CHAPTER

2

Descriptive Analytics

LEARNING OBJECTIVES

After completing this chapter, you will be able to

- Understand the concept of descriptive analytics.
- Learn to load structured data onto DataFrame and perform exploratory data analysis.
- Learn data preparation activities such as filtering, grouping, ordering, and joining of datasets.
- Learn to handle data with missing values.
- Learn to prepare plots such as bar plot, histogram, distribution plot, box plot, scatter plot, pair plot, and heat maps to find insights.

2.1 | WORKING WITH DATAFRAMES IN PYTHON

Descriptive analytics is a component of analytics and is the science of describing the past data; it thus captures "what happened" in a given context. The primary objective of descriptive analytics is comprehension of data using data summarization, basic statistical measures and visualization. Data visualization is an integral component of business intelligence (BI). Visuals are used as part of dashboards that are generated by companies for understanding the performance of the company using various key performance indicators.

Data scientists deal with structured data in most of their data analysis activities and are very familiar with the concept of structured query language (SQL) table. SQL tables represent data in terms of rows and columns and make it convenient to explore and apply transformations. The similar structure of presenting data is supported in Python through DataFrames, which can be imagined as in-memory SQL tables, that is data in tabular format. DataFrames are widely used and very popular in the world of R and are inherited into Python by **Pandas** library.

A DataFrame is very efficient two-dimensional data structure as shown in Figure 2.1. It is flat in structure and is arranged in rows and columns. Rows and columns can be indexed or named.

Pandas library support methods to explore, analyze, and prepare data. It can be used for performing activities such as load, filter, sort, group, join datasets and also for dealing with missing data. To demonstrate usage of DataFrame, we will use IPL dataset (described in the next section) to load data into DataFrame and perform descriptive analytics.

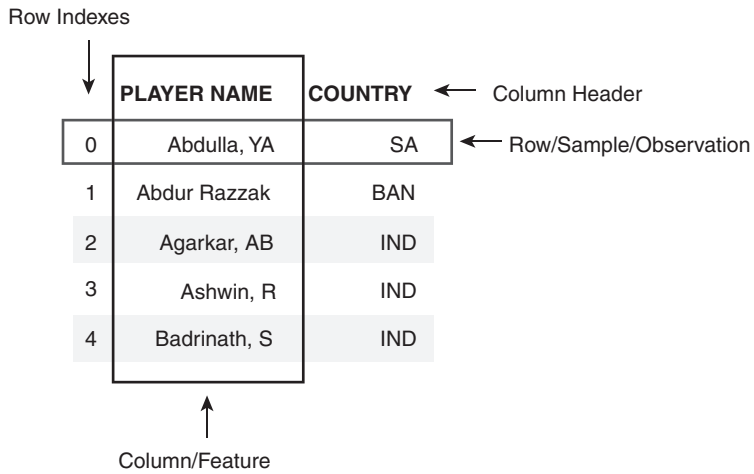

FIGURE 2.1 Structure of a DataFrame.

2.1.1 | IPL Dataset Description using DataFrame in Python

The Indian Premier League (IPL) is a professional league for Twenty20 (T20) cricket championships that was started in 2008 in India. It was initiated by the Board of Control for Cricket in India (BCCI) with eight franchises comprising players from across the world. The first IPL auction was held in 2008 for ownership of the teams for 10 years, with a base price of USD 50 million. The franchises acquire players through an English auction that is conducted every year. However, there are several rules imposed by the IPL. For example, there is a maximum cap on the money a franchise can spend on buying players.

The performance of the players could be measured through several metrics. Although the IPL follows the Twenty20 format of the game, it is possible that the performance of the players in the other formats of the game such as Test and One-Day matches can influence player pricing. A few players have excellent records in Test matches, but their records in Twenty20 matches are not very impressive. The dataset consists of the performance of 130 players measured through various performance metrics such as batting strike rate, economy rate and so on in the year 2013. The list of features is provided in Table 2.1.

TABLE 2.1 IPL auction price data description

Data Code	Data Type	Description
AGE	Categorical	Age of the player at the time of auction classified into 3 categories. Category 1 (L25) means the player is less than 25 years old, 2 means that age is between 25 and 35 years (B25–35) and category 3 means that the age is more than 35 (A35).
RUNS-S	Continuous	Number of runs scored by a player
RUNS-C	Continuous	Number of runs conceded by a player

(*Continued*)

TABLE 2.1 (Continued)

Data Code	Data Type	Description
HS	Continuous	Highest score by the batsman in IPL
AVE-B	Continuous	Average runs scored by the batsman in IPL
AVE-BL	Continuous	Bowling average (Number of runs conceded / number of wickets taken) in IPL
SR-B	Continuous	Batting strike rate (ratio of the number of runs scored to the number of balls faced) in IPL
SR-BL	Continuous	Bowling strike rate (ratio of the number of balls bowled to the number of wickets taken) in IPL
SIXERS	Continuous	Number of six runs scored by a player in IPL
WKTS	Continuous	Number of wickets taken by a player in IPL
ECON	Continuous	Economy rate of a bowler (number of runs conceded by the bowler per over) in IPL
CAPTAINCY EXP	Categorical	Captained either a T20 team or a national team
ODI-SR-B	Continuous	Batting strike rate in One-Day Internationals
ODI-SR-BL	Continuous	Bowling strike rate in One-Day Internationals
ODI-RUNS-S	Continuous	Runs scored in One-Day Internationals
ODI-WKTS	Continuous	Wickets taken in One-Day Internationals
T-RUNS-S	Continuous	Runs scored in Test maches
T-WKTS	Continuous	Wickets taken in Test maches
PLAYER-SKILL	Categorical	Player's primary skill (batsman, bowler, or all-rounder)
COUNTRY	Categorical	Country of origin of the player (AUS: Australia; IND: India; PAK: Pakistan; SA: South Africa; SL: Sri Lanka; NZ: New Zealand; WI: West Indies; OTH: Other countries)
YEAR-A	Categorical	Year of Auction in IPL
IPL TEAM	Categorical	Team(s) for which the player had played in the IPL (CSK: Chennai Super Kings, DC: Deccan Chargers, DD: Delhi Daredevils, KXI: Kings XI Punjab, KKR: Kolkata Knight Riders; MI: Mumbai Indians; PWI: Pune Warriors India; RR: Rajasthan Royals; RCB: Royal Challengers Bangalore). A + sign was used to indicate that the player had played for more than one team. For example, CSK+ would mean that the player had played tor CSK as well as for one or more other teams.

2.1.2 | Loading Dataset into Pandas DataFrame

Pandas is an open-source, Berkeley Software Distribution (BSD)-licensed library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language (source: *https://pandas.pydata.org/*). *Pandas* is very popular and most widely used library for data exploration and preparation. We will be using *Pandas* DataFrame throughout this book.

To use the Pandas library, we need to import *pandas* module into the environment using the *import* keyword. After importing, required *pandas* methods can be invoked using the format ***pandas.<method name>***.

In Python, *longer library* names can be avoided by assigning an alias name while importing. For example, in the code below, *pandas* is imported as alias *pd*. After the execution of this code, the required methods can be invoked using ***pd.<method name>***.

```
import pandas as pd
```

Pandas library has provided different methods for loading datasets with many different formats onto DataFrames. For example:

1. *read_csv* to read comma separated values.
2. *read_json* to read data with json format.
3. *read_fwf* to read data with fixed width format.
4. *read_excel* to read excel files.
5. *read_table* to read database tables.

To find out all available methods, we can type *pd.read_* and then press *TAB* in Jupyter notebook cell. It will list all available *read* methods. To find out what parameters a method takes and what data format it supports, we can type method name followed by a question mark (?) and press *SHIFT + ENTER*. For example, the command

```
pd.read_csv? (Press SHIFT + ENTER)
```

pops out detailed documentation at the bottom of the Jupyter notebook page as shown in Figure 2.2.

```
Signature: pd.read_csv(filepath_or_buffer, sep=',', delimiter=None, header='infer', names=None, index_col=None, usecols=None, squeeze=
False, prefix=None, mangle_dupe_cols=True, dtype=None, engine=None, converters=None, true_values=None, false_values=None, skipinitials
pace=False, skiprows=None, nrows=None, na_values=None, keep_default_na=True, na_filter=True, verbose=False, skip_blank_lines=True, par
se_dates=False, infer_datetime_format=False, keep_date_col=False, date_parser=None, dayfirst=False, iterator=False, chunksize=None, co
mpression='infer', thousands=None, decimal=b'.', lineterminator=None, quotechar='"', quoting=0, escapechar=None, comment=None, encodin
g=None, dialect=None, tupleize_cols=None, error_bad_lines=True, warn_bad_lines=True, skipfooter=0, skip_footer=0, doublequote=True, de
lim_whitespace=False, as_recarray=None, compact_ints=None, use_unsigned=None, low_memory=True, buffer_lines=None, memory_map=False, fl
oat_precision=None)
Docstring:
Read CSV (comma-separated) file into DataFrame
```

FIGURE 2.2 Sample documentation for Pandas method *read_csv*.

As per the documentation shown in Figure 2.2 for *pd.read_csv*

1. *read_csv* takes the file name as a parameter.
2. *read_csv* uses *comma* as separator. If any other separator, parameter *sep* to be set to appropriate character.
3. The first line in the dataset is expected to be header. If not, the *header* parameter needs to be set to *None*.

