Plotting with pandas and seaborn

Now that we have a basic sense of how to load and handle data in a **pandas** DataFrame object, let's get started with making some simple plots from data. While there are several plotting libraries in Python (including **matplotlib**, **plotly**, and **seaborn**), in this chapter, we will mainly explore the **pandas** and **seaborn** libraries, which are extremely useful, popular, and easy to use.

Creating Simple Plots to Visualize a Distribution of Variables

matplotlib is a plotting library available in most Python distributions and is the foundation for several plotting packages, including the built-in plotting functionality of **pandas** and **seaborn**. **matplotlib** enables control of every single aspect of a figure and is known to be verbose. Both **seaborn** and **pandas** visualization functions are built on top of **matplotlib**. The built-in plotting tool of **pandas** .is a useful exploratory tool to generate figures that are not ready for primetime but useful to understand the dataset you are working with. **seaborn**, on the other hand, has APIs to draw a wide variety of aesthetically pleasing plots.

To illustrate certain key concepts and explore the **diamonds** dataset, we will start with two simple visualizations—histograms and bar plots.

Histograms

A histogram of a feature is a plot with the range of the feature on the *x*-axis and the count of data points with the feature in the corresponding range on the *y*-axis.

Plotting and Analysing a Histogram

We will create a histogram of the frequency of diamonds in the dataset with their respective **carat** specifications on the *x*-axis:

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. Plot a histogram using the **diamonds** dataset where **x axis = carat**:

   diamonds_df.hist(column='carat')

   The output is as follows:

```
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x00000216D76A8DD8>]],
      dtype=object)
```

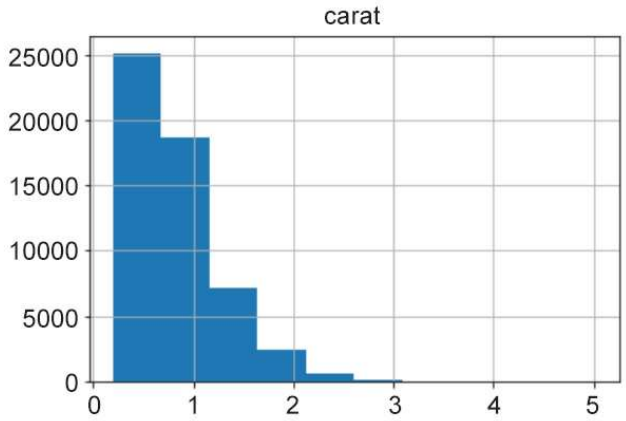

Figure 1.14: Histogram plot

The *y* axis in this plot denotes the number of diamonds in the dataset with the **carat** specification on the *x*-axis.

The **hist** function has a parameter called **bins**, which literally refers to the number of equally sized **bins** into which the data points are divided. By default, the bins parameter is set to **10** in **pandas**. We can change this to a different number, if we wish.

4. Change the **bins** parameter to **50**:

   diamonds_df.hist(column='carat', bins=50)

   The output is as follows:

```
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x00000216D79E7898>]],
      dtype=object)
```

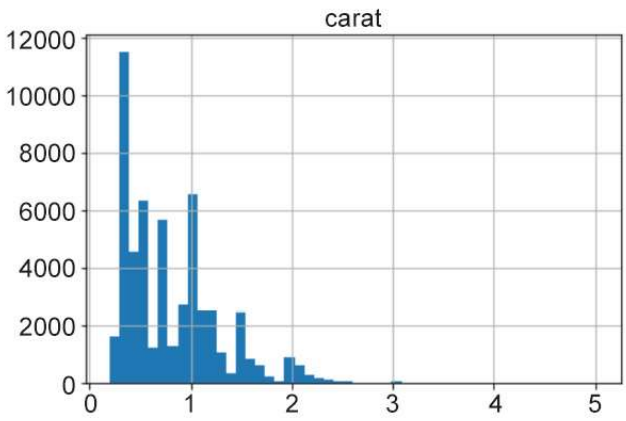

Figure 1.15: Histogram with bins = 50

This is a histogram with **50** bins. Notice how we can see a more fine-grained distribution as we increase the number of bins. It is helpful to test with multiple bin sizes to know the exact distribution of the feature. The range of **bin** sizes varies from **1**

(where all values are in the same bin) to the number of values (where each value of the feature is in one bin).

