CHAPTER 4 BSD2333

April 20, 2022

0.1 4.1 Subsetting, Filtering, and Grouping

In this exercise, we will read and examine an Excel file called Sample- Superstore.xls and will check all the columns to check if they are useful for analysis. We'll use the drop method to delete the columns that are unnecessary from the .xls file. Then, we'll use the shape function to check the number of rows and columns in the dataset.

Exercise 4.01: Examining the Superstore Sales Data in an Excel File

1. To read an Excel file into pandas, you will need a small package called xlrd to be installed on your system. Use the following code to install the xlrd package:

[ ]:
```
!pip install xlrd
```

2. Read the Excel file from GitHub into a pandas DataFrame using the read_excel method in pandas:

[ ]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
df = pd.read_excel("Sample - Superstore.xls")
df.head()
```

On examining the file, we can see that the first column, called Row ID, is not very useful because we already have a row index on the far left. This is a common occurrence in pandas and can be

3. Drop this column altogether from the DataFrame by using the drop method:

[ ]:
```
df.drop('Row ID',axis=1,inplace=True)
df.head()
```

4. Check the number of rows and columns in the newly created dataset. We will use the shape function here:

[ ]:
```
df.shape
```

In this exercise, we can see that the dataset has 9,994 rows and 20 columns. We have now seen that a simple way to remove unwanted columns such as a row count is simple with pandas. Think about how hard this would be if, instead of pandas, we used a list of dictionaries? We would have to write a loop to remove the rowid element from each dictionary in the list. pandas makes this functionality simple and easy.

Subsetting involves the extraction of partial data based on specific columns and rows, as per business needs. Let's pretend we are creating a report on our customers at the superstore. Suppose we are interested only in the following information from this dataset: Customer ID, Customer Name, City, Postal Code, and Sales. For demonstration purposes, let's assume that we are only interested in 5 records – rows 5-9. We can subset the DataFrame to extract only this much information using a single line of Python code.

We can use the loc method to index the Sample Superstore dataset by the names of the columns and the indexes of the rows, as shown in the following code:

```
[ ]: df_subset = df.loc[
     [i for i in range(5,10)],
     ['Customer ID','Customer Name','City','Postal Code','Sales']]
     df_subset
```

We need to pass on two arguments to the loc method – one for indicating the rows, and another for indicating the columns. When passing more than one value, you must pass them as a list for a row or column.

For the rows, we have to pass a list, that is, [5,6,7,8,9], but instead of writing that explicitly, we use a list comprehension, that is, [i for i in range(5,10)].

Because the columns we are interested in are not continuous and we cannot just put in a continuous range, we need to pass on a list containing the specific names. So, the second argument is just a simple list with specific column names. The dataset shows the fundamental concepts of the process of subsetting a DataFrame based on business requirements.

An Example Use Case – Determining Statistics on Sales and Profit Let's take a look at a typical use case of subsetting. Suppose we want to calculate descriptive statistics (mean, median, standard deviation, and so on) of records 100-199 for sales and profit in the SuperStore dataset. The following code shows how subsetting helps us achieve that:

```
[ ]: df_subset = df.loc[[i for i in range(100,199)],['Sales','Profit']]
     df_subset.describe()
```

We simply extract records 100-199 and run the describe function on them because we don't want to process all the data. For this particular business question, we are only interested in sales and profit numbers, and therefore we should not take the easy route and run a describe function on all the data. For a dataset that's being used in machine learning analysis, the number of rows and columns could often be in the millions, and we don't want to compute anything that is not asked for in the data wrangling task. We always aim to subset the exact data that needs to be processed and run statistical or plotting functions on that partial data. One of the most intuitive ways to try and understand the data is through charting. This can be a critical component of data wrangling.

To better understand sales and profit, let's create a box plot of the data using matplotlib:

```
[ ]: import matplotlib as plt
     boxplot = df_subset.boxplot()
```

As we can see from the preceding box plot, there are some outliers for profit. Now, they could be normal outliers, or they could be NaN values. At this point, we can't speculate, but this could cause some further analysis to see how we want to treat those outliers in profit. In some cases, outliers are fine, but for some predictive modeling techniques such as regression, outliers can have unwanted effects.

Before continuing further with filtering methods, let's take a quick detour and explore a super useful function called unique. As its name suggests, this function is used to scan through the data quickly and extract only the unique values in a column or row.

Exercise 4.02: The unique Function In the superstore sales data, you will notice that there are columns such as Country, State, and City. A natural question will be to ask how many countries/states/cities are present in the dataset. In this exercise, we'll use the unique function to find the number of unique countries/states/cities in the dataset. Let's go through the following steps:

1. Import the necessary libraries and read the file from GitHub by using the read_excel method in pandas into a DataFrame:

```
[ ]: import numpy as np
     import pandas as pd
     import matplotlib.pyplot as plt
     df = pd.read_excel("Sample - Superstore.xls")
```

2. Extract countries/states/cities for which the information is in the database, with one simple line of code, as follows:

```
[ ]: df['State'].unique()
```

You will see a list of all the states whose data is present in the dataset.

3. Use the nunique method to count the number of unique values in the State column, like so:

```
[ ]: df['State'].nunique()
```

This returns 49 for this dataset. So, one out of 50 states in the US does not appear in this dataset. Therefore, we can conclude that there's one repetition in the State column.

Similarly, if we run this function on the Country column, we get an array with only one element, United States. Immediately, we can see that we don't need to keep the country column at all because there is no useful information in that column, except that all the entries are the same. This is how a simple function helped us to decide about dropping a column altogether – that is, removing 9,994 pieces of unnecessary data.

Conditional Selection and Boolean Filtering Often, we don't want to process the whole dataset and would like to select only a partial dataset whose contents satisfy a particular condition. This is probably the most common use case of any data wrangling task. In the context of our superstore sales dataset, think of these common questions that may arise from the daily activities of the business analytics team:

- What are the average sales and profit figures in California?
- Which states have the highest and lowest total sales?
- What consumer segment has the most variance in sales/profit?

- Among the top five states in sales, which shipping mode and product category are the most popular choices?

Countless examples can be given where the business analytics team or the executive management wants to glean insight from a particular subset of data that meets certain criteria.

If you have any prior experience with SQL, you will know that these kinds of questions require fairly complex SQL query writing. Remember the WHERE clause? We will show you how to use conditional subsetting and boolean filtering to answer such questions.

First, we need to understand the critical concept of boolean indexing. This process essentially accepts a conditional expression as an argument and returns a dataset of booleans in which the TRUE value appears in places where the condition was satisfied. A simple example is shown in the following code. For demonstration purposes, we're subsetting a small dataset of 10 records and 3 columns:

```
[ ]: df_subset = df.loc[[i for i in range (10)],\
     ['Ship Mode','State','Sales']]
     df_subset
```

Now, if we just want to know the records with sales higher than $100, then we can write the following:

```
[ ]: df_subset['Sales'] > 100
```

Let's take a look at the True and False entries in the Sales column. The values in the Ship Mode and State columns were not impacted by this code because the comparison was with a numerical quantity, and the only numeric column in the original DataFrame was Sales.

Now, let's see what happens if we pass this boolean DataFrame as an index to the original DataFrame:

```
[ ]: df_subset[df_subset['Sales']>100]
```

We are not limited to conditional expressions involving numeric quantities only. Let's try to extract high sales values (>$100) for entries that do not involve California.

We can write the following code to accomplish this:

```
[ ]: df_subset[(df_subset['State']!='California') \
     & (df_subset['Sales']>100)]
```

Note the use of a conditional involving string. In this expression, we are joining two conditionals by an & operator. Both conditions must be wrapped inside parentheses. The first conditional expression simply matches the entries in the State column to the California string and assigns TRUE/FALSE accordingly. The second conditional is the same as before. Together, joined by the & operator, they extract only those rows for which State is not California and Sales is > $100.

Exercise 4.03: Setting and Resetting the Index In this exercise, we will create a pandas DataFrame and set and reset the index. We'll also add a new column and set it as the new index of this DataFrame. To do so, let's go through the following steps:

1. Import the numpy library:

```
[ ]: import numpy as np
```

2. Create the matrix_data, row_labels, and column_headings functions using the following commands:

```
[ ]: matrix_data = np.matrix('22,66,140;42,70,148;\
     30,62,125;35,68,160;25,62,152')
     row_labels = ['A','B','C','D','E']
     column_headings = ['Age', 'Height', 'Weight']
```

3. Import the pandas library and then create a DataFrame using the matrix_ data, row_labels, and column_headings functions:

```
[ ]: import pandas as pd
     df1 = pd.DataFrame(data=matrix_data,\
     index=row_labels,\
     columns=column_headings)
     print("\nThe DataFrame\n",'-'*25, sep='')
     df1
```

4. Reset the index, as follows:

```
[ ]: print("\nAfter resetting index\n",'-'*35, sep='')
     df1.reset_index()
```

5. Reset the index with drop set to True, as follows:

```
[ ]: print("\nAfter resetting index with 'drop' option TRUE\n",\
     '-'*45, sep='')
     df1.reset_index(drop=True)
```

6. Add a new column using the following command:

```
[ ]: print("\nAdding a new column 'Profession'\n",\
     '-'*45, sep='')
     df1['Profession'] = "Student Teacher Engineer Doctor Nurse"\
     .split()
     df1
```

7. Now, set the Profession column as an index using the following code:

```
[ ]: print("\nSetting 'Profession' column as index\n",\
     '-'*45, sep='')
     df1.set_index('Profession')
```

As we can see, the new data was added at the end of the table.

Exercise 4.04: The GroupBy Method In this exercise, we're going to create a subset from a dataset. We will use the groupBy object to filter the dataset and calculate the mean of that filtered dataset. To do so, let's go through the following steps:

1. Import the necessary Python modules and read the Excel file from GitHub by using the read_excel method in pandas:

```
[ ]: import numpy as np
     import pandas as pd
     import matplotlib.pyplot as plt
     df = pd.read_excel("Sample - Superstore.xls")
     df.head()
```

2. Create a 10-record subset using the following command:

```
[ ]: df_subset = df.loc[[i for i in range (10)],['Ship Mode','State','Sales']]
     df_subset
```

3. Create a pandas DataFrame using the groupby method, as follows:

```
[ ]: byState = df_subset.groupby('State')
     byState
```

4. Calculate the mean sales figure by State by using the following command:

```
[ ]: print("\nGrouping by 'State' column and listing mean sales\n",\
     '-'*50, sep='')
     byState.mean()
```

5. Calculate the total sales figure by State by using the following command:

```
[ ]: print("\nGrouping by 'State' column and listing total "\
     "sum of sales\n", '-'*50, sep='')
     byState.sum()
```

6. Subset that DataFrame for a particular state and show the statistics:

```
[ ]: pd.DataFrame(byState.describe().loc['California'])
```

7. Perform a similar summarization by using the Ship Mode attribute:

```
[ ]: df_subset.groupby('Ship Mode').describe().loc[['Second Class','Standard Class']]
```

8. Display the complete summary statistics of sales by every city in each state – all with two lines of code – by using the following command:

```
[ ]: byStateCity=df.groupby(['State','City'])
     byStateCity.describe()['Sales']
```

We now understand how to use pandas to group our dataset and then find aggregate values such as the mean sales return of our top employees. We also looked at how pandas will display descriptive statistics about our data for us. Both of these techniques can be used to perform analysis on our superstore data.

0.2 4.2 Concatenating, Merging, and Joining

Exercise 4.05: Concatenation in Datasets In this exercise, we will concatenate DataFrames along various axes (rows or columns).

This is a very useful operation as it allows you to grow a DataFrame as the new data comes in or new feature columns need to be inserted into the table. To do so, let's go through the following steps:

1. Read the Excel file from GitHub by using the read_excel method in pandas:

```
[ ]: import numpy as np
     import pandas as pd
     import matplotlib.pyplot as plt
     df = pd.read_excel("Sample - Superstore.xls")
     df.head()
```

2. Sample 4 records each to create three DataFrames at random from the original sales dataset we are working with:

```
[ ]: df_1 = df[['Customer Name','State','Sales','Profit']].sample(n=4)
     df_2 = df[['Customer Name','State','Sales','Profit']].sample(n=4)
     df_3 = df[['Customer Name','State','Sales','Profit']].sample(n=4)
```

3. Create a combined DataFrame with all the rows concatenated by using the following code:

```
[ ]: df_cat1 = pd.concat([df_1,df_2,df_3], axis=0)
     df_cat1
```

As you can see, concatenation will vertically combine multiple DataFrames. You can also try concatenating along the columns, although that does not make any practical sense for this particular example. However, pandas fills in the unavailable values with NaN for that operation.

4. Create a combined DataFrame with all the columns concatenated by using the following code:

```
[ ]: df_cat2 = pd.concat([df_1,df_2,df_3], axis=1)
     df_cat2
```

As we can observe, the cells in the dataset that do not contain any values are replaced with NaN values.

Merging by a Common Key Merging by a common key is an extremely common operation for data tables as it allows you to rationalize multiple sources of data in one master database – that is, if they have some common features/keys.

When joining and merging two DataFrames, we use two separate types: inner and outer {left|right}. Let's take a look at them:

• Inner: A combining method that uses a column or key to be compared on each dataset. Rows that share the same column or key will be present after the join.
• Outer: A way to combine datasets such as inner, but all data on the right or left (depending on which is chosen) is kept, and matching data from the opposite side is combined.

Exercise 4.06: Merging by a Common Key In this exercise, we'll create two DataFrames with the Customer Name common key from the Superstore dataset. Then, we will use the inner and outer joins to merge or combine these DataFrames. To do so, let's go through the following steps:

1. Import the necessary Python libraries and read the Excel file from GitHub by using the read_excel method in pandas:

```
[ ]: import numpy as np
     import pandas as pd
     import matplotlib.pyplot as plt
     df = pd.read_excel("Sample - Superstore.xls")
     df.head()
```

One DataFrame, df_1, will have shipping information associated with the customer name, and another table, df_2, will have the product information tabulated.

2. Create the df1 DataFrame with the Customer Name common key:

```
[ ]: df_1=df[['Ship Date','Ship Mode','Customer Name']][0:4]
     df_1
```

3. Create the second DataFrame, df2, with the Customer Name common key, as follows:

```
[ ]: df_2=df[['Customer Name','Product Name','Quantity']][0:4]
     df_2
```

4. Join these two tables with an inner join by using the following command:

```
[ ]: pd.merge(df_1,df_2,on='Customer Name',how='inner')
```

5. Drop the duplicates by using the following command:

```
[ ]: pd.merge(df_1,df_2,on='Customer Name',how='inner').drop_duplicates()
```

6. Extract another small table called df_3 to show the concept of an outer join:

```
[ ]: df_3=df[['Customer Name','Product Name','Quantity']][2:6]
     df_3
```

7. Perform an inner join on df_1 and df_3 by using the following command:

```
[ ]: pd.merge(df_1,df_3,on='Customer Name',how='inner').drop_duplicates()
```

8. Perform an outer join on df_1 and df_3 by using the following command:

```
[ ]: pd.merge(df_1,df_3,on='Customer Name',how='outer').drop_duplicates()
```

Notice how some NaN and NaT values are inserted automatically because no corresponding entries could be found for those records, as those are the entries with unique customer names from their respective tables. NaT represents a Not a Time object, as the objects in the Ship Date column are timestamped objects.

With this, we have gone over how to use the merge method to do inner and outer joins.

Exercise 4.07: The join Method In this exercise, we will create two DataFrames and perform the different kind of joins on these DataFrames.

To complete this exercise, perform the following steps:

1. Import the Python libraries and load the file from GitHub by using the read_ excel method in pandas:

```
[ ]: import numpy as np
     import pandas as pd
     import matplotlib.pyplot as plt
     df = pd.read_excel("Sample - Superstore.xls")
     df.head()
```

2. Create df1 with Customer Name as the index by using the following command:

```
[ ]: df_1=df[['Customer Name','Ship Date','Ship Mode']][0:4]
     df_1.set_index(['Customer Name'],inplace=True)
     df_1
```

3. Create df2 with Customer Name as the index by using the following command:

```
[ ]: df_2=df[['Customer Name','Product Name','Quantity']][2:6]
     df_2.set_index(['Customer Name'],inplace=True)
     df_2
```

4. Perform a left join on df_1 and df_2 by using the following command:

```
[ ]: df_1.join(df_2,how='left').drop_duplicates()
```

5. Perform a right join on df_1 and df_2 by using the following command:

```
[ ]: df_1.join(df_2,how='right').drop_duplicates()
```

6. Perform an inner join on df_1 and df_2 by using the following command:

```
[ ]: df_1.join(df_2,how='inner').drop_duplicates()
```

7. Perform an outer join on df_1 and df_2 by using the following command:

```
[ ]: df_1.join(df_2,how='outer').drop_duplicates()
```

0.3 4.3 Hierarchical Indexing

Exercise 4.08: Creating multiple index 1

```
[ ]: index = [('City A', 2018),('City A', 2019),
             ('City B', 2018), ('City B', 2019),
             ('City C', 2018), ('City C', 2019)]


     index
```

Second, provide the above multi-index data to Pandas pd.MultiIndex.from_tuples() function:

```
[ ]: index_pd = pd.MultiIndex.from_tuples(index)
     index_pd
```

Third, define the data, pop for our multi-index series, in the form of list:

```
[ ]: pop = [33871648, 37253956, 18976457, 19378102, 20851820, 25145561]
```

Fourth, use pd.Series constructor with data and index as arguments:

```
[ ]: pop = pd.Series(pop, index=index_pd)
     pop
```

In the above example, the first two columns of the Series representation show the multiple index values, while the third column shows the data.
Notice that some entries are missing in the first column: in this multi-index representation, any blank entry indicates the same value as the line above it

```
[ ]: pop['City A']
```

Population of all cities, for year 2018

```
[ ]: pop[:,2018]
```

Population of City A for year 2018

```
[ ]: pop['City A',2018]
```

Exercise 4.09: Creating multiple index 2

```
[ ]: import pandas as pd
     data = pd.Series([10, 20, 30, 40, 15, 25, 35, 25], index = [['a', 'a','a', 'a',␣
      ↪'b', 'b', 'b', 'b'], ['obj1', 'obj2', 'obj3', 'obj4', 'obj1','obj2', 'obj3',␣
      ↪'obj4']])
     data
```

There are two level of index here i.e. (a, b) and (obj1, ..., obj4). The index can be seen using 'index' command as shown below,

```
[ ]: data.index
```

Exercise 4.10: Partial indexing Choosing a particular index from a hierarchical indexing is known as partial indexing.

In the below code, index 'b' is extracted from the data,