IPL dataset is stored in comma separated values (csv) format, so we will use *pd.read_csv* method to read and load it onto a DataFrame. The dataset contains header information in the first line.

```
ipl_auction_df = pd.read_csv('IPL IMB381IPL2013.csv')
```

To find out the type of variable *ipl_auction_df*, we can pass the variable to *type()* function of python.

```
type(ipl_auction_df)
```

```
pandas.core.frame.DataFrame
```

That is, *ipl_auction_df* is of type DataFrame. Now we can use pandas features such as selecting, filtering, aggregating, joining, slicing/dicing of data to prepare and explore the dataset.

2.1.3 | Displaying First Few Records of the DataFrame

To display the first few rows from the DataFrame, use function *head(n)*. The parameter *n* is the number of records to display.

In this example, we will only print a maximum of 7 columns as the total width exceeds the page width and the display is distorted. Setting pandas option *display.max_columns* will limit the number of columns to be printed/displayed.

```
pd.set_option('display.max_columns', 7)
```

```
ipl_auction_df.head(5)
```

	Sl. NO.	Player Name	Age	...	Auction Year	Base Price	Sold Price
0	1	Abdulla, YA	2	...	2009	50000	50000
1	2	Abdur Razzak	2	...	2008	50000	50000
2	3	Agarkar, AB	2	...	2008	200000	350000
3	4	Ashwin, R	1	...	2011	100000	850000
4	5	Badrinath, S	2	...	2011	100000	800000

The values in the first column – 0 to 4 – are row indexes and the words in first row are the dataset header.

2.1.4 | Finding Summary of the DataFrame

To display all the column names, use *columns* attribute of the DataFrame *ipl_auction_df*.

```
list(ipl_auction_df.columns)
```

```
['Sl.NO.',
 'PLAYER NAME',
 'AGE',
 'COUNTRY',
 'TEAM',
 'PLAYING ROLE',
 'T-RUNS',
 'T-WKTS',
 'ODI-RUNS-S',
 'ODI-SR-B',
 'ODI-WKTS',
 'ODI-SR-BL',
 'CAPTAINCY EXP',
```

```
'RUNS-S',
'HS',
'AVE',
'SR-B',
'SIXERS',
'RUNS-C',
'WKTS',
'AVE-BL',
'ECON',
'SR-BL',
'AUCTION YEAR',
'BASE PRICE',
'SOLD PRICE']
```

The other way to print a DataFrame with a large number of columns (features) is to transpose the DataFrame and display the columns as rows and rows as columns. The row indexes will be shown as column headers and column names will be shown as row indexes.

```
ipl_auction_df.head(5).transpose()
```

	0	1	2	3	4
SI. NO.	1	2	3	4	5
PLAYER NAME	Abdulla, YA	Abdur Razzak	Agarkar, AB	Ashwin, R	Badrinath, S
AGE	2	2	2	1	2
COUNTRY	SA	BAN	IND	IND	IND
TEAM	KXIP	RCB	KKR	CSK	CSK
PLAYING ROLE	Allrounder	Bowler	Bowler	Bowler	Batsman
T-RUNS	0	214	571	284	63
T-WKTS	0	18	58	31	0
ODI-RUNS-S	0	657	1269	241	79
ODI-SR-B	0	71.41	80.62	84.56	45.93
ODI-WKTS	0	185	288	51	0
ODI-SR-BL	0	37.6	32.9	36.8	0
CAPTAINCY EXP	0	0	0	0	0
RUNS-S	0	0	167	58	1317
HS	0	0	39	11	71
AVE	0	0	18.56	5.8	32.93
SR-B	0	0	121.01	76.32	120.71

(*Continued*)

	0	1	2	3	4
SIXERS	0	0	5	0	28
RUNS-C	307	29	1059	1125	0
WKTS	15	0	29	49	0
AVE-BL	20.47	0	36.52	22.96	0
ECON	8.9	14.5	8.81	6.23	0
SR-BL	13.93	0	24.9	22.14	0
AUCTION YEAR	2009	2008	2008	2011	2011
BASE PRICE	50000	50000	200000	100000	100000
SOLD PRICE	50000	50000	350000	850000	800000

The dimension or size of the DataFrame can be retrieved through the *shape* attribute of the DataFrame, which returns a tuple. The first value of the tuple is the number of rows and the second value is the number of columns.

```
ipl_auction_df.shape
```

```
(130, 26)
```

In this case, IPL dataset contains 130 records and 26 columns.

More detailed summary about the dataset such as the number of records, columns names, number of actual populated values in each column, datatypes of columns and total computer memory consumed by the DataFrame can be retrieved using *info()* method of the DataFrame.

```
ipl_auction_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 130 entries, 0 to 129
Data columns (total 26 columns):
Sl.NO.          130   non-null  int64
PLAYER NAME     130   non-null  object
AGE             130   non-null  int64
COUNTRY         130   non-null  Object
TEAM            130   non-null  Object
PLAYING ROLE    130   non-null  Object
T-RUNS          130   non-null  int64
T-WKTS          130   non-null  int64
ODI-RUNS-S      130   non-null  int64
ODI-SR-B        130   non-null  float64
ODI-WKTS        130   non-null  int64
ODI-SR-BL       130   non-null  float64
CAPTAINCY EXP   130   non-null  int64
RUNS-S          130   non-null  int64
```

```
HS              130   non-null   int64
AVE             130   non-null   float64
SR-B            130   non-null   float64
SIXERS          130   non-null   int64
RUNS-C          130   non-null   int64
WKTS            130   non-null   int64
AVE-BL          130   non-null   float64
ECON            130   non-null   float64
SR-BL           130   non-null   float64
AUCTION YEAR    130   non-null   int64
BASE PRICE      130   non-null   int64
SOLD PRICE      130   non-null   int64
dtypes: float64(7), int64(15), object(4)
memory usage: 26.5+ KB
```

It shows a value of 130 against every column, which indicates none of the columns (features) has any missing values. Pandas automatically infer the data type of the columns by analyzing the values of each column. If the type cannot be inferred or contains texts or literals, the column is inferred as an *object*. Continuous variables are typically inferred as either *int64* or *float64*, whereas categorical variables (strings or literals) are inferred as objects. (The variable types are discussed in detail in Chapter 3.)

As shown by the output of *info()* method, in this dataset pandas found 7 columns to be of *float* type, 15 columns to be of *integer* type and remaining 4 columns as *object* type. The DataFrame consumes a total of 26.5 KB memory.

2.1.5 | Slicing and Indexing of DataFrame

Sometimes only a subset of rows and columns are analyzed from the complete dataset. To select few rows and columns, the DataFrame can be accessed or sliced by indexes or names. The row and column indexes always start with value 0.

Assume that we are interested in displaying the first 5 rows of the DataFrame. The index range takes two values separated by a colon. For example, [0:5] means start with row with index 0 and end with row with index 5, but not including 5. [0:5] is same as [:5]. By default, the indexing always starts with 0.

```
ipl_auction_df[0:5]
```

	Sl. NO.	Player Name	Age	...	Auction Year	Base Price	Sold Price
0	1	Abdulla, YA	2	...	2009	50000	50000
1	2	Abdur Razzak	2	...	2008	50000	50000
2	3	Agarkar, AB	2	...	2008	200000	350000
3	4	Ashwin, R	1	...	2011	100000	850000
4	5	Badrinath, S	2	...	2011	100000	800000

Negative indexing is an excellent feature in Python and can be used to select records from the bottom of the DataFrame. For example, [−5:] can be used to select the last five records.

```
ipl_auction_df[-5:]
```

	Sl. NO.	Player Name	Age	...	Auction Year	Base Price	Sold Price
125	126	Yadav, AS	2	...	2010	50000	750000
126	127	Younis Khan	2	...	2008	225000	225000
127	128	Yuvraj Singh	2	...	2011	400000	1800000
128	129	Zaheer Khan	2	...	2008	200000	450000
129	130	Zoysa, DNT	2	...	2008	100000	110000

Specific columns of a DataFrame can also be selected or sliced by column names. For example, to select only the player names of the first five records, we can use the following code:

```
ipl_auction_df['PLAYER NAME'][0:5]
```

```
0        Abdulla, YA
1       Abdur Razzak
2        Agarkar, AB
3          Ashwin, R
4       Badrinath, S
Name: PLAYER NAME, dtype: object
```

To select two columns, for example, player name and the country name of the first five records, pass a list of column names to the DataFrame, as shown below:

```
ipl_auction_df[['PLAYER NAME', 'COUNTRY']][0:5]
```

	Player Name	Country
0	Abdulla, YA	SA
1	Abdur Razzak	BAN
2	Agarkar, AB	IND
3	Ashwin, R	IND
4	Badrinath, S	IND

Specific rows and columns can also be selected using row and column indexes. For example, to select first five records starting from row index 4 and columns ranging from column index 1 (second column) to column index 4, pass as below. [4:9, 1:4] takes the row index ranges first and column ranges as second parameter. It should be passed to *iloc method of DataFrame*.

```
ipl_auction_df.iloc[4:9, 1:4]
```

	Player Name	Age	Country
4	Badrinath, S	2	IND
5	Bailey, GJ	2	AUS
6	Balaji, L	2	IND
7	Bollinger, DE	2	AUS
8	Botha, J	2	SA

In the subsequent sections, we will discuss many built-in functions of pandas to explore the dataset further. But rather than exploring each function one by one, to make our learning interesting, we will seek specific insights from the dataset, and figure out how to accomplish that using pandas.