5. Now, let's look at the same function using **seaborn**:

   sns.distplot(diamonds_df.carat)

   The output is as follows:

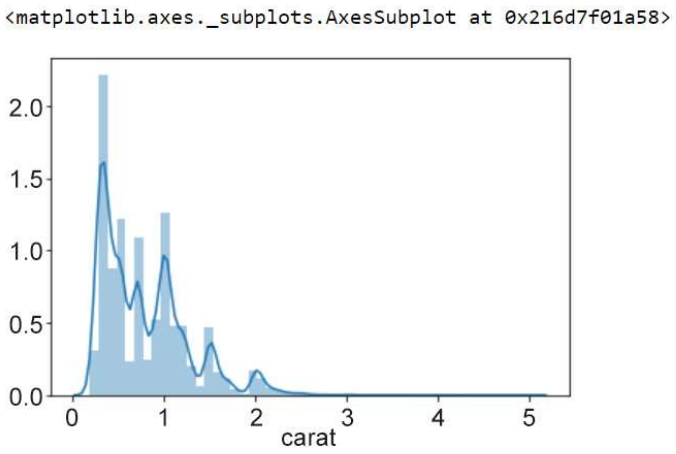

Figure 1.16: Histogram plot using seaborn

   There are two noticeable differences between the **pandas hist** function and **seaborn distplot**:

   - **pandas** sets the **bins** parameter to a default of **10**, but **seaborn** infers an appropriate bin size based on the statistical distribution of the dataset.
   - By default, the **distplot** function also includes a smoothed curve over the histogram, called a **kernel density estimation**.

     The **kernel density estimation** (**KDE**) is a non-parametric way to estimate the probability density function of a random variable. Usually, a KDE doesn't tell us anything more than what we can infer from the histogram itself. However, it is helpful when comparing multiple histograms on the same plot. If we want to remove the KDE and look at the histogram alone, we can use the **kde=False** parameter.

6. Change **kde=False** to remove the KDE:

   sns.distplot(diamonds_df.carat, kde=False)

The output is as follows:

```
<matplotlib.axes._subplots.AxesSubplot at 0x216d800e390>
```

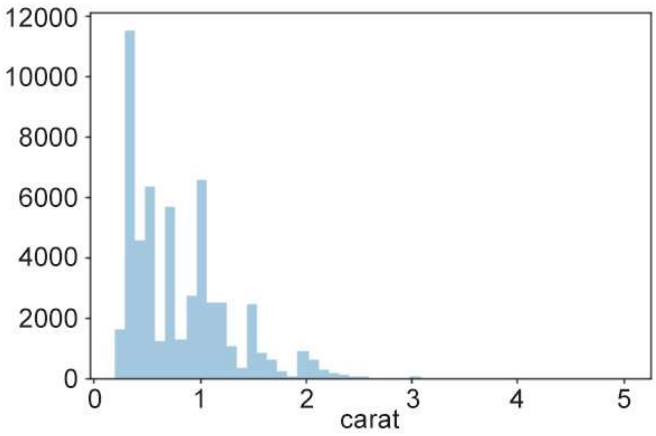

Figure 1.17: Histogram plot with KDE = false

Also note that the **bins** parameter seemed to render a more detailed plot when the bin size was increased from **10** to **50**. Now, let's try to increase it to 100.

7. Increase the **bins** size to **100**:

```
sns.distplot(diamonds_df.carat, kde=False, bins=100)
```

The output is as follows:

```
<matplotlib.axes._subplots.AxesSubplot at 0x216d80d35c0>
```

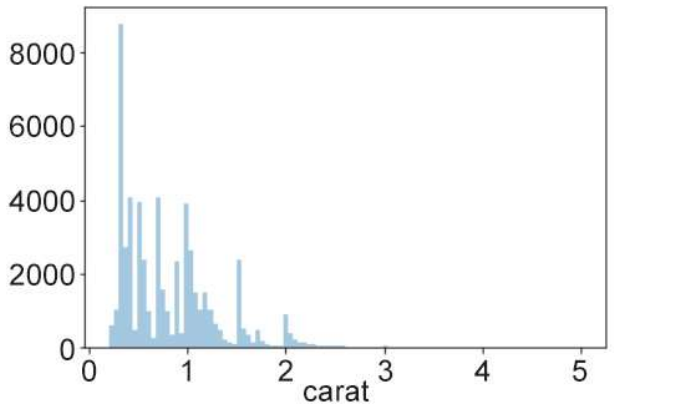

Figure 1.18: Histogram plot with increased bin size

The histogram with **100** bins shows a better visualization of the distribution of the variable—we see there are several peaks at specific carat values. Another observation is that most **carat** values are concentrated toward lower values and the **tail** is on the right—in other words, it is right-skewed.

A log transformation helps in identifying more trends. For instance, in the following graph, the *x*-axis shows log-transformed values of the **price** variable, and we see that

there are two peaks indicating two kinds of diamonds—one with a high price and another with a low price.

