Handling Data with pandas DataFrame

The **pandas** library is an extremely resourceful open source toolkit for handling, manipulating, and analyzing structured data. Data tables can be stored in the DataFrame object available in **pandas**, and data in multiple formats (for example, **.csv**, **.tsv**, **.xlsx**, and **.json**) can be read directly into a DataFrame. Utilizing built-in functions, DataFrames can be efficiently manipulated (for example, converting tables between different views, such as, long/wide; grouping by a specific column/feature; summarizing data; and more).

Reading Data from Files

Most small-to medium-sized datasets are usually available or shared as delimited files such as **comma-separated values** (**CSV**), **tab-separated values (TSV**), **Excel** (**.xslx**), and JSON files. Pandas provides built-in I/O functions to read files in several formats, such as, **read_csv**, **read_excel**, and **read_json**, and so on into a DataFrame. In this section, we will use the **diamonds** dataset (available on Moodle).

Exercise 1: Reading Data from Files

In this exercise, we will read from a dataset. The example here uses the **diamonds** dataset:

1. Open a jupyter notebook and load the **pandas** and **seaborn** libraries:

   #Load pandas library

   import pandas as pd

   import seaborn as sns

2. Specify the URL of the dataset:

   #URL of the dataset

   diamonds_url = "https://raw.githubusercontent.com/TrainingByPackt/Interactive-Data-Visualization-with-Python/master/datasets/diamonds.csv"

3. Read files from the URL into the **pandas** DataFrame:

   #Yes, we can read files from a URL straight into a pandas DataFrame!

   diamonds_df = pd.read_csv(diamonds_url)

   # Since the dataset is available in seaborn, we can alternatively read it in using the following line of code

   diamonds_df = sns.load_dataset('diamonds')

The dataset is read directly from the URL!

Note

Use the **usecols** parameter if only specific columns need to be read.

The syntax can be followed for other datatypes using, as shown here:

```
diamonds_df_specific_cols = pd.read_csv(diamonds_url,usecols=['carat','cut','color','clarity'])
```

Observing and Describing Data

Now that we know how to read from a dataset, let's go ahead with observing and describing data from a dataset. **pandas** also offers a way to view the first few rows in a DataFrame using the **head()** function. By default, it shows **5** rows. To adjust that, we can use the argument **n**—for instance, **head(n=5)**.

Exercise 2: Observing and Describing Data

Here we see how to observe and describe data in a DataFrame. We'll be again using the **diamonds** dataset:

1. Load the **pandas** and **seaborn** libraries:

   ```
   #Load pandas library

   import pandas as pd

   import seaborn as sns
   ```

2. Specify the URL of the dataset:

   ```
   #URL of the dataset

   diamonds_url = "https://raw.githubusercontent.com/TrainingByPackt/Interactive-Data-Visualization-with-Python/master/datasets/diamonds.csv"
   ```

3. Read files from the URL into the **pandas** DataFrame:

   ```
   #Yes, we can read files from a URL straight into a pandas DataFrame!

   diamonds_df = pd.read_csv(diamonds_url)

   # Since the dataset is available in seaborn, we can alternatively read it in using the following line of code

   diamonds_df = sns.load_dataset('diamonds')
   ```

4. Observe the data by using the **head** function:

```
diamonds_df.head()
```

The output is as follows:

	carat	cut	color	clarity	depth	table	price	x	y	z
0	0.23	Ideal	E	SI2	61.5	55.0	326	3.95	3.98	2.43
1	0.21	Premium	E	SI1	59.8	61.0	326	3.89	3.84	2.31
2	0.23	Good	E	VS1	56.9	65.0	327	4.05	4.07	2.31
3	0.29	Premium	I	VS2	62.4	58.0	334	4.20	4.23	2.63
4	0.31	Good	J	SI2	63.3	58.0	335	4.34	4.35	2.75

Figure 1.1: Displaying the diamonds dataset

The data contains different features of diamonds, such as **carat**, **cut quality**, **color**, and **price**, as columns. Now, **cut**, **clarity**, and **color** are **categorical variables**, and **x**, **y**, **z**, **depth**, **table**, and **price** are **continuous variables**. While categorical variables take unique categories/names as values, continuous values take real numbers as values.

cut, **color**, and **clarity** are ordinal variables with **5**, **7**, and **8** unique values (can be obtained by **diamonds_df.cut.nunique()**, **diamonds_df.color.nunique()**, **diamonds_df.clarity.nunique()** – try it!), respectively. **cut** is the quality of the cut, described as **Fair**, **Good**, **Very Good**, **Premium**, or **Ideal**; **color** describes the diamond color from **J (worst)** to **D (best)**. There's also **clarity**, which measures how clear the diamond is—the degrees are **I1 (worst)**, **SI1**, **SI2**, **VS1**, **VS2**, **VVS1**, **VVS2**, and **IF (best)**.