2.1.6 | Value Counts and Cross Tabulations

value_counts() provides the occurrences of each unique value in a column. For example, we would like to know how many players from different countries have played in the IPL, then the method *value_counts()* can be used. It should primarily be used for categorical variables.

```
ipl_auction_df.COUNTRY.value_counts()
```

```
IND       53
AUS       22
SA        16
SL        12
PAK        9
NZ         7
WI         6
ENG        3
ZIM        1
BAN        1
Name: COUNTRY, dtype: int64
```

As expected, most players auctioned are from India, followed by Australia and then South Africa.

Passing parameter *normalize=True* to the *value_counts()* will calculate the percentage of occurrences of each unique value.

```
ipl_auction_df.COUNTRY.value_counts(normalize=True)*100
```

```
IND      40.769231
AUS      16.923077
SA       12.307692
SL        9.230769
```

```
PAK        6.923077
NZ         5.384615
WI         4.615385
ENG        2.307692
ZIM        0.769231
BAN        0.769231
Name: COUNTRY, dtype: float64
```

Cross-tabulation features will help find occurrences for the combination of values for two columns. For example, cross tabulation between *PLAYING ROLE* and *AGE* will give number of players in each age category for each playing role. Ages of the players at the time of auction are classified into three categories. Category 1 means the player is less than 25 years old, category 2 means that the age is between 25 and 35 years, and category 3 means that the age is more than 35. To find out such occurrences across the combination of two categories, we can use *crosstab()* method as shown below:

```
pd.crosstab( ipl_auction_df['AGE'], ipl_auction_df['PLAYING ROLE'] )
```

Playing Role / Age	Allrounder	Batsman	Bowler	W. Keeper
1	4	5	7	0
2	25	21	29	11
3	6	13	8	1

Most of the players auctioned are from the age category 2. In category 1, there are more bowlers than other playing roles and in category 3, there are more batsman than other playing roles.

2.1.7 | Sorting DataFrame by Column Values

sort_values() takes the column names, based on which the records need to be sorted. By default, sorting is done in ascending order. The following line of code selects two columns *PLAYER NAME* and *SOLD PRICE* from the DataFrame and then sorts the rows by *SOLD PRICE*.

```
ipl_auction_df[['PLAYER NAME', 'SOLD PRICE']].sort_values
('SOLD PRICE')[0:5]
```

	Player Name	Sold Price
73	Noffke, AA	20000
46	Kamran Khan	24000
0	Abdulla, YA	50000
1	Abdur Razzak	50000
118	Van der Merwe	50000

To sort the DataFrame records in descending order, pass *False* to *ascending* parameter.

```
ipl_auction_df[['PLAYER NAME', 'SOLD PRICE']].sort_values('SOLD
PRICE', ascending = False)[0:5]
```

	Player Name	Sold Price
93	Sehwag, V	1800000
127	Yuvraj Singh	1800000
50	Kohli, V	1800000
111	Tendulkar, SR	1800000
113	Tiwary, SS	1600000

2.1.8 | Creating New Columns

We can create new columns (new features) by applying basic arithmetic operations on existing columns. For example, let us say we would like to create a new feature which is the difference between the sold price and the base price (called by new feature name premium). We will create a new column called *premium* and populate by taking difference between the values of columns *SOLD PRICE* and *BASE PRICE.*

```
ipl_auction_df['premium'] = ipl_auction_df['SOLD PRICE'] -
                            ipl_auction_df['BASE PRICE']
```

```
ipl_auction_df[['PLAYER NAME', 'BASE PRICE', 'SOLD PRICE',
'premium']][0:5]
```

	Player Name	Base Price	Sold Price	Premium
0	Abdulla, YA	50000	50000	0
1	Abdur Razzak	50000	50000	0
2	Agarkar, AB	200000	350000	150000
3	Ashwin, R	100000	850000	750000
4	Badrinath, S	100000	800000	700000

To find which players got the maximum premium offering on their base price, the DataFrame can be sorted by the values of column *premium* in descending order. *sort_values()* method sorts a DataFrame by a column whose name is passed as a parameter along with order type. The default order type is ascending. For sorting in descending order, the parameter *ascending* needs to be set to *False.*

```
ipl_auction_df[['PLAYER NAME',
                'BASE PRICE',
                'SOLD PRICE', 'premium']].sort_values('premium',
                                         ascending = False)[0:5]
```

	Player Name	Base Price	Sold Price	Premium
50	Kohli, V	150000	1800000	1650000
113	Tiwary, SS	100000	1600000	1500000
127	Yuvraj Singh	400000	1800000	1400000
111	Tendulkar, SR	400000	1800000	1400000
93	Sehwag, V	400000	1800000	1400000

The result shows Virat Kohli was auctioned with maximum premium on the base price set.

2.1.9 | Grouping and Aggregating

Sometimes, it may be required to group records based on column values and then apply aggregated operations such as mean, maximum, minimum, etc. For example, to find average *SOLD PRICE* for each age category, group all records by *AGE* and then apply *mean()* on *SOLD PRICE* column.

```
ipl_auction_df.groupby('AGE')['SOLD PRICE'].mean()
```

```
AGE
1     720250.000000
2     484534.883721
3     520178.571429
Name: SOLD PRICE, dtype: float64
```

The average price is highest for age category 1 and lowest for age category 2.

The above operation returns a *pd.Series data* structure. To create a DataFrame, we can call *reset_index()* as shown below on the returned data structure.

```
soldprice_by_age = ipl_auction_df.groupby('AGE')['SOLD PRICE'].
                   mean().reset_index()
print(soldprice_by_age)
```

	Age	Sold Price
0	1	720250.000000
1	2	484534.883721
2	3	520178.571429

Data can be grouped using multiple columns. For example, to find average *SOLD PRICE* for players for each *AGE* and *PLAYING ROLE* category, multiple columns can be passed to *groupby()* method as follows:

```
soldprice_by_age_role = ipl_auction_df.groupby(['AGE', 'PLAYING
                        ROLE'])['SOLD PRICE'].mean().reset_index()
print(soldprice_by_age_role)
```

	Age	Playing Role	Sold Price
0	1	Allrounder	587500.000
1	1	Batsman	1110000.000
2	1	Bowler	517714.286
3	2	Allrounder	449400.000
4	2	Batsman	654761.905
5	2	Bowler	397931.034
6	2	W. Keeper	467727.273
7	3	Allrounder	766666.667
8	3	Batsman	457692.308
9	3	Bowler	414375.000
10	3	W. Keeper	700000.000

2.1.10 | Joining DataFrames

We may need to combine columns from multiple DataFrames into one single DataFrame. In this case both DataFrames need to have a common column. Then the remaining column values can be retrieved from both the DataFrames and joined to create a row in the resulting DataFrame. To merge two DataFrames, pandas method *merge()* can be called from one of the DataFrames and the other DataFrame can be passed as a parameter to it. It also takes a parameter *on*, which is the common column in both the DataFrames and should be present in both the DataFrames. DataFrames can be joined based on multiple common columns. The join type can be of inner, outer, left or right joins and should be specified in the *how* parameter. For understanding different joins, please refer to the examples given at *https://www.w3schools.com/sql/sql_join.asp*

For example, to compare the average *SOLD PRICE* for different *AGE* categories with the different age and *PLAYING ROLE* categories, we need to merge the DataFrames *soldprice_by_age* and *soldprice_by_age_role*. The common column is *AGE* and this needs outer join.

We will join the above two DataFrames created and then compare the difference in auction prices.

```
soldprice_comparison = soldprice_by_age_role.merge (soldprice_by_age,
                                                    on = 'AGE',
                                                   how = 'outer')
```

```
soldprice_comparison
```

	Age	Playing Role	Sold Price_x	Sold Price_y
0	1	Allrounder	587500.000	720250.000
1	1	Batsman	1110000.000	720250.000
2	1	Bowler	517714.286	720250.000

	Age	Playing Role	Sold Price_x	Sold Price_y
3	2	Allrounder	449400.000	484534.884
4	2	Batsman	654761.905	484534.884
5	2	Bowler	397931.034	484534.884
6	2	W. Keeper	467727.273	484534.884
7	3	Allrounder	766666.667	520178.571
8	3	Batsman	457692.308	520178.571
9	3	Bowler	414375.000	520178.571
10	3	W. Keeper	700000.000	520178.571

Because the column name SOLD PRICE is same in both the DataFrames, it automatically renames them to_x and_y. *SOLD PRICE_x* comes from the left table (*soldprice_by_age_role*) and *SOLD PRICE_y* comes from the right table (*soldprice_by_age*).