8. Use a log transformation on the histogram:

   import numpy as np

   sns.distplot(np.log(diamonds_df.price), kde=False)

   The output is as follows:

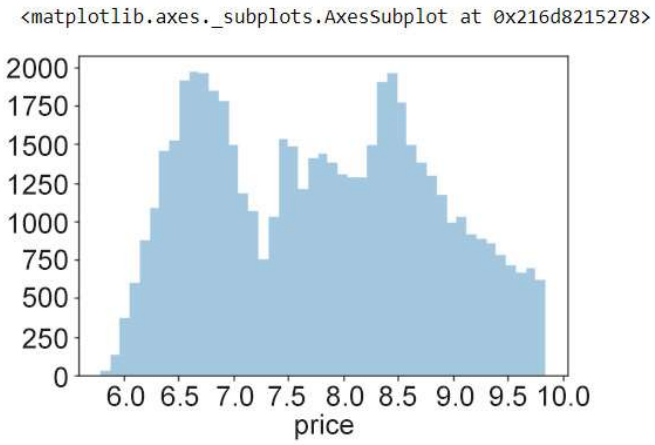

Figure 1.19: Histogram using a log transformation

That's pretty neat. Looking at the histogram, even a naive viewer immediately gets a picture of the distribution of the feature. Specifically, three observations are important in a histogram:

- Which feature values are more frequent in the dataset (in this case, there is a peak at around 6.8 and another peak between **8.5** and **9**—note that **log(price) = values**, in this case,
- How many *peaks* exist in the data (the peaks need to be further inspected for possible causes in the context of the data)
- Whether there are any outliers in the data

Bar Plots

In their simplest form, *bar plots* display counts of categorical variables. More broadly, bar plots are used to depict the relationship between a categorical variable and a numerical variable. Histograms, meanwhile, are plots that show the statistical distribution of a continuous numerical feature.

First, we shall present the counts of diamonds of each cut quality that exist in the data. Second, we shall look at the price associated with the different types of cut quality (**Ideal**, **Good**,

Premium, and so on) in the dataset and find out the mean price distribution. We will use both **pandas** and **seaborn** to get a sense of how to use the built-in plotting functions in both libraries.

Before generating the plots, let's look at the unique values in the **cut** and **clarity** columns, just to refresh our memory.

Creating a Bar Plot and Calculating the Mean Price Distribution

Here we'll learn how to create a table using the **pandas crosstab** function. We'll use a table to generate a bar plot. We'll then explore a bar plot generated using the **seaborn** library and calculate the mean price distribution. To do so, let's go through the following steps:

1. Import the necessary modules and dataset:

   ```
   import seaborn as sns
   import pandas as pd
   ```

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

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

3. Print the unique values of the **cut** column:

   ```
   diamonds_df.cut.unique()
   ```

   The output will be as follows:

   ```
   array(['Ideal', 'Premium', 'Good', 'Very Good', 'Fair'], dtype=object)
   ```

4. Print the unique values of the **clarity** column:

   ```
   diamonds_df.clarity.unique()
   ```

   The output will be as follows:

   ```
   array(['SI2', 'SI1', 'VS1', 'VS2', 'VVS2', 'VVS1', 'I1', 'IF'],
       dtype=object)
   ```

   Note

   unique() returns an array. There are five unique **cut** qualities and eight unique values in **clarity**. The number of unique values can be obtained using **nunique()** in **pandas**.

5. To obtain the counts of diamonds of each cut quality, we first create a table using the **pandas crosstab()** function:

   cut_count_table = pd.crosstab(index=diamonds_df['cut'],columns='count')

   cut_count_table

   The output will be as follows:

col_0	count
cut	
Fair	1610
Good	4906
Ideal	21551
Premium	13791
Very Good	12082

Figure 1.20: Table using the crosstab function

6. Pass these counts to another **pandas** function, **plot(kind='bar')**:

   cut_count_table.plot(kind='bar')

   The output will be as follows:

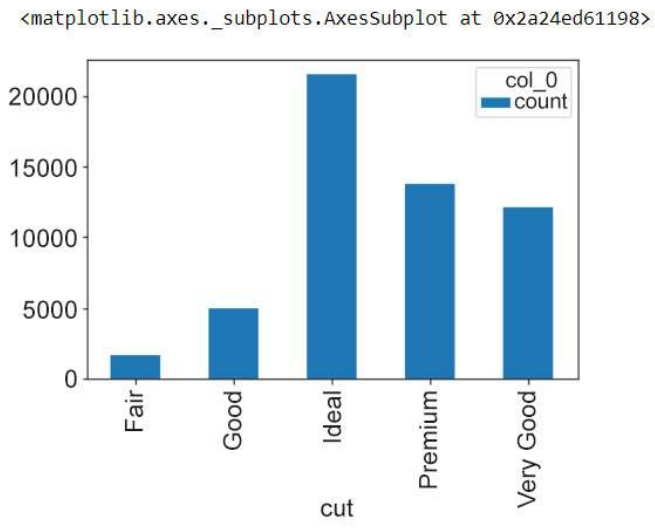

Figure 1.21: Bar plot using a pandas DataFrame

We see that most of the diamonds in the dataset are of the **Ideal** cut quality, followed by **Premium**, **Very Good**, **Good**, and **Fair**. Now, let's see how to generate the same plot using **seaborn**.