```
[ ]: data['b']
```

Further, the data can be extracted based on inner level i.e. 'obj'. Below result shows the two available values for 'obj2' in the Series.

[ ]:
```
data[:, 'obj2']
```

Exercise 4.11: Unstack the data We saw the use of unstack operation in the Section Unstack. Unstack changes the row header to column header. Since the row index is changed to column index, therefore the Series will become the DataFrame in this case. Following are the some more example of unstacking the data,

[ ]:
```
# unstack based on first level i.e. a, b
# note that data row-labels are a and b
data.unstack(0)
```

[ ]:
```
# unstack based on second level i.e. 'obj'
data.unstack(1)
```

[ ]:
```
# by default innermost level is used for unstacking
d = data.unstack()
d
```

'stack()' operation converts the column index to row index again. In above code, DataFrame 'd' has 'obj' as column index, this can be converted into row index using 'stack' operation,

[ ]:
```
d.stack()
```

Exercise 4.12: Column indexing Remember that, the column indexing is possible for DataFrame only (not for Series), because column-indexing require two dimensional data. Let's create a new DataFrame as below for understanding the columns with multiple index,

[ ]:
```
import numpy as np
df = pd.DataFrame(np.arange(12).reshape(4, 3),index = [['a', 'a', 'b', 'b'],␣
 ↪['one', 'two', 'three', 'four']],columns = [['num1', 'num2', 'num3'],␣
 ↪['red', 'green', 'red']])
df
```

[ ]:
```
# display row index
df.index
```

[ ]:
```
# display column index
df.columns
```

Note that, in previous section, we used the numbers for stack and unstack operation e.g. unstack(0) etc. We can give name to index as well as below,

[ ]:
```
df.index.names=['key1', 'key2']
df.columns.names=['n', 'color']
df
```

Now, we can perform the partial indexing operations. In following code, various ways to access the data in a DataFrame are shown,

[ ]:
```
# accessing the column for num1
df['num1']  #  df.ix[:, 'num1']
```

Exercise 4.13: Swap and sort level We can swap the index level using 'swaplevel' command, which takes two level-numbers as input,

[ ]:
```
df.swaplevel('key1', 'key2')
```

Levels can be sorted using 'sort_index' command. In below code, data is sorted by 'key2' names i.e. key2 is arranged alphabetically,

[ ]:
```
df.sort_index(level='key2')
```

Exercise 4.14: Summary statistics by level We saw the example of groupby command in Section Groupby. Pandas provides some easier ways to perform those operations using 'level' shown below,

[ ]:
```
# add all rows with similar key1 name
df.sum(level = 'key1')
```

[ ]:
```
# add all the columns based on similar color
df.sum(level= 'color', axis=1)
```

0.4 4.4 Reshaping and Pivoting

Exercise 4.15: Reshaping by pivoting DataFrame objects Data is often stored in so-called "stacked" or "record" format:

[ ]:
```
import pandas._testing as tm

def unpivot(frame):
    N, K = frame.shape
    data = {
        "value": frame.to_numpy().ravel("F"),
        "variable": np.asarray(frame.columns).repeat(N),
        "date": np.tile(np.asarray(frame.index), K),
    }
    return pd.DataFrame(data, columns=["date", "variable", "value"])


df = unpivot(tm.makeTimeDataFrame(3))

df
```

To select out everything for variable A we could do:

[ ]:
```
filtered = df[df["variable"] == "A"]
filtered
```

But suppose we wish to do time series operations with the variables. A better representation would be where the columns are the unique variables and an index of dates identifies individual observations. To reshape the data into this form, we use the DataFrame.pivot() method (also implemented as a top level function pivot()):

```
[ ]: pivoted = df.pivot(index="date", columns="variable", values="value")
     pivoted
```

If the values argument is omitted, and the input DataFrame has more than one column of values which are not used as column or index inputs to pivot(), then the resulting "pivoted" DataFrame will have hierarchical columns whose topmost level indicates the respective value column:

```
[ ]: df["value2"] = df["value"] * 2
     pivoted = df.pivot(index="date", columns="variable")
     pivoted
```

You can then select subsets from the pivoted DataFrame:

```
[ ]: pivoted["value2"]
```

Note that this returns a view on the underlying data in the case where the data are homogeneously-typed.

Exercise 4.16: Reshaping by stacking and unstacking Let's take a prior example data set from the hierarchical indexing section:

```
[ ]: tuples = list(
         zip(
             *[
                 ["bar", "bar", "baz", "baz", "foo", "foo", "qux", "qux"],
                 ["one", "two", "one", "two", "one", "two", "one", "two"],
             ]
         )
     )


     index = pd.MultiIndex.from_tuples(tuples, names=["first", "second"])
     df = pd.DataFrame(np.random.randn(8, 2), index=index, columns=["A", "B"])
     df2 = df[:4]
     df2
```

The stack() function "compresses" a level in the DataFrame columns to produce either:

A Series, in the case of a simple column Index.

A DataFrame, in the case of a MultiIndex in the columns.

If the columns have a MultiIndex, you can choose which level to stack. The stacked level becomes the new lowest level in a MultiIndex on the columns:

```
[ ]: stacked = df2.stack()
     stacked
```

With a “stacked” DataFrame or Series (having a MultiIndex as the index), the inverse operation of stack() is unstack(), which by default unstacks the last level:

```
[ ]: stacked.unstack()
```

```
[ ]: stacked.unstack(1)
```

```
[ ]: stacked.unstack(0)
```

Notice that the stack() and unstack() methods implicitly sort the index levels involved. Hence a call to stack() and then unstack(), or vice versa, will result in a sorted copy of the original DataFrame or Series:

```
[ ]: index = pd.MultiIndex.from_product([[2, 1], ["a", "b"]])
     df = pd.DataFrame(np.random.randn(4), index=index, columns=["A"])
     df
```

```
[ ]: all(df.unstack().stack() == df.sort_index())
```

The above code will raise a TypeError if the call to sort_index() is removed.

0.5 4.5 DATA AGGREGATION

Python has several methods are available to perform aggregations on data. It is done using the pandas and numpy libraries. The data must be available or converted to a dataframe to apply the aggregation functions.

Exercise 4.17: Applying Aggregations on DataFrame

1. Let us create a DataFrame and apply aggregations on it.

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

     df = pd.DataFrame(np.random.randn(10, 4),
           index = pd.date_range('1/1/2000', periods=10),
           columns = ['A', 'B', 'C', 'D'])
     print(df)

     r = df.rolling(window=3,min_periods=1)
     print(r)
```

We can aggregate by passing a function to the entire DataFrame, or select a column via the standard get item method.

Apply Aggregation on a Whole Dataframe

[ ]:
```python
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(10, 4),
      index = pd.date_range('1/1/2000', periods=10),
      columns = ['A', 'B', 'C', 'D'])
print(df)

r = df.rolling(window=3,min_periods=1)
print(r.aggregate(np.sum))
```

Apply Aggregation on a Single Column of a Dataframe

[ ]:
```python
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(10, 4),
      index = pd.date_range('1/1/2000', periods=10),
      columns = ['A', 'B', 'C', 'D'])
print(df)
r = df.rolling(window=3,min_periods=1)
print(r['A'].aggregate(np.sum))
```

Apply Aggregation on Multiple Columns of a DataFrame

[ ]:
```python
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(10, 4),
      index = pd.date_range('1/1/2000', periods=10),
      columns = ['A', 'B', 'C', 'D'])
print(df)
r = df.rolling(window=3,min_periods=1)
print(r[['A','B']].aggregate(np.sum))
```

0.6 4.6 Pivot Tables and Cross-Tabulation

Use crosstab() to compute a cross-tabulation of two (or more) factors. By default crosstab() computes a frequency table of the factors unless an array of values and an aggregation function are passed.

Exercise 4.18: Cross Tabulation

[ ]:
```python
foo, bar, dull, shiny, one, two = "foo", "bar", "dull", "shiny", "one", "two"

a = np.array([foo, foo, bar, bar, foo, foo], dtype=object)

b = np.array([one, one, two, one, two, one], dtype=object)

c = np.array([dull, dull, shiny, dull, dull, shiny], dtype=object)
```

```
pd.crosstab(a, [b, c], rownames=["a"], colnames=["b", "c"])
```

If crosstab() receives only two Series, it will provide a frequency table.

```
[ ]: df = pd.DataFrame({"A": [1, 2, 2, 2, 2], "B": [3, 3, 4, 4, 4], "C": [1, 1, np.
      ↪nan, 1, 1]})
     df
```

```
[ ]: pd.crosstab(df["A"], df["B"])
```

crosstab() can also be implemented to Categorical data.

```
[ ]: foo = pd.Categorical(["a", "b"], categories=["a", "b", "c"])

     bar = pd.Categorical(["d", "e"], categories=["d", "e", "f"])

     pd.crosstab(foo, bar)
```

If you want to include all of data categories even if the actual data does not contain any instances of a particular category, you should set dropna=False.

