight":['sum']})
# if any specific aggregation is not applied on a column
# then it has NaN value corresponding to it
```

The **output** of the preceding code is as follows:

	length	width	height	curb-weight
max	NaN	1.0000	NaN	NaN
min	0.678039	0.8375	47.8	NaN
sum	168.257568	NaN	10807.1	513689.0

Check the preceding output. The maximum, minimum, and the sum of rows present the values for each column. Note that some values are NaN based on their column values.

Group-wise operations in data aggregation

- The most important operations groupBy implements are aggregate, filter, transform and apply.
- An efficient way of implementing aggregation functions in the dataset is by doing so after grouping the required columns.
- The aggregated function will return a single aggregated value for each group.
- Once these groups have been created, several aggregation operations are applied to that grouped data.
- It is possible to group the DataFrame, df, by passing a dictionary of aggregation functions:

```
# Group the data frame df by body-style and drive-wheels and extract stats
from each group
df.groupby( ["body-style","drive-wheels"]).agg(
   { 'height':min, # minimum height of car in each group
      'length': max, # maximum length of car in each group
      'price': 'mean', # average price of car in each group })
```

The **output** of the preceding code is as follows:

		height	length	price
body-style	drive-wheels			
convertible	fwd	55.6	0.765497	11595.000000
	rwd	48.8	0.866410	23949.600000
hardtop	fwd	53.3	0.780394	8249.000000
	rwd	51.6	0.957232	24202.714286
hatchback	4wd	55.7	0.755887	7603.000000
	fwd	49.4	0.896684	8396.387755
	rwd	49.6	0.881788	14337.777778
sedan	4wd	54.3	0.848630	12647.333333
	fwd	50.6	0.925997	9811.800000
	rwd	47.8	1.000000	21711.833333
wagon	4wd	54.9	0.834214	9095.750000
	fwd	53.0	0.925997	9997.333333
	rwd	54.1	0.955790	16994.222222

- The preceding code groups the dataframe according to body-style and then driver-wheels.
- The aggregate functions are applied to the height, length, and price columns, which return the minimum height, maximum length, and average price in the respective groups.
- We can make an aggregation dictionary of functions we want to perform in groups, and then use it later:

```
# create dictionary of aggregations
aggregations=({
        'height':min, # minimum height of car in each group
        'length': max, # maximum length of car in each group
        'price': 'mean', # average price of car in each group})
# implementing aggregations in groups
df.groupby(["body-style","drive-wheels"]).agg(aggregations)
```

The **output** of the preceding code is as follows:

body-style	drive-wheels	height	length	price
convertible	fwd	55.6	0.765497	11595.000000
	rwd	48.8	0.866410	23949.600000
hardtop	fwd	53.3	0.780394	8249.000000
	rwd	51.6	0.957232	24202.714286
hatchback	4wd	55.7	0.755887	7603.000000
	fwd	49.4	0.896684	8396.387755
	rwd	49.6	0.881788	14337.777778
sedan	4wd	54.3	0.848630	12647.333333
	fwd	50.6	0.925997	9811.800000
	rwd	47.8	1.000000	21711.833333
wagon	4wd	54.9	0.834214	9095.750000
	fwd	53.0	0.925997	9997.333333
	rwd	54.1	0.955790	16994.222222

We can use numpy functions in aggregation as well:

```
# import the numpy library as np
import numpy as np
# using numpy libraries for operations
df.groupby(["body-style","drive-wheels"])["price"].agg([np.sum, np.mean, np.std])
```

The **output** of the preceding code is as follows:

body-style	drive-wheels	sum	mean	std
convertible	fwd	11595.0	11595.000000	NaN
	rwd	119748.0	23949.600000	11165.099700
hardtop	fwd	8249.0	8249.000000	NaN
	rwd	169419.0	24202.714286	14493.311190
hatchback	4wd	7603.0	7603.000000	NaN
	fwd	411423.0	8396.387755	3004.675695
	rwd	258080.0	14337.777778	3831.795195
sedan	4wd	37942.0	12647.333333	4280.814681
	fwd	539649.0	9811.800000	3519.517598
	rwd	781626.0	21711.833333	9194.820239
wagon	4wd	36383.0	9095.750000	1775.652063
	fwd	119968.0	9997.333333	3584.185551
	rwd	152948.0	16994.222222	4686.703313

Renaming grouped aggregation columns

- We can perform aggregation in each group and rename the columns according to the operation performed.

- This is useful for understanding the output dataset:

```
df.groupby(["body-style","drive-wheels"]).agg(
    # Get max of the price column for each group
    max_price=('price', max),
    # Get min of the price column for each group
    min_price=('price', min),
    # Get sum of the price column for each group
    total_price=('price', 'mean') )
```

The **output** of the preceding code is as follows:

body-style	drive-wheels	max_price	min_price	total_price
convertible	fwd	11595.0	11595.0	11595.000000
	rwd	37028.0	13495.0	23949.600000
hardtop	fwd	8249.0	8249.0	8249.000000
	rwd	45400.0	8449.0	24202.714286
hatchback	4wd	7603.0	7603.0	7603.000000
	fwd	18150.0	5118.0	8396.387755
	rwd	22018.0	8238.0	14337.777778
sedan	4wd	17450.0	9233.0	12647.333333
	fwd	23875.0	5499.0	9811.800000
	rwd	41315.0	6785.0	21711.833333
wagon	4wd	11694.0	7898.0	9095.750000
	fwd	18920.0	6918.0	9997.333333
	rwd	28248.0	12440.0	16994.222222

- As shown in the preceding screenshot, we only selected two categories: body-style and drive-wheels. For each row in these categories, the maximum price, the minimum price, and the total price is computed in the successive columns.

PIVOT TABLES AND CROSS-TABULATIONS TECHNIQUES

- Pandas offer several options for grouping and summarizing data.
- Like groupby, aggregation, and transformation, there are other options available, such as pivot_table and crosstab.

Pivot tables

- The pandas.pivot_table() function creates a spreadsheet-style pivot table as a dataframe.
- The levels in the pivot table will be stored in MultiIndex objects (hierarchical indexes) on the index and columns of the resulting dataframe.
- The simplest pivot tables must have a dataframe and an index/listof the index.

1. Creating a pivot table of a new dataframe that consists of the body- style, drive-wheels, length, width, height, curb-weight, and price columns:

new_dataset1=df.filter(["body-style","drive wheels","length","width","height","curbweight","price"],axis=1)

#simplest pivot table with dataframe df and index body-styletable = pd.pivot_table(new_dataset1, index =["body-style"]) table

Output:

body-style	curb-weight	height	length	price	width
convertible	2801.666667	51.433333	0.818757	24079.550000	0.910880
hardtop	2810.625000	52.850000	0.850252	24429.350000	0.925174
hatchback	2322.852941	52.133824	0.799078	10953.185294	0.904228
sedan	2625.893617	54.387234	0.855583	15905.730851	0.921070
wagon	2784.240000	56.728000	0.871235	13609.156000	0.920222

- The output table is similar to grouping a dataframe with respect to body-style.

2. The values in the preceding table are the mean of the values in thecorresponding category.Design a pivot table with the new_dataset1 dataframe and make body-style and drive-wheels as an index.
   - Providing multiple indexes will make a grouping of the dataframe first and then summarize the data

#pivot table with dataframe df and index body-style and drivewheels table = pd.pivot_table(new_dataset1, index =["body-style","drivewheels"])table

Output:

body-style	drive-wheels	curb-weight	height	length	price	width
convertible	fwd	2254.000000	55.600000	0.765497	12754.500000	0.891667
	rwd	2911.200000	50.600000	0.829409	26344.560000	0.914722
hardtop	fwd	2008.000000	53.300000	0.780394	9073.900000	0.886111
	rwd	2925.285714	52.785714	0.860232	26622.985714	0.930754
hatchback	4wd	2240.000000	55.700000	0.755887	8363.300000	0.886111
	fwd	2181.551020	52.442857	0.787818	9236.026531	0.898214
	rwd	2712.111111	51.094444	0.832132	15771.555556	0.921605
sedan	4wd	2573.000000	54.300000	0.833894	13912.066667	0.912963
	fwd	2313.018182	53.956364	0.828404	10792.980000	0.908182
	rwd	3108.305556	55.052778	0.898913	23883.016667	0.941435
wagon	4wd	2617.500000	57.000000	0.824844	10005.325000	0.895833
	fwd	2464.333333	56.008333	0.843064	10997.066667	0.910185
	rwd	3284.888889	57.566667	0.929414	18693.644444	0.944444

- The output is a pivot table grouped by body-style and drive- wheels.
- It contains the average of the numerical values of the corresponding columns.
- The syntax for the pivot table takes some arguments, such as c, values, index, column, and aggregation function.
- We can apply the aggregation function to a pivot table at the same time.
- We can pass the aggregation function, values, and columns that aggregation will be applied to, in order to create a pivot table of a summarized subset of a dataframe:

```
import numpy as np
new_dataset3 = df.filter(["body-style","drivewheels","price"],axis=1)
table = pd.pivot_table(new_dataset3, values='price', index=["bodystyle"],
columns=["drivewheels"],
aggfunc=np.mean,fill_value=0)table
```

In terms of syntax, the preceding code represents the following:

- A pivot table with a dataset called new_dataset3.
- The values are the columns that the aggregation function is to beapplied to.
- The index is a column for grouping data.
- Columns for specifying the category of data.
- aggfunc is the aggregation function to be applied.
- fill_value is used to fill in missing values.

Output:

drive-wheels	4wd	fwd	rwd
body-style			
convertible	0.000000	12754.500000	26344.560000
hardtop	0.000000	9073.900000	26622.985714
hatchback	8363.300000	9236.026531	15771.555556
sedan	13912.066667	10792.980000	23883.016667
wagon	10005.325000	10997.066667	18693.644444

- The preceding pivot table represents the average price of cars with different body-style and available drive-wheels in those body-style.

3. A different aggregation function can also be applied to differentcolumns:

```
table = pd.pivot_table(new_dataset1,values=['price','height','width'],
index =["body-style","drive-wheels"], aggfunc={'price': np.mean,'height':
[min,max],'width': [min, max]}, fill_value=0)
table
```

Output:

		height		price	width	
		max	min	mean	max	min
body-style	drive-wheels					
convertible	fwd	55.6	55.6	12754.500000	0.891667	0.891667
	rwd	53.0	48.8	26344.560000	0.979167	0.890278
hardtop	fwd	53.3	53.3	9073.900000	0.886111	0.886111
	rwd	55.4	51.6	26622.985714	1.000000	0.902778
hatchback	4wd	55.7	55.7	8363.300000	0.886111	0.886111
	fwd	56.1	49.4	9236.026531	0.925000	0.837500
	rwd	54.8	49.6	15771.555556	0.948611	0.888889
sedan	4wd	54.3	54.3	13912.066667	0.922222	0.908333
	fwd	56.1	50.6	10792.980000	0.991667	0.868056
	rwd	56.7	47.8	23883.016667	0.995833	0.858333
wagon	4wd	59.1	54.9	10005.325000	0.908333	0.883333
	fwd	59.8	53.0	10997.066667	0.991667	0.883333
	rwd	58.7	54.1	18693.644444	0.976389	0.923611

- This pivot table represents the maximum and minimum of the height and width and the average price of cars in the respective categories mentioned in the index.

Cross-tabulations

- The pandas dataframe can be customized with another technique called cross-tabulation.
- This allows us to cope with groupby and aggregation for better data analysis.
- Pandas have the crosstab function, which helps to build a cross-tabulation table.
- The cross-tabulation table shows the frequency with which certain groups of data appear.
- To identify how many different body styles cars are made by different makers, use pd.crosstab()

pd.crosstab(df[“make”], df[“body-style”])

Output

body-style make	convertible	hardtop	hatchback	sedan	wagon
alfa-romero	2	0	1	0	0
audi	0	0	0	5	1
bmw	0	0	0	8	0
chevrolet	0	0	2	1	0
dodge	0	0	5	3	1
honda	0	0	7	5	1
isuzu	0	0	1	1	0
jaguar	0	0	0	3	0
mazda	0	0	10	7	0
mercedes-benz	1	2	0	4	1
mercury	0	0	1	0	0

Apply margins and margins_name attribute to display the row wise andcolumn wise sum of the cross table

pd.crosstab(df["make"], df["bodystyle"], margins=True,margins_name="Total Made")

Output:

body-style make	convertible	hardtop	hatchback	sedan	wagon	Total Made
alfa-romero	2	0	1	0	0	3
audi	0	0	0	5	1	6
bmw	0	0	0	8	0	8
chevrolet	0	0	2	1	0	3
dodge	0	0	5	3	1	9
honda	0	0	7	5	1	13
isuzu	0	0	1	1	0	2
jaguar	0	0	0	3	0	3
mazda	0	0	10	7	0	17
mercedes-benz	1	2	0	4	1	8
mercury	0	0	1	0	0	1
mitsubishi	0	0	9	4	0	13
nissan	0	1	5	9	3	18
peugot	0	0	0	7	4	11

- Applying multiple columns in the crosstab function for the row index or column index or both will print the output with grouping automatically.

2. Data distribution by the body-type and drive_wheels columns withinthe maker of car and their door type in a crosstab:

pd.crosstab([df["make"],df["num-of-doors"]], [df["bodystyle"], df["drive-wheels"]], margins=True,margins_name="Total Made")

Output:

body-style		convertible		hardtop		hatchback			sedan			wagon			Total Made
drive-wheels		fwd	rwd	fwd	rwd	4wd	fwd	rwd	4wd	fwd	rwd	4wd	fwd	rwd	
make	num-of-doors														
alfa-romero	two	0	2	0	0	0	0	1	0	0	0	0	0	0	3
audi	four	0	0	0	0	0	0	0	1	3	0	0	1	0	5
	two	0	0	0	0	0	0	0	0	1	0	0	0	0	1
bmw	four	0	0	0	0	0	0	0	0	0	5	0	0	0	5
	two	0	0	0	0	0	0	0	0	0	3	0	0	0	3
chevrolet	four	0	0	0	0	0	0	0	0	1	0	0	0	0	1
	two	0	0	0	0	0	2	0	0	0	0	0	0	0	2
dodge	four	0	0	0	0	0	1	0	0	3	0	0	1	0	5
	two	0	0	0	0	0	4	0	0	0	0	0	0	0	4
honda	four	0	0	0	0	0	0	0	0	4	0	0	1	0	5
	two	0	0	0	0	0	7	0	0	1	0	0	0	0	8
isuzu	four	0	0	0	0	0	0	0	0	0	1	0	0	0	1

#

Rename the columns and row index for better understanding ofcrosstab

pd.crosstab([df["make"],df["num-of-doors"]], [df["bodystyle"],df["drive-wheels"]], rownames=['Auto Manufacturer', "Doors"], colnames=['Body Style', "Drive Type"], margins=True,margins_name="Total Made").head()

Output:

Body Style		convertible		hardtop		hatchback			sedan			wagon			Total Made
Drive Type		fwd	rwd	fwd	rwd	4wd	fwd	rwd	4wd	fwd	rwd	4wd	fwd	rwd	
Auto Manufacturer	Doors														
alfa-romero	two	0	2	0	0	0	0	1	0	0	0	0	0	0	3
audi	four	0	0	0	0	0	0	0	1	3	0	0	1	0	5
	two	0	0	0	0	0	0	0	0	1	0	0	0	0	1
bmw	four	0	0	0	0	0	0	0	0	0	5	0	0	0	5
	two	0	0	0	0	0	0	0	0	0	3	0	0	0	3

- The pivot table syntax of pd.crosstab also takes some arguments, such as dataframe columns, values, normalize, and the aggregation function.
- The aggregation function can be applied to a cross table at the same time.
- Passing the aggregation function and values, which are the columns that aggregation will be applied to, gives a cross table ofa summarized subset of the dataframe.

3. The average curb-weight of cars made by different makers with respect to their body-style by applying the mean() aggregation function to the crosstable:

values are the column in which aggregation function is to beapplied
aggfunc is the aggregation function to be appliedround() to round output