2.1.11 | Re-Naming Columns

The existing columns of a DataFrame can be renamed using *rename()* method. For renaming multiple columns simultaneously, the method can take a dictionary as a parameter, where the keys should be existing column names and the values should the new names to be assigned.

```
soldprice_comparison.rename( columns = { 'SOLD PRICE_x': 'SOLD_PRICE_
AGE_ROLE','SOLD PRICE_y': 'SOLD_PRICE_AGE' }, inplace = True )
```

```
soldprice_comparison.head(5)
```

	Age	Playing Role	Sold_price_age_role	Sold_price_age
0	1	Allrounder	587500.000	720250.000
1	1	Batsman	1110000.000	720250.000
2	1	Bowler	517714.286	720250.000
3	2	Allrounder	449400.000	484534.884
4	2	Batsman	654761.905	484534.884

2.1.12 | Applying Operations to Multiple Columns

Consider as an example a situation where we would like to find whether players carry a premium if they belong to a specific *AGE* and *PLAYING ROLE* category. The premium (we will call it change) is calculated in percentage terms and calculated as follows:

$$\text{Change} = \frac{\begin{array}{c}(\text{Average } \textit{SOLD PRICE} \text{ for all player in an } \textit{AGE} \text{ and } \textit{PLAYING ROLE} \text{ category} \\ - \text{ Average } \textit{SOLD PRICE} \text{ for all player in an } \textit{AGE} \text{ category})\end{array}}{\text{Average } \textit{SOLD PRICE} \text{ for all player in an } \textit{AGE} \text{ category})}$$

To accomplish this, we need to iterate through each row in the DataFrame and then apply the above calculations to the columns. The resulting value should be added as a new column to the existing DataFrame. The function *apply()* can apply a function along any axis of the DataFrame.

```
soldprice_comparison['change'] = soldprice_comparison.apply(lambda
                                 rec:(rec.SOLD_PRICE_AGE_ROLE -
                                 rec.SOLD_PRICE_AGE) / rec.SOLD_
                                 PRICE_AGE, axis = 1)
```

```
soldprice_comparison
```

	Age	Playing Role	Sold_price_age_role	Sold_price_age	Change
0	1	Allrounder	587500.000	720250.000	–0.184
1	1	Batsman	1110000.000	720250.000	0.541
2	1	Bowler	517714.286	720250.000	–0.281
3	2	Allrounder	449400.000	484534.884	–0.073
4	2	Batsman	654761.905	484534.884	0.351
5	2	Bowler	397931.034	484534.884	–0.179
6	2	W. Keeper	467727.273	484534.884	–0.035
7	3	Allrounder	766666.667	520178.571	0.474
8	3	Batsman	457692.308	520178.571	–0.120
9	3	Bowler	414375.000	520178.571	–0.203
10	3	W. Keeper	700000.000	520178.571	0.346

2.1.13 | Filtering Records Based on Conditions

Assume that we would like to filter certain records such as the players who have hit more than 80 sixers in the IPL tournament. DataFrame records can be filtered using a condition as indexing mechanism. Those records for which the condition returns *True* are selected to be part of the resulting DataFrame.

```
ipl_auction_df[ipl_auction_df['SIXERS'] > 80 ][['PLAYER NAME',
'SIXERS']]
```

	Player Name	Sixers
26	Gayle, CH	129
28	Gilchrist, AC	86
82	Pathan, YK	81
88	Raina, SK	97
97	Sharma, RG	82

2.1.14 | Removing a Column or a Row from a Dataset

To remove a column or a row, we can use *drop()* method on the DataFrame. It takes a parameter *axis* to specify if a column or a row needs to be dropped.

1. To drop a column, pass the column name and axis as 1.
2. To drop a row, pass the row index and axis as 0.

We will drop the column *Sl.NO.* in the IPL dataset as we do not need this column for our exploration. Most of these DataFrame operations always result in creating new DataFrames. But creating new DataFrame each time can result in over consumption of computer memory. To avoid creating new DataFrames and make changes to the existing DataFrame, there is another parameter *inplace* available, which can be set to *True.*

```
ipl_auction_df.drop('Sl.NO.', inplace = True, axis = 1)
```

```
ipl_auction_df.columns
```

```
Index (['PLAYER NAME', 'AGE', 'COUNTRY', 'TEAM', 'PLAYING ROLE',
        'T-RUNS', 'T-WKTS', 'ODI-RUNS-S', 'ODI-SR-B', 'ODI-WKTS',
        'ODI-SR-BL', 'CAPTAINCY EXP', 'RUNS-S', 'HS', 'AVE',
        'SR-B', 'SIXERS', 'RUNS-C', 'WKTS', 'AVE-BL', 'ECON', 'SR-BL',
        'AUCTION YEAR', 'BASE PRICE', 'SOLD PRICE', 'premium'],
         dtype = 'object')
```

2.2 | HANDLING MISSING VALUES

In real world, the datasets are not clean and may have missing values. We must know how to find and deal with these missing values. One of the strategies to deal with missing values is to remove them from the dataset. However, whenever possible, we would like to impute the data, which is a process of filling the missing values. Data imputation techniques will be discussed in the later chapter. Here we will discuss how to find missing values and remove them. We will use an example dataset called *autos-mpg.data*[1], which contains information about different cars and their characteristics such as

1. mpg – miles per gallon
2. cylinders – Number of cylinders (values between 4 and 8)
3. displacement – Engine displacement (cu. inches)
4. horsepower – Engine horsepower
5. weight – Vehicle weight (lbs.)
6. acceleration – Time to accelerate from 0 to 60 mph (sec.)
7. year – Model year (modulo 100)
8. origin – Origin of car (1. American, 2. European, 3. Japanese)
9. name – Vehicle name

1 The dataset can be downloaded from https://archive.ics.uci.edu/ml/datasets/auto+mpg (https://archive.ics.uci.edu/ml/datasets/auto+mpg).

The dataset contains *SPACE* separated values and has no header. To read the dataset onto the DataFrame,

Use *pd.read_csv* method, Set *separator* to *\s+* and Set *header* to None

```
autos = pd.read_csv('auto-mpg.data', sep = '\s+', header = None)
autos.head(5)
```

	0	1	2	...	6	7	8
0	18.0	8	307.0	...	70	1	Chevrolet chevelle malibu
1	15.0	8	350.0	...	70	1	buick skylark 320
2	18.0	8	318.0	...	70	1	plymouth satellite
3	16.0	8	304.0	...	70	1	amc rebel sst
4	17.0	8	302.0	...	70	1	ford torino

5 rows × 9 columns

As the dataset does not have a header, the columns are not named. We can name the columns by assigning a list of names to the DataFrame header. The column names are given in the dataset description page at the link *https://archive.ics.uci.edu/ml/machine-learning-databases/auto-mpg/auto-mpg.names*

1. mpg: continuous
2. cylinders: multi-valued discrete
3. displacement: continuous
4. horsepower: continuous
5. weight: continuous
6. acceleration: continuous
7. model year: multi-valued discrete
8. origin: multi-valued discrete
9. car name: string (unique for each instance)

We will create a list of the above names and assign to the DataFrame's *columns* attribute as follows:

```
autos.columns = ['mpg','cylinders', 'displacement',
                 'horsepower', 'weight', 'acceleration',
                 'year', 'origin', 'name']
autos.head(5)
```

	mpg	cylinders	displacement	...	year	origin	name
0	18.0	8	307.0	...	70	1	chevrolet chevelle malibu
1	15.0	8	350.0	...	70	1	buick skylark 320
2	18.0	8	318.0	...	70	1	plymouth satellite
3	16.0	8	304.0	...	70	1	amc rebel sst
4	17.0	8	302.0	...	70	1	ford torino

5 rows × 9 columns

The schema of the DataFrame is as follows:

```
autos.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 398 entries, 0 to 397
Data columns (total 9 columns):
mpg              398  non-null  float64
cylinders        398  non-null  int64
displacement     398  non-null  float64
horsepower       398  non-null  object
weight           398  non-null  float64
acceleration     398  non-null  float64
year             398  non-null  int64
origin           398  non-null  int64
name             398  non-null  object
dtypes: float64(4), int64(3), object(2)
memory usage: 28.1+ KB
```

Here the column *horsepower* has been inferred as *object*, whereas it should have been inferred as *float64*. This may be because some of the rows contain non-numeric values in the *horsepower* column. One option to deal with this is to force convert the *horsepower* column into numeric, which should convert the non-numeric values into null values. *pd.to_numeric()* can be used to convert any column with non-numeric datatype to a column with numeric datatype. It takes a parameter *errors,* which specifies how to deal with non-numeric values. The following are the possible parameter values (source: *pandas.pydata.org*):

errors : {'ignore', 'raise', 'coerce'}, default 'raise'

1. If 'raise', then invalid parsing will raise an exception.
2. If 'coerce', then invalid parsing will be set as NaN.
3. If 'ignore', then invalid parsing will return the input.