7. Generate the same bar plot using **seaborn**:

   sns.catplot("cut", data=diamonds_df, aspect=1.5, kind="count", color="b")

   The output will be as follows:

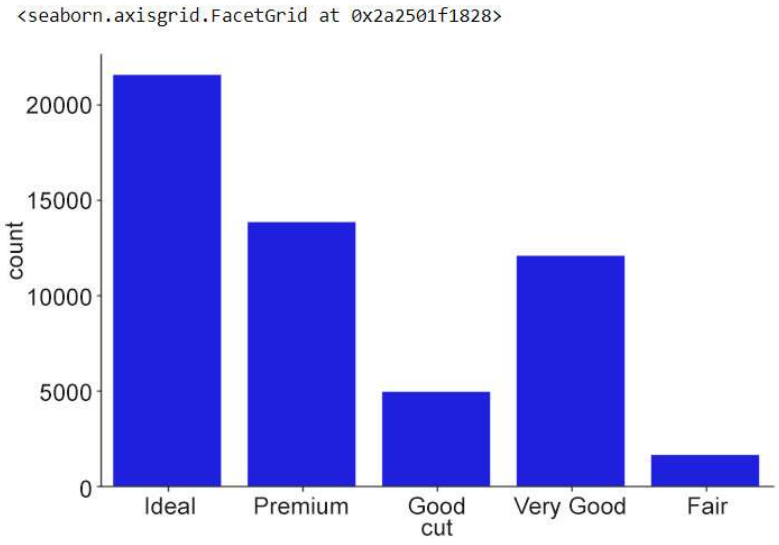

Figure 1.22: Bar plot using seaborn

   Notice how the **catplot()** function does not require us to create the intermediate count table (using **pd.crosstab()**), and reduces one step in the plotting process.

8. Next, here is how we obtain the mean price distribution of different cut qualities using **seaborn**:

   import seaborn as sns

   from numpy import median, mean

   sns.set(style="whitegrid")

   ax = sns.barplot(x="cut", y="price", data=diamonds_df,estimator=mean)

   The output will be as follows:

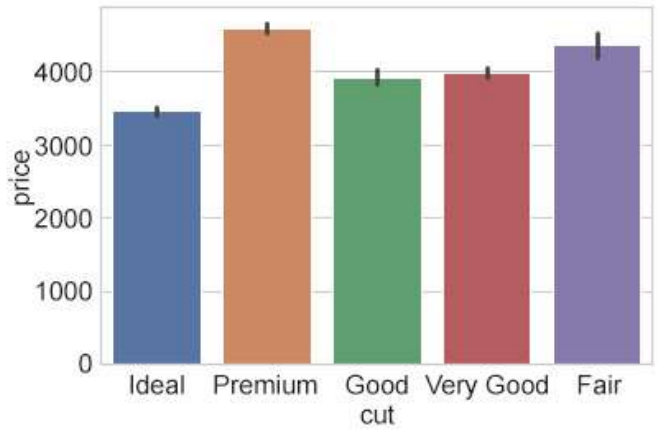

Figure 1.23: Bar plot with the mean price distribution

Here, the black lines (*error bars*) on the rectangles indicate the uncertainty (or spread of values) around the mean estimate. By default, this value is set to **95%** confidence. How do we change it? We use the **ci=68** parameter, for instance, to set it to **68%**. We can also plot the standard deviation in the prices using **ci=sd**.

9. Reorder the *x* axis bars using **order**:

   ax = sns.barplot(x="cut", y="price", data=diamonds_df, estimator=mean, ci=68, order=['Ideal','Good','Very Good','Fair','Premium'])

   The output will be as follows:

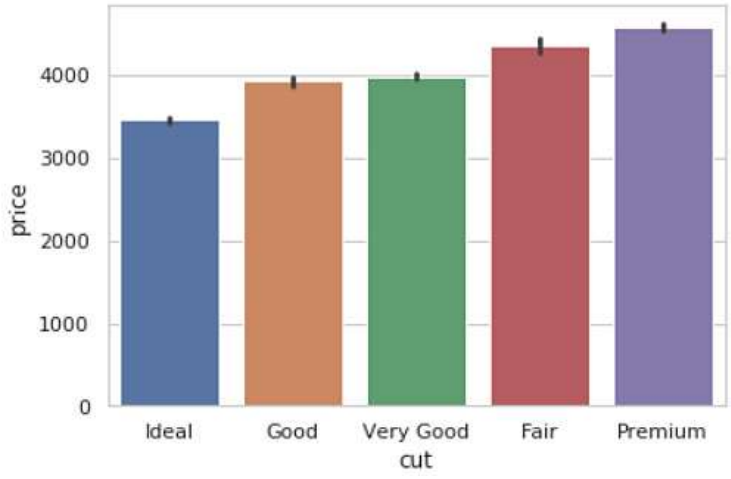

Figure 1.24: Bar plot with proper order

Grouped bar plots can be very useful for visualizing the variation of a particular feature within different groups. Now that you have looked into tweaking the plot parameters in a grouped bar plot, let's see how to generate a bar plot grouped by a specific feature.

Creating Bar Plots Grouped by a Specific Feature

We will use the **diamonds** dataset to generate the distribution of prices with respect to **color** for each **cut** quality. In *Creating a Bar Plot and Calculating the Mean Price Distribution*, we looked at the price distribution for diamonds of different cut qualities. Now, we would like to look at the variation in each color:

1. Import the necessary modules—in this case, only **seaborn**:

   #Import seaborn

   import seaborn as sns

2. Load the dataset:

   diamonds_df = sns.load_dataset('diamonds')

3. Use the **hue** parameter to plot nested groups:

   ax = sns.barplot(x="cut", y="price", hue='color', data=diamonds_df)

   The output is as follows:

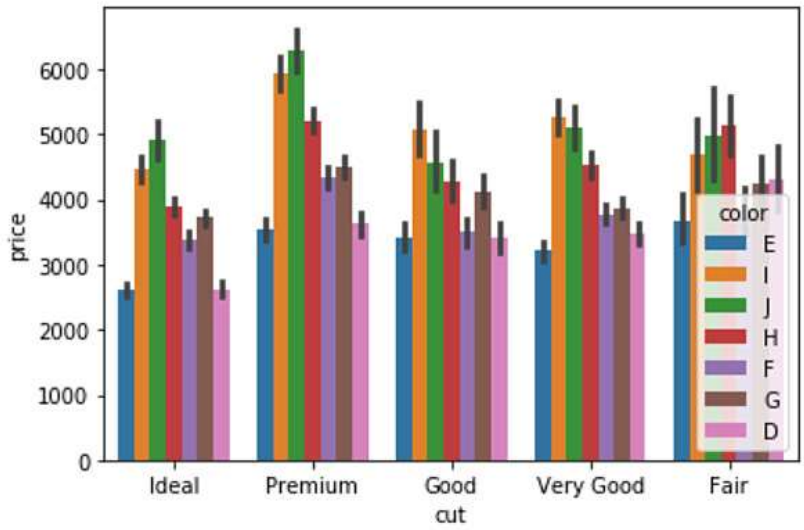

Figure 1.25: Grouped bar plot with legends

Here, we can observe that the price patterns for diamonds of different colours are similar for each cut quality. For instance, for **Ideal** diamonds, the price distribution of diamonds of different colours is the same as that for **Premium**, and other diamonds.

Tweaking Plot Parameters

Looking at the last figure in our previous section, we find that the legend is not appropriately placed. We can tweak the plot parameters to adjust the placements of the legends and the axis labels, as well as change the font-size and rotation of the tick labels.

Tweaking the Plot Parameters of a Grouped Bar Plot

Now, we'll tweak the plot parameters, for example, **hue**, of a grouped bar plot. We'll see how to place legends and axis labels in the right places and also explore the rotation feature:

1. Import the necessary modules—in this case, only **seaborn**:

   #Import seaborn

   import seaborn as sns

2. Load the dataset:

   diamonds_df = sns.load_dataset('diamonds')

3. Use the **hue** parameter to plot nested groups:

   ax = sns.barplot(x="cut", y="price", hue='color', data=diamonds_df)

   The output is as follows:

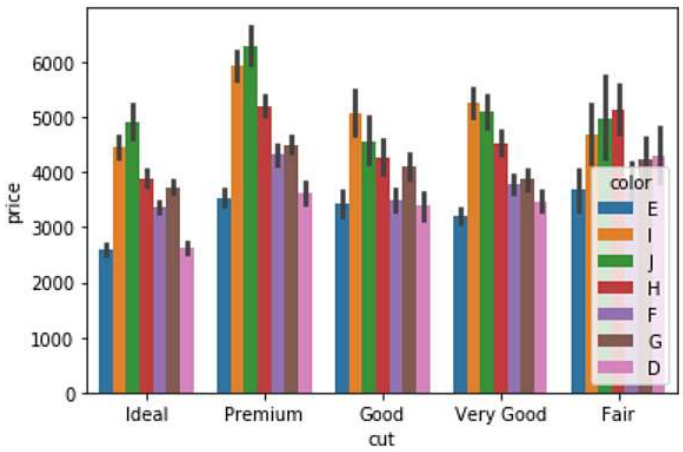

Figure 1.26: Nested bar plot with the hue parameter

4. Place the legend appropriately on the bar plot:

   ax = sns.barplot(x='cut', y='price', hue='color', data=diamonds_df)

   ax.legend(loc='upper right',ncol=4)