5. Count the number of rows and columns in the DataFrame using the **shape** function:

   ```
   diamonds_df.shape
   ```

   The output is as follows:

   ```
   (53940, 10)
   ```

   The first number, **53940**, denotes the number of rows and the second, **10**, denotes the number of columns.

6. Summarize the columns using **describe()** to obtain the distribution of variables, including **mean**, **median**, **min**, **max**, and the different quartiles:

   ```
   diamonds_df.describe()
   ```

The output is as follows:

	carat	depth	table	price	x	y	z
count	53940.000000	53940.000000	53940.000000	53940.000000	53940.000000	53940.000000	53940.000000
mean	0.797940	61.749405	57.457184	3932.799722	5.731157	5.734526	3.538734
std	0.474011	1.432621	2.234491	3989.439738	1.121761	1.142135	0.705699
min	0.200000	43.000000	43.000000	326.000000	0.000000	0.000000	0.000000
25%	0.400000	61.000000	56.000000	950.000000	4.710000	4.720000	2.910000
50%	0.700000	61.800000	57.000000	2401.000000	5.700000	5.710000	3.530000
75%	1.040000	62.500000	59.000000	5324.250000	6.540000	6.540000	4.040000
max	5.010000	79.000000	95.000000	18823.000000	10.740000	58.900000	31.800000

Figure 1.2: Using the describe function to show continuous variables

This works for continuous variables. However, for categorical variables, we need to use the **include=object** parameter.

7. Use **include=object** inside the **describe** function for categorical variables (**cut**, **color**, **clarity**):

   diamonds_df.describe(include=object)

   The output is as follows:

	cut	color	clarity
count	53940	53940	53940
unique	5	7	8
top	Ideal	G	SI1
freq	21551	11292	13065

Figure 1.3: Use the describe function to show categorical variables

Now, what if you would want to see the column types and how much memory a DataFrame occupies?

8. To obtain information on the dataset, use the **info()** method:

   diamonds_df.info()

   The output is as follows:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 53940 entries, 0 to 53939
Data columns (total 10 columns):
carat      53940 non-null float64
cut        53940 non-null object
color      53940 non-null object
clarity    53940 non-null object
depth      53940 non-null float64
table      53940 non-null float64
price      53940 non-null int64
x          53940 non-null float64
y          53940 non-null float64
z          53940 non-null float64
dtypes: float64(6), int64(1), object(3)
memory usage: 4.1+ MB
```

Figure 1.4: Information on the diamonds dataset

The preceding figure shows the data type (**float64**, **object**, **int64**..) of each of the columns, and memory (**4.1MB**) that the DataFrame occupies. It also tells the number of rows (**53940**) present in the DataFrame.

Selecting Columns from a DataFrame

Let's see how to select specific columns from a dataset. A column in a **pandas** DataFrame can be accessed in two simple ways: with the **.** operator or the **[ ]** operator. For example, we can access the **cut** column of the **diamonds_df** DataFrame with **diamonds_df.cut** or **diamonds_df['cut']**. However, there are some scenarios where the **.** operator cannot be used:

- When the column name contains spaces
- When the column name is an integer
- When creating a new column

Now, how about selecting all rows corresponding to diamonds that have the **Ideal** cut and storing them in a separate DataFrame? We can select them using the **loc** functionality:

diamonds_low_df = diamonds_df.loc[diamonds_df['cut']=='Ideal']

diamonds_low_df.head()

The output is as follows:

	carat	cut	color	clarity	depth	table	price	x	y	z
0	0.23	Ideal	E	SI2	61.5	55.0	326	3.95	3.98	2.43
11	0.23	Ideal	J	VS1	62.8	56.0	340	3.93	3.90	2.46
13	0.31	Ideal	J	SI2	62.2	54.0	344	4.35	4.37	2.71
16	0.30	Ideal	I	SI2	62.0	54.0	348	4.31	4.34	2.68
39	0.33	Ideal	I	SI2	61.8	55.0	403	4.49	4.51	2.78

Figure 1.5: Selecting specific columns from a DataFrame

Here, we obtain indices of rows that meet the criterion:

[diamonds_df['cut']=='Ideal' and then select them using **loc**.

Adding New Columns to a DataFrame

Now, we'll see how to add new columns to a DataFrame. We can add a column, such as, **price_per_carat**, in the **diamonds** DataFrame. We can divide the values of two columns and populate the data fields of the newly added column.

Exercise 3: Adding New Columns to the DataFrame

We are going to add new columns to the **diamonds** dataset in the **pandas** library. We'll start with the simple addition of columns and then move ahead and look into the conditional addition of columns. To do so, let's go through the following steps:

1. Load the **pandas** and **seaborn** libraries:

   ```
   #Load pandas library