In our case, we will use *errors* = '*coerce*' to convert any non-numeric values to null *NaN*.

```
autos["horsepower"] = pd.to_numeric(autos["horsepower"],
errors = 'corece') autos.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 398 entries, 0 to 397
Data columns (total 9 columns):
mpg              398  non-null  float64
cylinders        398  non-null  int64
displacement     398  non-null  float64
horsepower       392  non-null  float64
weight           398  non-null  float64
acceleration     398  non-null  float64
```

```
year              398  non-null  int64
origin            398  non-null  int64
name              398  non-null  object
dtypes: float64(5), int64(3), object(1)
memory usage: 28.1+ KB
```

The column *horsepower* has been converted into *float64* type. We can verify if some of the rows contain null values in *horsepower* column. This can be done by using *isnull()* method on the DataFrame column as shown in the code below.

```
autos[autos.horsepower.isnull()]
```

	mpg	cylinders	displacement	...	year	origin	name
32	25.0	4	98.0	...	71	1	ford pinto
126	21.0	6	200.0	...	74	1	ford maverick
330	40.9	4	85.0	...	80	2	renault lecar deluxe
336	23.6	4	140.0	...	80	1	ford mustang cobra
354	34.5	4	100.0	...	81	2	renault 18i
374	23.0	4	151.0	...	82	1	ame concord di

6 rows × 9 columns

There are 6 rows which contain null values. These rows can be dropped from the DataFrame using *dropna()* method. *Dropna()* method removes all rows with NaN values. In this dataset, only one feature – horse power – contains NaNs. Hence subset argument as *dropna()* method should be used.

```
autos = autos.dropna(subset = ['horsepower']
```

We can verify if the rows with null values have been removed by applying the filtering condition again.

```
autos[autos.horsepower.isnull()]
```

mpg	cylinders	displacement	...	year	origin	name

0 rows × 9 columns

Filtering with *isnull()* shows no rows with null values anymore. Similarly, *fillna()* can take a default value to replace the null values in a column. For example, *fillna(0)* replaces all null values in a column with value 0.

2.3 | EXPLORATION OF DATA USING VISUALIZATION

Data visualization is useful to gain insights and understand what happened in the past in a given context. It is also helpful for feature engineering. In this section we will be discussing various plots that we can draw using Python.

2.3.1 | Drawing Plots

Matplotlib is a Python 2D plotting library and most widely used library for data visualization. It provides extensive set of plotting APIs to create various plots such as scattered, bar, box, and distribution plots with custom styling and annotation. Detailed documentation for *matplotlib* can be found at *https://matplotlib.org/*. Seaborn is also a Python data visualization library based on matplotlib. It provides a high-level interface for drawing innovative and informative statistical charts (source: *https://seaborn.pydata.org/*).

Matplotlib is a library for creating 2D plots of arrays in Python. Matplotlib is written in Python and makes use of NumPy arrays. It is well integrated with pandas to read columns and create plots. *Seaborn*, which is built on top of matplotlib, is a library for making elegant charts in Python and is well integrated with pandas DataFrame.

To create graphs and plots, we need to import *matplotlib.pyplot* and *seaborn* modules. To display the plots on the Jupyter Notebook, we need to provide a directive *%matplotlib inline*. Only if the directive is provided, the plots will be displayed on the notebook.

```
import matplotlib.pyplot as plt
import seaborn as sn
%matplotlib inline
```

2.3.2 | Bar Chart

Bar chart is a frequency chart for qualitative variable (or categorical variable). Bar chart can be used to assess the most-occurring and least-occurring categories within a dataset.

To draw a bar chart, call *barplot()* of seaborn library. The DataFrame should be passed in the parameter *data*. To display the average sold price by each age category, pass *SOLD PRICE* as *y* parameter and *AGE* as *x* parameter. Figure 2.3 shows a bar plot created to show the average SOLD PRICE for each age category.

```
sn.barplot(x = 'AGE', y = 'SOLD PRICE', data = soldprice_by_age);
```

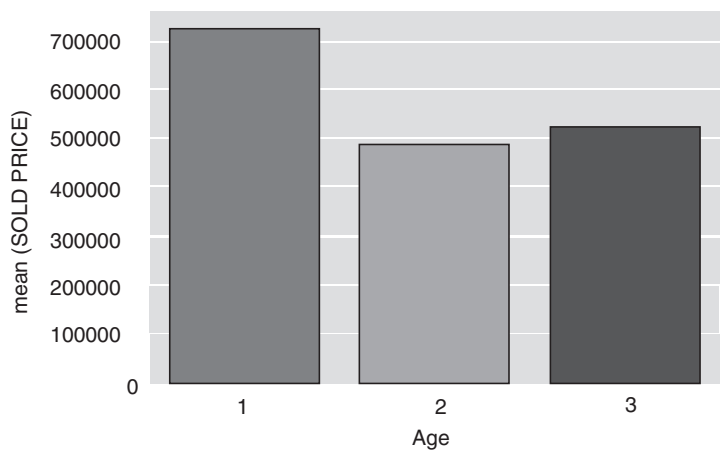

FIGURE 2.3 Bar plot for average sold price versus age.

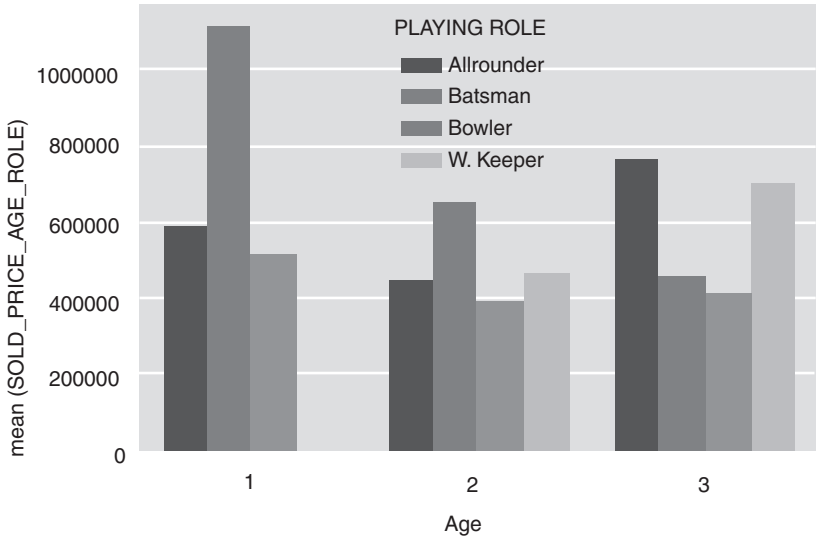

FIGURE 2.4 Bar plot for average sold price versus age by player roles. Here the first bar represents Allrounder, the second Batsman, the third Bowler, the fourth W. Keeper.

In Figure 2.3, it can be noted that the average sold price is higher for age category 1. We can also create bar charts, which are grouped by a third variable.

In the following example (Figure 2.4), average sold price is shown for each age category but grouped by a third variable, that is, playing roles. The parameter *hue* takes the third variable as parameter. In this case, we pass *PLAYING ROLE* as *hue* parameter.

```
sn.barplot(x = 'AGE', y = 'SOLD_PRICE_AGE_ROLE', hue = 'PLAYING
ROLE', data = soldprice_comparison);
```

In Figure 2.4, it can be noted that in the age categories 1 and 2, batsmen are paid maximum, whereas allrounders are paid maximum in the age category 3. This could be because allrounders establish their credentials as good allrounders over a period.

2.3.3 | Histogram

A **histogram** is a plot that shows the frequency distribution of a set of continuous variable. Histogram gives an insight into the underlying distribution (e.g., normal distribution) of the variable, outliers, skewness, etc. To draw a histogram, invoke *hist()* method of matplotlib library. The following is an example of how to draw a histogram for *SOLD PRICE* and understand its distribution (Figure 2.5).

```
plt.hist( ipl_auction_df['SOLD PRICE'] );
```

The histogram shows that *SOLD PRICE* is right skewed. Most players are auctioned at low price range of 250000 and 500000, whereas few players are paid very highly, more than 1 million dollars.

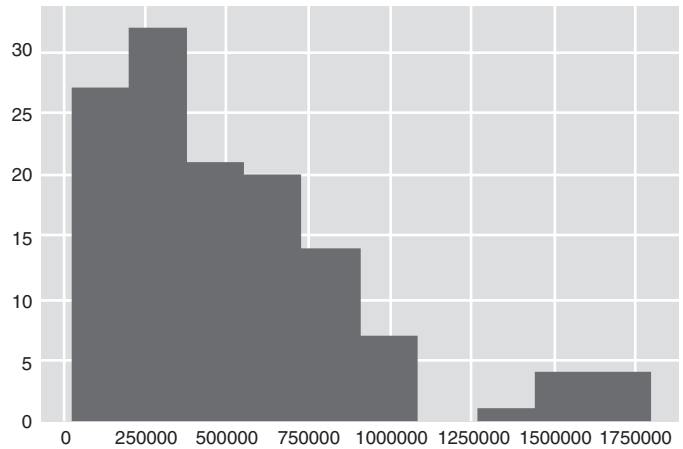

FIGURE 2.5 Histogram for SOLD PRICE.

By default, it creates 10 bins in the histogram. To create more bins, the bins parameter can be set in the *hist()* method as follows:

```
plt.hist( ipl_auction_df['SOLD PRICE'], bins = 20 );
```

Histogram for SOLD PRICE with 20 bins is shown in Figure 2.6.