   The output is as follows:

```
<matplotlib.legend.Legend at 0x1d1d7320400>
```

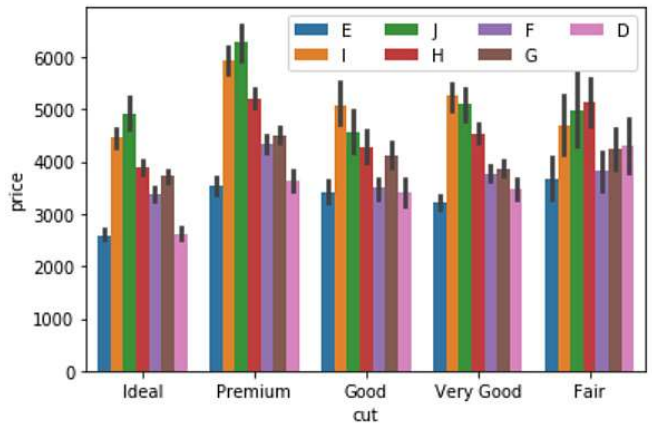

Figure 1.27: Grouped bar plot with legends placed appropriately

In the preceding **ax.legend()** call, the **ncol** parameter denotes the number of columns into which values in the legend are to be organized, and the **loc** parameter specifies the location of the legend and can take any one of eight values (*upper left*, *lower center*, and so on).

5. To modify the axis labels on the *x* axis and *y* axis, input the following code:

```
ax = sns.barplot(x='cut', y='price', hue='color', data=diamonds_df)

ax.legend(loc='upper right', ncol=4)

ax.set_xlabel('Cut', fontdict={'fontsize' : 15})

ax.set_ylabel('Price', fontdict={'fontsize' : 15})
```

The output is as follows:

```
Text(0, 0.5, 'Price')
```

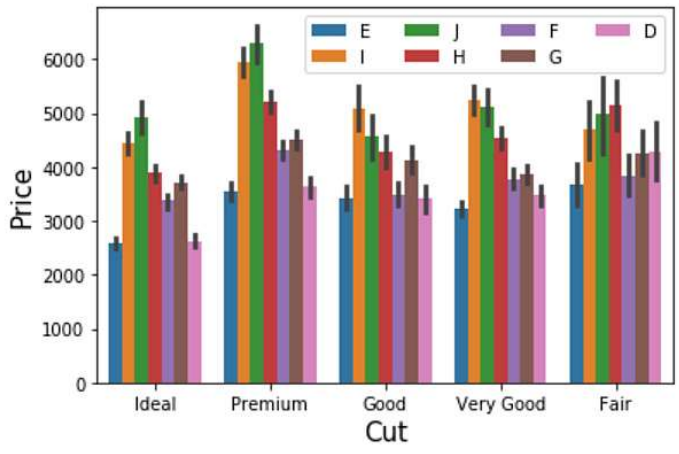

Figure 1.28: Grouped bar plot with modified labels

6. Similarly, use this to modify the font-size and rotation of the *x* axis of the tick labels:

```
ax = sns.barplot(x='cut', y='price', hue='color', data=diamonds_df)

ax.legend(loc='upper right',ncol=4)

# set fontsize and rotation of x-axis tick labels

ax.set_xticklabels(ax.get_xticklabels(), fontsize=13, rotation=30)
```

The output is as follows:

```
[Text(0, 0, 'Ideal'),
 Text(0, 0, 'Premium'),
 Text(0, 0, 'Good'),
 Text(0, 0, 'Very Good'),
 Text(0, 0, 'Fair')]
```