   import pandas as pd

   import seaborn as sns
   ```

2. Specify the URL of the dataset:

   ```
   #URL of the dataset

   diamonds_url = "https://raw.githubusercontent.com/TrainingByPackt/Interactive-Data-Visualization-with-Python/master/datasets/diamonds.csv"
   ```

3. Read files from the URL into the **pandas** DataFrame:

   ```
   #Yes, we can read files from a URL straight into a pandas DataFrame!

   diamonds_df = pd.read_csv(diamonds_url)

   # Since the dataset is available in seaborn, we can alternatively read it in using the following line of code

   diamonds_df = sns.load_dataset('diamonds')
   ```

   Let's look at simple addition of columns.

4. Add a **price_per_carat** column to the DataFrame:

   ```
   diamonds_df['price_per_carat'] = diamonds_df['price']/diamonds_df['carat']
   ```

5. Call the DataFrame **head** function to check whether the new column was added as expected:

   diamonds_df.head()

   The output is as follows:

	carat	cut	color	clarity	depth	table	price	x	y	z	price_per_carat
0	0.23	Ideal	E	SI2	61.5	55.0	326	3.95	3.98	2.43	1417.391304
1	0.21	Premium	E	SI1	59.8	61.0	326	3.89	3.84	2.31	1552.380952
2	0.23	Good	E	VS1	56.9	65.0	327	4.05	4.07	2.31	1421.739130
3	0.29	Premium	I	VS2	62.4	58.0	334	4.20	4.23	2.63	1151.724138
4	0.31	Good	J	SI2	63.3	58.0	335	4.34	4.35	2.75	1080.645161

Figure 1.6: Simple addition of columns

Similarly, we can also use addition, subtraction, and other mathematical operators on two numeric columns.

Now, we'll look at *conditional addition of columns*. Let's try and add a column based on the value in **price_per_carat**, say anything more than **3500** as high (coded as **1**) and anything less than **3500** as low (coded as **0**).

6. Use the **np.where** function from Python's **numpy** package:

   #Import numpy package for linear algebra

   import numpy as np

   diamonds_df['price_per_carat_is_high'] = np.where(diamonds_df['price_per_carat']>3500,1,0)

   diamonds_df.head()

   The output is as follows:

	carat	cut	color	clarity	depth	table	price	x	y	z	price_per_carat	price_per_carat_is_high
0	0.23	Ideal	E	SI2	61.5	55.0	326	3.95	3.98	2.43	1417.391304	0
1	0.21	Premium	E	SI1	59.8	61.0	326	3.89	3.84	2.31	1552.380952	0
2	0.23	Good	E	VS1	56.9	65.0	327	4.05	4.07	2.31	1421.739130	0
3	0.29	Premium	I	VS2	62.4	58.0	334	4.20	4.23	2.63	1151.724138	0
4	0.31	Good	J	SI2	63.3	58.0	335	4.34	4.35	2.75	1080.645161	0

Figure 1.7: Conditional addition of columns

Therefore, we have successfully added two new columns to the dataset.

Applying Functions on DataFrame Columns

You can apply *simple functions* on a DataFrame column—such as, addition, subtraction, multiplication, division, squaring, raising to an exponent, and so on. It is also possible to apply more *complex functions* on single and multiple columns in a **pandas** DataFrame. As an example, let's say we want to round off the price of diamonds to its ceil (nearest integer equal to or higher than the actual price). Let's explore this through an exercise.

Exercise 4: Applying Functions on DataFrame columns

In this exercise, we'll consider a scenario where the price of diamonds has increased, and we want to apply an increment factor of **1.3** to the price of all the diamonds in our record. We can achieve this by applying a simple function. Next, we'll round off the price of diamonds to its ceil. We'll achieve that by applying a complex function. Let's go through the following steps:

1. Load the **pandas** and **seaborn** libraries:

   #Load pandas library

   import pandas as pd

   import seaborn as sns

2. Specify the URL of the dataset:

   #URL of the dataset

   diamonds_url = "https://raw.githubusercontent.com/TrainingByPackt/Interactive-Data-Visualization-with-Python/master/datasets/diamonds.csv"

3. Read files from the URL into the **pandas** DataFrame:

   #Yes, we can read files from a URL straight into a pandas DataFrame!

   diamonds_df = pd.read_csv(diamonds_url)

   # Since the dataset is available in seaborn, we can alternatively read it in using the following line of code

   diamonds_df = sns.load_dataset('diamonds')

4. Add a **price_per_carat** column to the DataFrame:

   diamonds_df['price_per_carat'] = diamonds_df['price']/diamonds_df['carat']

5. Use the **np.where** function from Python's **numpy** package:

```
#Import numpy package for linear algebra

import numpy as np

diamonds_df['price_per_carat_is_high'] =
np.where(diamonds_df['price_per_carat']>3500,1,0)
```

6. Apply a simple function on the columns using the following code:

```
diamonds_df['price']= diamonds_df['price']*1.3
```

7. Apply a complex function to round off the price of diamonds to its ceil:

```
import math

diamonds_df['rounded_price']=diamonds_df['price'].apply(math.ceil)

diamonds_df.head()
```

The output is as follows:

	carat	cut	color	clarity	depth	table	price	x	y	z	price_per_carat	price_per_carat_is_high	rounded_price
0	0.23	Ideal	E	SI2	61.5	55.0	423.8	3.95	3.98	2.43	1417.391304	0	424
1	0.21	Premium	E	SI1	59.8	61.0	423.8	3.89	3.84	2.31	1552.380952	0	424
2	0.23	Good	E	VS1	56.9	65.0	425.1	4.05	4.07	2.31	1421.739130	0	426
3	0.29	Premium	I	VS2	62.4	58.0	434.2	4.20	4.23	2.63	1151.724138	0	435
4	0.31	Good	J	SI2	63.3	58.0	435.5	4.34	4.35	2.75	1080.645161	0	436

Figure 1.8: Dataset after applying simple and complex functions

In this case, the function we wanted for rounding off to the ceil was already present in an existing library. However, there might be times when you have to write your own function to perform the task you want to accomplish. In the case of small functions, you can also use the **lambda** operator, which acts as a one-liner function taking an argument. For example, say you want to add another column to the DataFrame indicating the rounded-off price of the diamonds to the nearest multiple of **100** (equal to or higher than the price).

8. Use the **lambda** function as follows to round off the price of the diamonds to the nearest multiple of **100**:

```
import math

diamonds_df['rounded_price_to_100multiple']=diamonds_df['price'].apply(lambda x:
math.ceil(x/100)*100)

diamonds_df.head()
```

The output is as follows:

	carat	cut	color	clarity	depth	table	price	x	y	z	price_per_carat	price_per_carat_is_high	rounded_price	rounded_price_to_100multiple
0	0.23	Ideal	E	SI2	61.5	55.0	423.8	3.95	3.98	2.43	1417.391304	0	424	500
1	0.21	Premium	E	SI1	59.8	61.0	423.8	3.89	3.84	2.31	1552.380952	0	424	500
2	0.23	Good	E	VS1	56.9	65.0	425.1	4.05	4.07	2.31	1421.739130	0	426	500
3	0.29	Premium	I	VS2	62.4	58.0	434.2	4.20	4.23	2.63	1151.724138	0	435	500
4	0.31	Good	J	SI2	63.3	58.0	435.5	4.34	4.35	2.75	1080.645161	0	436	500

Figure 1.9: Dataset after applying the lambda function

Not all functions can be written as one-liners and it is important to know how to include user-defined functions in the **apply** function. Let's write the same code with a *user-defined function* for illustration.

9. Write code to create a user-defined function to round off the price of the diamonds to the nearest multiple of **100**:

```
import math

def get_100_multiple_ceil(x):

    y = math.ceil(x/100)*100

    return y

diamonds_df['rounded_price_to_100multiple']=diamonds_df['price'].apply(get_100_
multiple_ceil)

diamonds_df.head()
```

The output is as follows:

	carat	cut	color	clarity	depth	table	price	x	y	z	price_per_carat	price_per_carat_is_high	rounded_price	rounded_price_to_100multiple
0	0.23	Ideal	E	SI2	61.5	55.0	423.8	3.95	3.98	2.43	1417.391304	0	424	500
1	0.21	Premium	E	SI1	59.8	61.0	423.8	3.89	3.84	2.31	1552.380952	0	424	500
2	0.23	Good	E	VS1	56.9	65.0	425.1	4.05	4.07	2.31	1421.739130	0	426	500
3	0.29	Premium	I	VS2	62.4	58.0	434.2	4.20	4.23	2.63	1151.724138	0	435	500
4	0.31	Good	J	SI2	63.3	58.0	435.5	4.34	4.35	2.75	1080.645161	0	436	500

Figure 1.10: Dataset after applying a user-defined function

Interesting! Now, we have created a user-defined function to add a column to the dataset.

Exercise 5: Applying Functions on Multiple Columns

When applying a function on multiple columns of a DataFrame, we can similarly use **lambda** or user-defined functions. We will continue to use the **diamonds** dataset. Suppose we are interested in buying diamonds that have an **Ideal** cut and a **color** of **D** (entirely colorless). This exercise is for adding a new column, **desired** to the DataFrame, whose value will be **yes** if our criteria are satisfied and **no** if not satisfied.

1. Import the necessary modules:

    import seaborn as sns

    import pandas as pd

2. Import the **diamonds** dataset from **seaborn**:

    diamonds_df_exercise = sns.load_dataset('diamonds')

3. Write a function to determine whether a record, **x**, is desired or not:

    def is_desired(x):

      bool_var = 'yes' if (x['cut']=='Ideal' and x['color']=='D') else 'no'

      return bool_var

4. Use the **apply** function to add the new column, **desired**:

    diamonds_df_exercise['desired']=diamonds_df_exercise.apply(is_desired, axis=1)

    diamonds_df_exercise.head()

    The output is as follows:

	carat	cut	color	clarity	depth	table	price	x	y	z	desired
0	0.23	Ideal	E	SI2	61.5	55.0	326	3.95	3.98	2.43	no
1	0.21	Premium	E	SI1	59.8	61.0	326	3.89	3.84	2.31	no
2	0.23	Good	E	VS1	56.9	65.0	327	4.05	4.07	2.31	no
3	0.29	Premium	I	VS2	62.4	58.0	334	4.20	4.23	2.63	no
4	0.31	Good	J	SI2	63.3	58.0	335	4.34	4.35	2.75	no

Figure 1.11: Dataset after applying the function on multiple columns

The new column **desired** is added!

Deleting Columns from a DataFrame

Finally, let's see how to delete columns from a **pandas** DataFrame. For example, we will delete the **rounded_price** and **rounded_price_to_100multiple** columns.

Exercise 6: Deleting Columns from a DataFrame

Here we will delete columns from a **pandas** DataFrame. We'll be using the **diamonds** dataset:

1. Import the necessary modules:

   ```
   import seaborn as sns

   import pandas as pd
   ```

2. Import the **diamonds** dataset from **seaborn**:

   ```
   diamonds_df = sns.load_dataset('diamonds')
   ```

3. Add a **price_per_carat** column to the DataFrame:

   ```
   diamonds_df['price_per_carat'] = diamonds_df['price']/diamonds_df['carat']
   ```

4. Use the **np.where** function from Python's **numpy** package:

   ```
   #Import numpy package for linear algebra

   import numpy as np

   diamonds_df['price_per_carat_is_high'] =
   np.where(diamonds_df['price_per_carat']>3500,1,0)
   ```

5. Apply a *complex function* to round off the price of diamonds to its ceil:

   ```
   import math

   diamonds_df['rounded_price']=diamonds_df['price'].apply(math.ceil)
   ```

6. Write a code to create a *user-defined function*:

   ```
   import math

   def get_100_multiple_ceil(x):

      y = math.ceil(x/100)*100

      return y

   diamonds_df['rounded_price_to_100multiple']=diamonds_df['price'].apply(get_100_
   multiple_ceil)
   ```

7. Delete the **rounded_price** and **rounded_price_to_100multiple** columns using the **drop** function:

```
diamonds_df=diamonds_df.drop(columns=['rounded_price', 'rounded_price_to_100multiple'])

diamonds_df.head()
```

The output is as follows:

	carat	cut	color	clarity	depth	table	price	x	y	z	price_per_carat	price_per_carat_is_high
0	0.23	Ideal	E	SI2	61.5	55.0	326	3.95	3.98	2.43	1417.391304	0
1	0.21	Premium	E	SI1	59.8	61.0	326	3.89	3.84	2.31	1552.380952	0
2	0.23	Good	E	VS1	56.9	65.0	327	4.05	4.07	2.31	1421.739130	0
3	0.29	Premium	I	VS2	62.4	58.0	334	4.20	4.23	2.63	1151.724138	0
4	0.31	Good	J	SI2	63.3	58.0	335	4.34	4.35	2.75	1080.645161	0

Figure 1.12: Dataset after deleting columns

Note

By default, when the **apply** or **drop** function is used on a **pandas** DataFrame, the original DataFrame is not modified. Rather, a copy of the DataFrame post modifications is returned by the functions. Therefore, you should assign the returned value back to the variable containing the DataFrame (for example, **diamonds_df=diamonds_df.drop(columns=['rounded_price', 'rounded_price_to_100multiple'])**).

In the case of the **drop** function, there is also a provision to avoid assignment by setting an **inplace=True** parameter, wherein the function performs the column deletion on the original DataFrame and does not return anything.

Writing a DataFrame to a File

The last thing to do is write a DataFrame to a file. We will be using the **to_csv()** function. The output is usually a **.csv** file that will include column and row headers. Let's see how to write our DataFrame to a **.csv** file.

Exercise 7: Writing a DataFrame to a File

In this exercise, we will write a **diamonds** DataFrame to a **.csv** file. To do so, we'll be using the following code:

1. Import the necessary modules:

```
import seaborn as sns

import pandas as pd
```

2. Load the **diamonds** dataset from **seaborn**:

diamonds_df = sns.load_dataset('diamonds')

3. Write the diamonds dataset into a .csv file:

   diamonds_df.to_csv('diamonds_modified.csv')

4. Let's look at the first few rows of the DataFrame:

   print(diamonds_df.head())

   The output is as follows:

```
   carat      cut color clarity  depth  table  price     x     y     z
0   0.23    Ideal     E     SI2   61.5   55.0    326  3.95  3.98  2.43
1   0.21  Premium     E     SI1   59.8   61.0    326  3.89  3.84  2.31
2   0.23     Good     E     VS1   56.9   65.0    327  4.05  4.07  2.31
3   0.29  Premium     I     VS2   62.4   58.0    334  4.20  4.23  2.63
4   0.31     Good     J     SI2   63.3   58.0    335  4.34  4.35  2.75
```

   Figure 1.13: The generated .csv file in the source folder

   By default, the **to_csv** function outputs a file that includes column headers as well as row numbers. Generally, the row numbers are not desirable, and an **index** parameter is used to exclude them:

5. Add a parameter **index=False** to exclude the row numbers:

   diamonds_df.to_csv('diamonds_modified.csv', index=False)

And that's it! You can find this **.csv** file on Moodle. You are now equipped to perform all the basic functions on **pandas** DataFrames required to get started with data visualization in Python.

In order to prepare the ground for using various visualization techniques, we went through the following aspects of handling **pandas** DataFrames:

- Reading data from files using the **read_csv( )**, **read_excel( )**, and **readjson( )** functions
- Observing and describing data using the **dataframe.head( )**, **dataframe.tail( )**, **dataframe.describe( )**, and **dataframe.info( )** functions
- Selecting columns using the **dataframe.column__name** or **dataframe['column__name']** notation
- Adding new columns using the **dataframe['newcolumnname']=...** notation
- Applying functions to existing columns using the **dataframe.apply(func)** function
- Deleting columns from DataFrames using the **_dataframe.drop(column_list)** function
- Writing DataFrames to files using the **_dataframe.tocsv()** function

These functions are useful for preparing data in a format suitable for input to visualization functions in Python libraries such as **seaborn**.