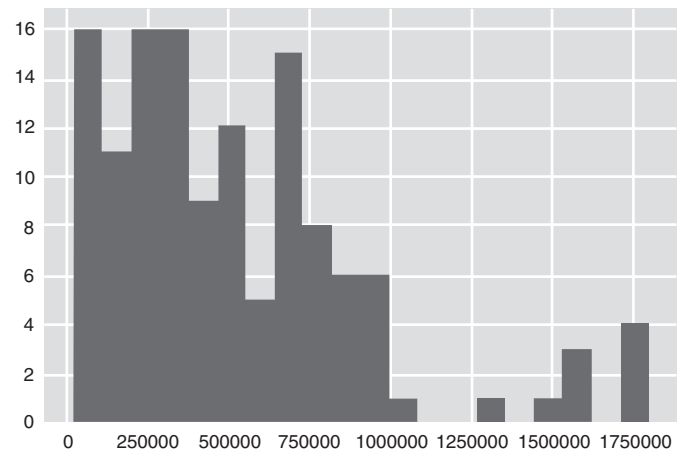

FIGURE 2.6 Histogram for SOLD PRICE with 20 bins.

2.3.4 | Distribution or Density Plot

A **distribution or density** plot depicts the distribution of data over a continuous interval. Density plot is like smoothed histogram and visualizes distribution of data over a continuous interval. So, a density plot also gives insight into what might be the distribution of the population.

To draw the distribution plot, we can use *distplot()* of seaborn library. Density plot for the outcome variable "SOLD PRICE" is shown in Figure 2.7.

```
sn.distplot(ipl_auction_df['SOLD PRICE']);
```

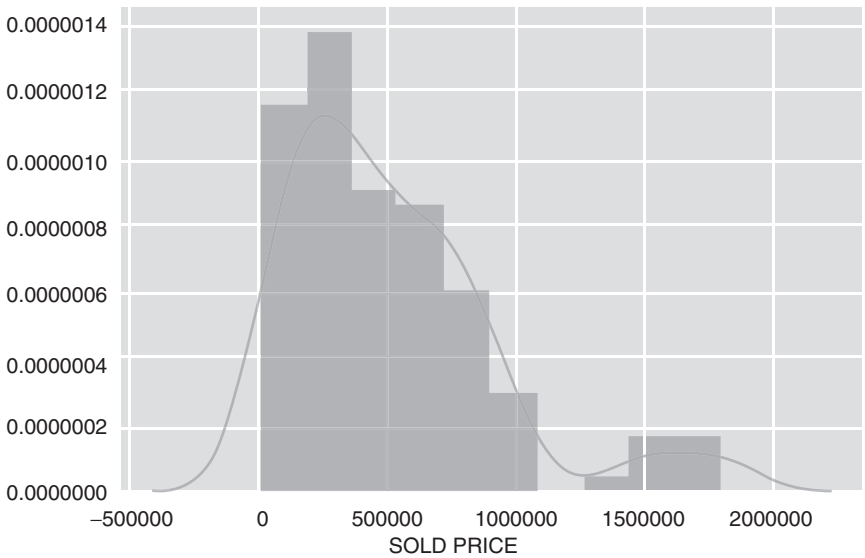

FIGURE 2.7 Distribution plot for SOLD PRICE.

2.3.5 | Box Plot

Box plot (aka Box and Whisker plot) is a graphical representation of numerical data that can be used to understand the variability of the data and the existence of outliers. Box plot is designed by identifying the following descriptive statistics:

1. Lower quartile (1st quartile), median and upper quartile (3rd quartile).
2. Lowest and highest values.
3. Inter-quartile range (IQR).

Box plot is constructed using IQR, minimum and maximum values. IQR is the distance (difference) between the 3rd quartile and 1st quartile. The length of the box is equivalent to IQR. It is possible that the data may contain values beyond Q1 – 1.5IQR and Q3 + 1.5IQR. The whisker of the box plot extends till Q1 – 1.5IQR and Q3 + 1.5IQR; observations beyond these two limits are potential outliers.

To draw the box plot, call *boxplot()* of seaborn library. The box plot for SOLD PRICE is shown in Figure 2.8 and indicates that there are few outliers in the data.

```
box = sn.boxplot(ipl_auction_df['SOLD PRICE']);
```

To obtain min, max, 25 percentile (1st quartile) and 75 percentile (3rd quantile) values in the boxplot, *boxplot()* method of matplotlib can be used. Let us assign the return value of boxplot() to a variable *box*.

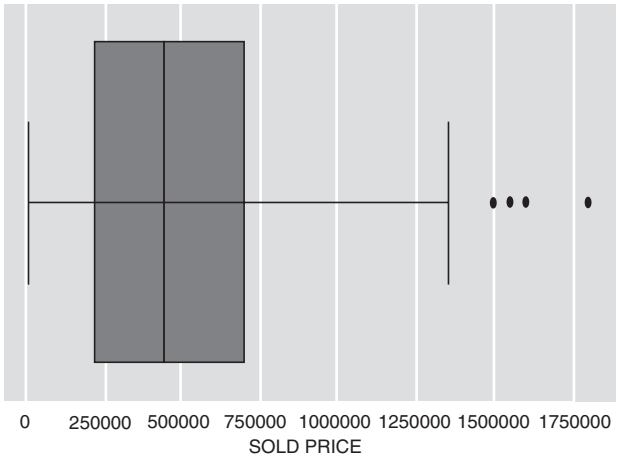

FIGURE 2.8 Box plot for SOLD PRICE.

```
box = plt.boxplot(ipl_auction_df['SOLD PRICE']);
```

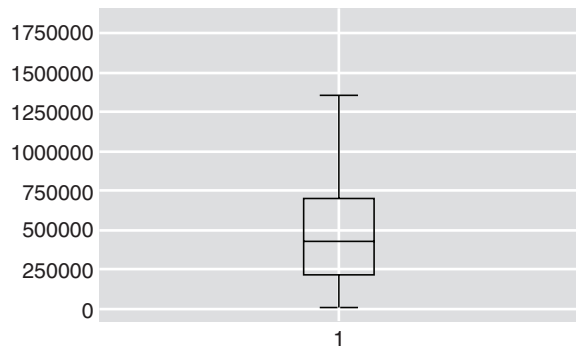

FIGURE 2.9 Box plot for SOLD PRICE with median line.

The above line of code creates a box plot as shown in Figure 2.9 and returns details of box plot in variable box. The *caps* key in box variable returns the min and max values of the distribution. Here the minimum auction price offered is 20,000 and maximum price is 13,50,000.

```
[item.get_ydata()[0] for item in box['caps']]
```

```
[20000.0, 1350000.0]
```

The *whiskers* key in box variable returns the values of the distribution at 25 and 75 quantiles. Here the min value is 225,000 and max is 700,000.

```
[item.get_ydata()[0] for item in box['whiskers']]
```

```
[225000.0, 700000.0]
```

So, inter-quartile range (IQR) is 700,000 – 225,000 = 475,000.

The *medians* key in box variable returns the median value of the distribution. Here the median value is 437,500.

```
[item.get_ydata()[0] for item in box['medians']]
```

```
[437500.0]
```

The box plot in Figure 2.8 shows that some of the players are auctioned at *SOLD PRICE*, which seem to be outliers. Let us find out the player names.

```
ipl_auction_df[ipl_auction_df['SOLD PRICE'] > 1350000.0][['PLAYER
NAME', 'PLAYING ROLE', 'SOLD PRICE']]
```

	Player Name	Playing Role	Sold Price
15	Dhoni, MS	W. Keeper	1500000
23	Flintoff, A	Allrounder	1550000
50	Kohli, V	Batsman	1800000
83	Pietersen, KP	Batsman	1550000
93	Sehwag, V	Batsman	1800000
111	Tendulkar, SR	Batsman	1800000
113	Tiwary, SS	Batsman	1600000
127	Yuvraj Singh	Batsman	1800000

The SOLD PRICE amount for the above players seems to be outlier considering what is paid to other players.

2.3.6 | Comparing Distributions

The distribution for different categories can be compared by overlapping the distributions. For example, the ***SOLD PRICE* of players with and without *CAPTAINCY EXP* can be compared as below.** Distributions can be compared by drawing distribution for each category and overlapping them in one plot.

```
sn.distplot(ipl_auction_df[ipl_auction_df['CAPTAINCY EXP'] == 1]
           ['SOLD PRICE'],
           color = 'y',
           label = 'Captaincy Experience')
sn.distplot(ipl_auction_df[ipl_auction_df['CAPTAINCY EXP'] == 0]
           ['SOLD PRICE'],
           color = 'r',
           label = 'No Captaincy Experience');
plt.legend();
```

Comparison plot for SOLD PRICE with and without captaincy experience is shown in Figure 2.10. It can be observed that players with captaincy experience seem to be paid higher.

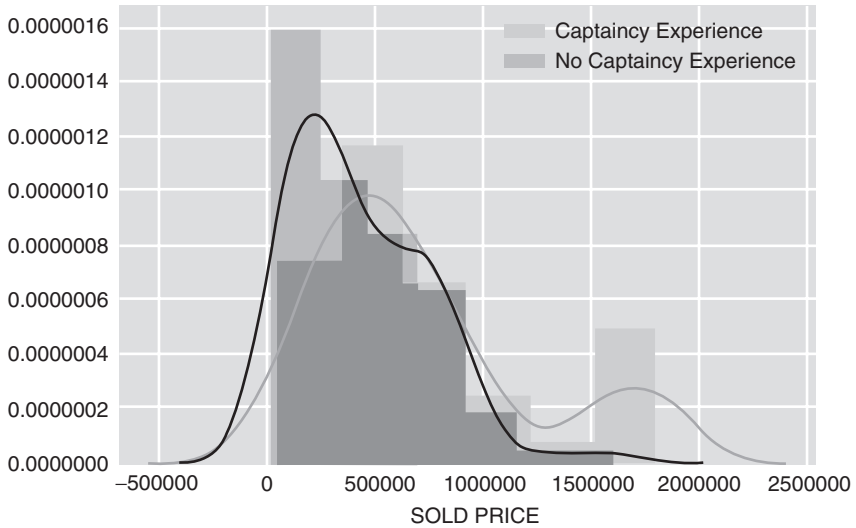

FIGURE 2.10 Distribution plots comparing SOLD PRICE with and without captaincy experience.