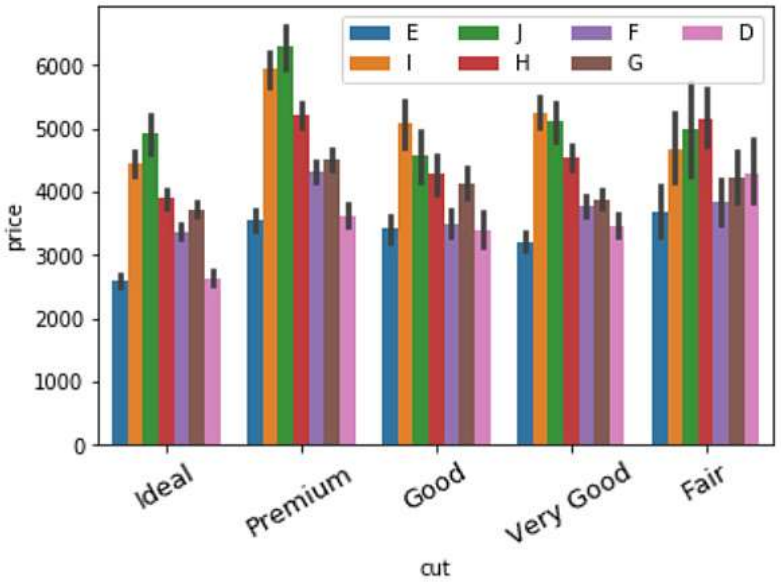

Figure 1.29: Grouped bar plot with the rotation feature of the labels

The *rotation feature* is particularly useful when the tick labels are long and crowd up together on the *x* axis.

Annotations

Another useful feature to have in plots is the annotation feature. In the following exercise, we'll make a simple bar plot more informative by adding some annotations.Suppose we want to add more information to the plot about *ideally* cut diamonds.

Annotating a Bar Plot

We will annotate a bar plot, generated using the **catplot** function of **seaborn**, using a note right above the plot. Let's see how:

1. Import the necessary modules:

   import matplotlib.pyplot as plt

   import seaborn as sns

2. Load the **diamonds** dataset:

   diamonds_df = sns.load_dataset('diamonds')

3. Generate a bar plot using **catplot** function of the **seaborn** library:

   ax = sns.catplot("cut", data=diamonds_df, aspect=1.5, kind="count", color="b")

   The output is as follows:

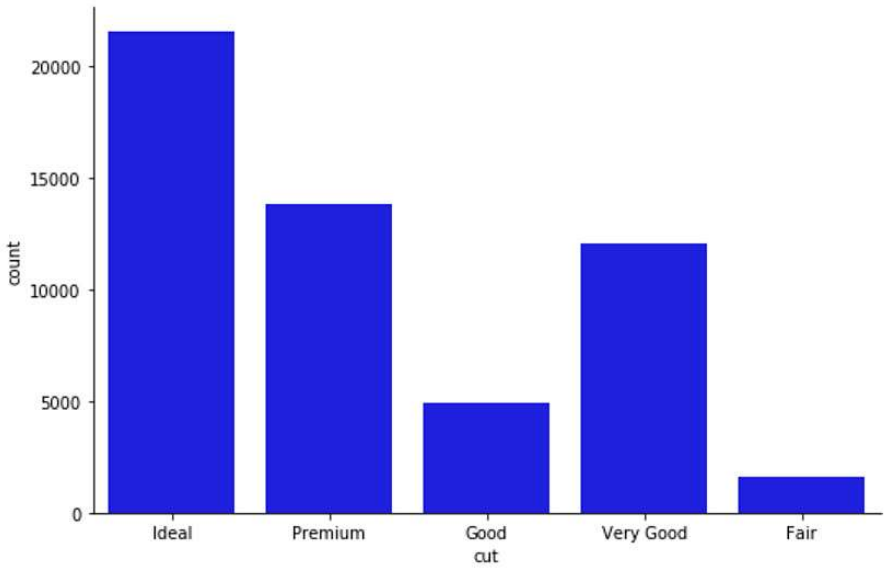

Figure 1.30: Bar plot with seaborn's catplot function

4. Annotate the column belonging to the **Ideal** category:

   # get records in the DataFrame corresponding to ideal cut

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

5. Find the location of the *x* coordinate where the annotation has to be placed:

   # get the location of x coordinate where the annotation has to be placed

   x = ideal_group.index.tolist()[0]

6. Find the location of the *y* coordinate where the annotation has to be placed:

   # get the location of y coordinate where the annotation has to be placed

   y = len(ideal_group)

7. Print the location of the *x* and *y* co-ordinates:

   print(x)

   print(y)

   The output is:

   0

   21551

8. Annotate the plot with a note:

   # annotate the plot with any note or extra information

   sns.catplot("cut", data=diamonds_df, aspect=1.5, kind="count", color="b")

   plt.annotate('excellent polish and symmetry ratings;\nreflects almost all the light that enters it', xy=(x,y), xytext=(x+0.3, y+2000), arrowprops=dict(facecolor='red'))

   The output is as follows:

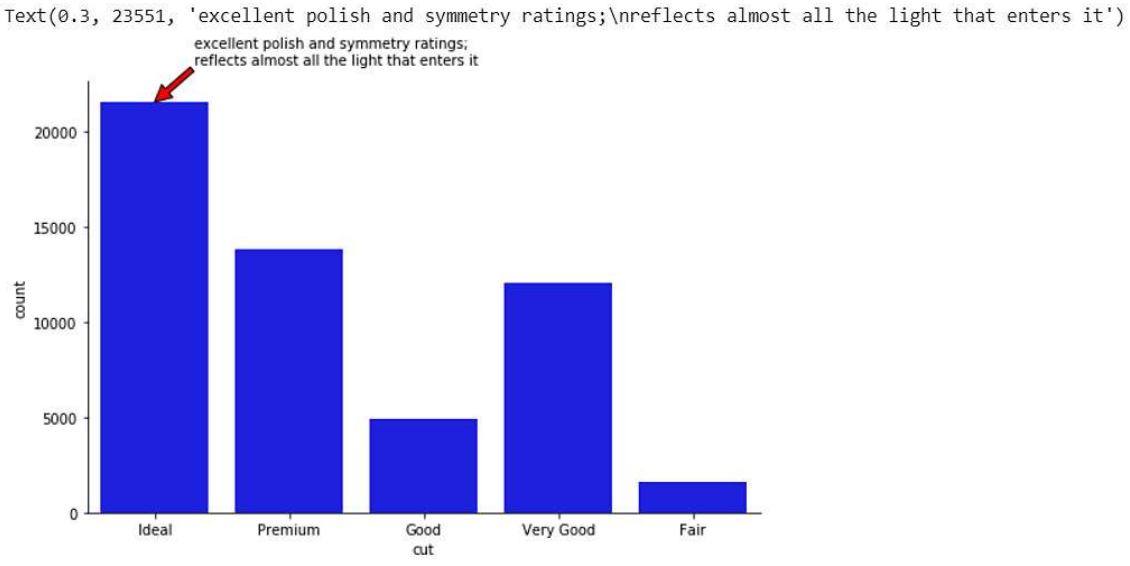

Figure 1.31: Annotated bar plot

Now, there seem to be a lot of parameters in the **annotate** function, but worry not! Matplotlib's https://matplotlib.org/3.1.0/api/_as_gen/matplotlib.pyplot.annotate.html official documentation covers all the details. For instance, the **xy** parameter denotes the point (**x,y**) on the figure to annotate. **xytext** denotes the position (*x*,*y*) to place the text at. If **None**, it defaults to **xy**. Note that we added an offset of **.3** for *x* and **2000** for *y* (since *y* is close to **20,000**) for the sake of readability of the text. The colour of the arrow is specified using the **arrowprops** parameter in the **annotate** function.

There are several other bells and whistles associated with visualization libraries in Python, some of which we will see as we progress in the book. At this stage, we will go through a chapter activity to revise the concepts in this chapter.

So far, we have seen how to generate two simple plots using **seaborn** and **pandas**—histograms and bar plots:

- **Histograms**: Histograms are useful for understanding the statistical distribution of a numerical feature in a given dataset. They can be generated using the **hist()** function in **pandas** and **distplot()** in **seaborn**.
- **Bar plots**: Bar plots are useful for gaining insight into the values taken by a categorical feature in a given dataset. They can be generated using the **plot(kind='bar')** function in **pandas** and the **catplot(kind='count')**, and **barplot()** functions in **seaborn**.

With the help of various considerations arising in the process of plotting these two types of visualizations, we presented some basic concepts in data visualization:

- Formatting legends to present labels for different elements in the plot with **loc** and other parameters in the **legend** function
- Changing the properties of tick labels, such as font-size, and rotation, with parameters in the **set_xticklabels()** and **set_yticklabels()** functions
- Adding annotations for additional information with the **annotate()** function

Summary

In this tutorial, we covered the basics of handling **pandas** DataFrames to format them as inputs for different visualization functions in libraries such as **pandas** , **seaborn** and more, and we covered some essential concepts in generating and modifying plots to create pleasing figures.

The **pandas** library contains functions such as **read_csv()**, **read_excel()**, and **read_json()** to read structured text data files. Functions such as **describe()** and **info()** are useful to get information on the summary statistics and memory usage of the features in a DataFrame. Other important operations on **pandas** DataFrames include subletting based on user-specified conditions/constraints, adding new columns to a DataFrame, transforming existing columns with built-in Python functions as well as user-defined functions, deleting specific columns in a DataFrame, and writing a modified DataFrame to a file on the local system.

Once equipped with knowledge of these common operations on **pandas** DataFrames, we went over the basics of visualization and learned how to refine the visual appeal of the plots. We illustrated these concepts with the plotting of histograms and bar plots. Specifically, we learned about different ways of presenting labels and legends, changing the properties of tick labels, and adding annotations.