```
pd.crosstab(df["make"], df["body-style"],values=df["curb-weight"],
aggfunc='mean').round(0)
```

Output:

body-style make	convertible	hardtop	hatchback	sedan	wagon
alfa-romero	2548.0	NaN	2823.0	NaN	NaN
audi	NaN	NaN	NaN	2720.0	2954.0
bmw	NaN	NaN	NaN	2929.0	NaN
chevrolet	NaN	NaN	1681.0	1909.0	NaN
dodge	NaN	NaN	2132.0	2056.0	2535.0
honda	NaN	NaN	1970.0	2289.0	2024.0
isuzu	NaN	NaN	2734.0	2337.0	NaN
jaguar	NaN	NaN	NaN	4027.0	NaN
mazda	NaN	NaN	2254.0	2361.0	NaN
mercedes-benz	3685.0	3605.0	NaN	3731.0	3750.0
mercury	NaN	NaN	2910.0	NaN	NaN
mitsubishi	NaN	NaN	2377.0	2394.0	NaN
nissan	NaN	2008.0	2740.0	2238.0	2452.0
peugot	NaN	NaN	NaN	3143.0	3358.0

- A normalized crosstab will show the percentage of time eachcombination occurs.
- This can be accomplished using the normalize parameter, asfollows:

- Cross-tabulation techniques is useful to analyze two or morevariables.
- This helps in inspecting the relationships between them.

```
pd.crosstab(df["make"], df["body-style"],normalize=True).head(10)
```

Output:

body-style	convertible	hardtop	hatchback	sedan	wagon
make					
alfa-romero	0.009950	0.00000	0.004975	0.000000	0.000000
audi	0.000000	0.00000	0.000000	0.024876	0.004975
bmw	0.000000	0.00000	0.000000	0.039801	0.000000
chevrolet	0.000000	0.00000	0.009950	0.004975	0.000000
dodge	0.000000	0.00000	0.024876	0.014925	0.004975
honda	0.000000	0.00000	0.034826	0.024876	0.004975
isuzu	0.000000	0.00000	0.004975	0.004975	0.000000
jaguar	0.000000	0.00000	0.000000	0.014925	0.000000
mazda	0.000000	0.00000	0.049751	0.034826	0.000000
mercedes-benz	0.004975	0.00995	0.000000	0.019900	0.004975