Similarly, the distributions can be compared using box plots. For example, if we want to visualize *SOLD PRICE* for each *PLAYING ROLE*, then pass *PLAYING ROLE* as x parameter and *SOLD PRICE* as y parameter to *boxplot()* method of seaborn.

An example of comparing the *SOLD PRICE* for players with different *PLAYING ROLE* is shown in Figure 2.11.

```
sn.boxplot(x = 'PLAYING ROLE', y = 'SOLD PRICE',
               data = ipl_auction_df);
```

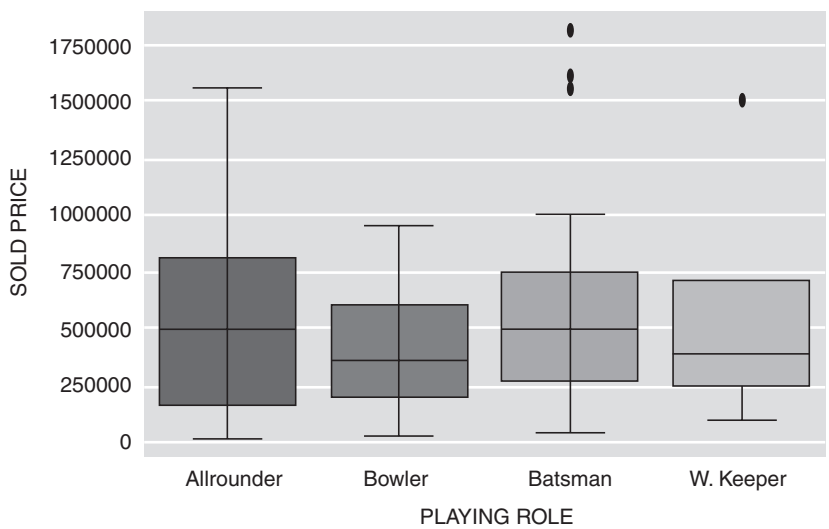

FIGURE 2.11 Box plot of SOLD PRICE for different playing roles.

Few observations from the plot in Figure 2.11 are as follows:

1. The median SOLD PRICE for allrounders and batsmen are higher than bowlers and wicket keepers.
2. Allrounders who are paid more than 1,35,0000 USD are not considered outliers. Allrounders have relatively high variance.
3. There are outliers in batsman and wicket keeper category. In Section 2.3.5, we have already found that MS DHONI is an outlier in the wicket keeper category.

2.3.7 | Scatter Plot

In a scatter plot, the values of two variables are plotted along two axes and resulting pattern can reveal correlation present between the variables, if any. The relationship could be linear or non-linear. A scatter plot is also useful for assessing the strength of the relationship and to find if there are any outliers in the data.

Scatter plots are used during regression model building to decide on the initial model, that is whether to include a variable in a regression model or not.

Since IPL is T20 cricket, it is believed that the number of sixers a player has hit in past would have influenced his SOLD PRICE. A scatter plot between *SOLD PRICE* of batsman and number of sixes the player has hit can establish this correlation. The *scatter()* method of matplotlib can be used to draw the scatter plot which takes both the variables. We will draw the scatter plot only for batsman playing role. Figure 2.12 shows the scatter plot between SIXERS and SOLD PRICE.

```
ipl_batsman_df = ipl_auction_df[ipl_auction_df['PLAYING ROLE'] ==
                 'Batsman']
```

```
plt.scatter(x = ipl_batsman_df.SIXERS,
            y = ipl_batsman_df['SOLD PRICE']);
```

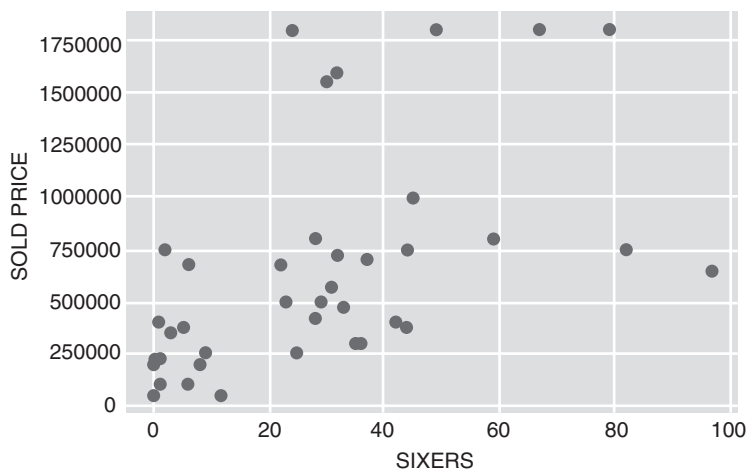

FIGURE 2.12 Scatter plot for SOLD PRICE versus SIXERS.

To draw the direction of relationship between the variables, *regplot()* of seaborn can be used (Figure 2.13).

```
sn.regplot( x = 'SIXERS',
            y = 'SOLD PRICE',
            data = ipl_batsman_df );
```

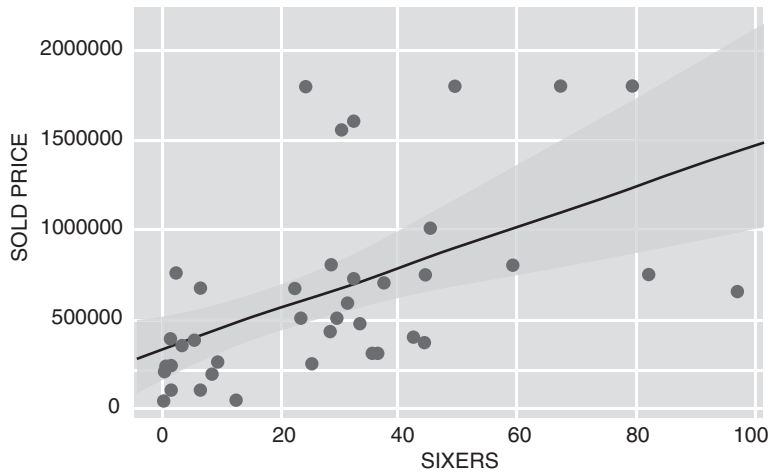

FIGURE 2.13 Scatter plot for SOLD PRICE versus SIXERS with a regression line.

The line in Figure 2.13 shows there is a positive correlation between number of sixes hit by a batsman and the SOLD PRICE.

2.3.8 | Pair Plot

If there are many variables, it is not convenient to draw scatter plots for each pair of variables to understand the relationships. So, a pair plot can be used to depict the relationships in a single diagram which can be plotted using *pairplot()* method.

In this example, we will explore the relationship of four variables, *SR-B, AVE, SIXERS, SOLD PRICE*, which we think may be the influential features in determining the *SOLD PRICE* of batsmen (Figure 2.14).

```
influential_features = ['SR-B', 'AVE', 'SIXERS', 'SOLD PRICE']
```

```
sn.pairplot(ipl_auction_df[influential_features], size=2)
```

The plot is drawn like a matrix and each row and column is represented by a variable. Each cell depicts the relationship between two variables, represented by that row and column variable. For example, the cell on second row and first column shows the relationship between AVE and SR-B. The diagonal of the matrix shows the distribution of the variable.

For all the correlations, *AVE* and *SIXERS* seem to be highly correlated with *SOLD PRICE* compared to *SR-B*.

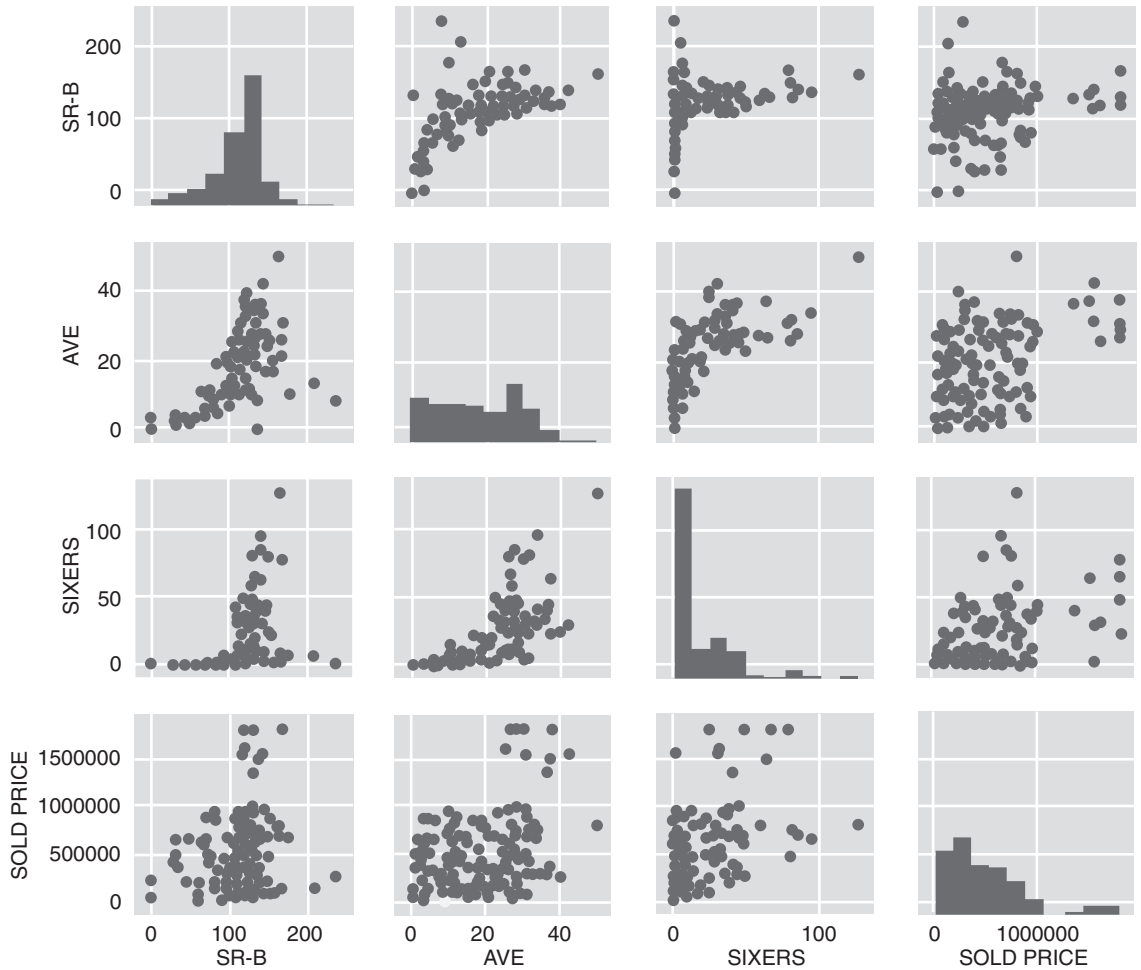

FIGURE 2.14 Pair plot between variables.

2.3.9 | Correlation and Heatmap

Correlation is used for measuring the strength and direction of the linear relationship between two continuous random variables *X* and *Y*. It is a statistical measure that indicates the extent to which two variables change together. A positive correlation means the variables increase or decrease together; a negative correlation means if one variable increases, the other decreases.

1. The correlation value lies between −1.0 and 1.0. The sign indicates whether it is positive or negative correlation.
2. −1.0 indicates a perfect negative correlation, whereas +1.0 indicates perfect positive correlation.

Correlation values can be computed using *corr()* method of the DataFrame and rendered using a heatmap (Figure 2.15).

```
ipl_auction_df[influential_features].corr()
```

	SR-B	AVE	Sixers	Sold Price
SR-B	1.000000	0.583579	0.425394	0.184278
AVE	0.583579	1.000000	0.705365	0.396519
Sixers	0.425394	0.705365	1.000000	0.450609
Sold Price	0.184278	0.396519	0.450609	1.000000

```
sn.heatmap(ipl_auction_df[influential_features].corr(), annot=True);
```

FIGURE 2.15 Heatmap of correlation values.

The color map scale is shown along the heatmap. Setting *annot* attribute to *true* prints the correlation values in each box of the heatmap and improves readability of the heatmap. Here the heatmap shows that *AVE* and *SIXER* show positive correlation, while *SOLD PRICE* and *SR-B* are not so strongly correlated.

CONCLUSION

1. The objective of descriptive analytics is simple comprehension of data using data summarization, basic statistical measures and visualization.
2. DataFrames, which can be visualized as in-memory structured query language (SQL) tables, are widely used data structures for loading and manipulating structured data.
3. *Pandas* library provides excellent support for DataFrame and its operations. Operations like filtering, grouping, aggregations, sorting, joining and many more are readily available in this library.
4. *Matplotlib* and *seaborn* are two most widely used libraries for creating visualization.
5. Plots like histogram, distribution plots, box plots, scatter plots, pair plots, heatmap can be created to find insights during exploratory analysis.

EXERCISES

Use the Bollywood Dataset to Answer Questions 1 to 12.

The data file *bollywood.csv* (link to the datasets is provided in the Preface) contains box office collection and social media promotion information about movies released in 2013–2015 period. Following are the columns and their descriptions.

- *SlNo* – Release Date
- *MovieName* – Name of the movie
- *ReleaseTime* – Mentions special time of release. LW (Long weekend), FS (Festive Season), HS (Holiday Season), N (Normal)
- *Genre* – Genre of the film such as Romance, Thriller, Action, Comedy, etc.
- *Budget* – Movie creation budget
- *BoxOfficeCollection* – Box office collection
- *YoutubeViews* – Number of views of the YouTube trailers
- *YoutubeLikes* – Number of likes of the YouTube trailers
- *YoutubeDislikes* – Number of dislikes of the YouTube trailers

Use Python code to answer the following questions:

1. How many records are present in the dataset? Print the metadata information of the dataset.
2. How many movies got released in each genre? Which genre had highest number of releases? Sort number of releases in each genre in descending order.
3. How many movies in each genre got released in different release times like long weekend, festive season, etc. (Note: Do a cross tabulation between *Genre* and *ReleaseTime.*)
4. Which month of the year, maximum number movie releases are seen? (Note: Extract a new column called month from *ReleaseDate* column.)
5. Which month of the year typically sees most releases of high budgeted movies, that is, movies with budget of 25 crore or more?
6. Which are the top 10 movies with maximum return on investment (ROI)? Calculate return on investment (ROI) as (*BoxOfficeCollection – Budget) / Budget.*
7. Do the movies have higher ROI if they get released on festive seasons or long weekend? Calculate the average ROI for different release times.
8. Draw a histogram and a distribution plot to find out the distribution of movie budgets. Interpret the plot to conclude if the most movies are high or low budgeted movies.
9. Compare the distribution of ROIs between movies with comedy genre and drama. Which genre typically sees higher ROIs?
10. Is there a correlation between box office collection and YouTube likes? Is the correlation positive or negative?
11. Which genre of movies typically sees more YouTube likes? Draw boxplots for each genre of movies to compare.
12. Which of the variables among *Budget, BoxOfficeCollection, YoutubeView, YoutubeLikes, YoutubeDislikes* are highly correlated? Note: Draw pair plot or heatmap.

Use the SAheart Dataset to Answer Questions 13 to 20.

The dataset SAheart.data is taken from the link below:

http://www-stat.stanford.edu/~tibs/ElemStatLearn/datasets/SAheart.data

The dataset contains retrospective sample of males in a heart-disease high-risk region of the Western Cape, South Africa. There are roughly two controls per case of Coronary Heart Disease (CHD). Many of the CHD-positive men have undergone blood pressure reduction treatment and other programs to reduce their risk factors after their CHD event. In some cases, the measurements were made after these treatments. These data are taken from a larger dataset, described in Rousseauw et al. (1983), *South African Medical Journal*. It is a tab separated file (csv) and contains the following columns (source: *http://www-stat.stanford.edu)*

- *sbp* – Systolic blood pressure
- *tobacco* – Cumulative tobacco (kg)
- *ldl* – Low densiity lipoprotein cholesterol
- *adiposity*
- *famhist* – Family history of heart disease (Present, Absent)
- *typea* – Type-A behavior
- *obesity*
- *alcohol* – Current alcohol consumption
- *age* – Age at onset
- *chd* – Response, coronary heart disease

13. How many records are present in the dataset? Print the metadata information of the dataset.
14. Draw a bar plot to show the number of persons having CHD or not in comparison to they having family history of the disease or not.
15. Does age have any correlation with sbp? Choose appropriate plot to show the relationship.
16. Compare the distribution of tobacco consumption for persons having CHD and not having CHD. Can you interpret the effect of tobacco consumption on having coronary heart disease?
17. How are the parameters sbp, obesity, age and ldl correlated? Choose the right plot to show the relationships.
18. Derive a new column called *agegroup* from *age* column where persons falling in different age ranges are categorized as below.
(0–15): young
(15–35): adults
(35–55): mid
(55–): old
19. Find out the number of CHD cases in different age categories. Do a barplot and sort them in the order of age groups.
20. Draw a box plot to compare distributions of *ldl* for different age groups.

REFERENCES

1. U Dinesh Kumar (2017). *Business Analytics: The Science of Data-Driven Decision Making*, Wiley India Pvt. Ltd., India.
2. UC Irvine Machine Learning Repository: https://archive.ics.uci.edu/ml/index.php
3. The Elements of Statistical Learning (Stanford): https://web.stanford.edu/~hastie/ElemStatLearn/
4. Pandas Library: https://pandas.pydata.org/
5. Matplotlib Library: https://matplotlib.org/
6. Seabon Library: https://seaborn.pydata.org/
7. Rousseauw J, du Plessis J, Benade A, Jordaan P, Kotze J, and Ferreira J (1983). Coronary Risk Factor Screening in Three Rural Communities, *South African Medical Journal*, 64, pp. 430–436.