For example:

```
[ ]: pd.crosstab(foo, bar, dropna=False)
```

Normalization Frequency tables can also be normalized to show percentages rather than counts using the normalize argument:

```
[ ]: pd.crosstab(df["A"], df["B"], normalize=True)
```

normalize can also normalize values within each row or within each column:

```
[ ]: pd.crosstab(df["A"], df["B"], normalize="columns")
```

crosstab() can also be passed a third Series and an aggregation function (aggfunc) that will be applied to the values of the third Series within each group defined by the first two Series:

```
[ ]: pd.crosstab(df["A"], df["B"], values=df["C"], aggfunc=np.sum)
```

Adding margins Finally, one can also add margins or normalize this output.

```
[ ]: pd.crosstab(df["A"], df["B"], values=df["C"], aggfunc=np.sum, normalize=True,␣
      ↪margins=True)
```

Exercise 4.19: Cross Tabulation on automobile data set

1. Let's get started by importing all the modules we need.

```
[ ]: import pandas as pd
     import seaborn as sns
     %matplotlib inline
```

Now we'll read in the automobile data set from the UCI Machine Learning Repository and make some label changes for clarity:

```
[ ]: # Define the headers since the data does not have any
     headers = ["symboling", "normalized_losses", "make", "fuel_type", "aspiration",
                "num_doors", "body_style", "drive_wheels", "engine_location",
                "wheel_base", "length", "width", "height", "curb_weight",
                "engine_type", "num_cylinders", "engine_size", "fuel_system",
                "bore", "stroke", "compression_ratio", "horsepower", "peak_rpm",
                "city_mpg", "highway_mpg", "price"]
```

```
[ ]: import numpy as np
     import pandas as pd
     import seaborn as sns
     import matplotlib.pyplot as plt
     df_raw = pd.read_csv("Automobile_imports-85.csv", header=None, names=headers,␣
      ↪na_values="?")
     df_raw.head()
```

```
[ ]: # Take a quick look at all the values in the data
     df_raw.describe()
```

```
[ ]: # Filter out the top 8 manufacturers
     models = ["toyota","nissan","mazda", "honda", "mitsubishi", "subaru",␣
      ↪"volkswagen", "volvo"]
```

```
[ ]: df = df_raw[df_raw.make.isin(models)].copy()
```

```
[ ]: df.head()
```

Basic Crosstab functions

```
[ ]: # Create a simple crosstab that counts the number of occurences of each␣
      ↪combination
     pd.crosstab(df.make, df.num_doors)
```

```
[ ]: # Add a subtotal
     pd.crosstab(df.make, df.num_doors, margins=True, margins_name="Total")
```

```
[ ]: # Another example, this time of make and body_style
     pd.crosstab(df.make, df.body_style)
```

```
[ ]: # Add custom names for the rows and columns?
     pd.crosstab(df.make, df.body_style, rownames=['Auto Manufacturer'],␣
      ↪colnames=['Body Style'])
```

Normalizing Results

[ ]:
```
# Convert the occurrences to percentages
pd.crosstab(df.make, df.body_style, normalize=True)
```

[ ]:
```
# Convert the occurrences to percentages for each row
pd.crosstab(df.make, df.body_style, normalize='index')
```

[ ]:
```
# Convert the occurrences to percentages for each column
pd.crosstab(df.make, df.body_style, normalize='columns')
```

[ ]:
```
# If you want to make the percentages a little easier to see, multiple all␣
 ↪values by 100
pd.crosstab(df.make, df.body_style, normalize='columns').mul(100).round(0)
```

Custom Aggregations and Grouping

[ ]:
```
# Perform aggregation functions - not just a simple count
pd.crosstab(df.make, df.body_style, values=df.curb_weight, aggfunc='mean').
 ↪round(0)
```

[ ]:
```
pd.crosstab(df.make, [df.body_style, df.drive_wheels], values=df.curb_weight,␣
 ↪aggfunc='mean').fillna('-')
```

[ ]:
```
# Crosstab supports grouping as well. In this case, group the columns
pd.crosstab(df.make, [df.body_style, df.drive_wheels])
```

[ ]:
```
# A more complex example showing the grouping of rows and columns
pd.crosstab([df.make, df.num_doors], [df.body_style, df.drive_wheels],
            rownames=['Auto Manufacturer', "Doors"],
            colnames=['Body Style', "Drive Type"],
            dropna=False)
```

[ ]:
```
# You can also use agg functions when grouping
pd.crosstab(df.make, [df.body_style, df.drive_wheels], values=df.curb_weight,␣
 ↪aggfunc='mean').fillna('-')
```

[ ]:
```
# You can also use margins when grouping
pd.crosstab(df.make, [df.body_style, df.drive_wheels],
            values=df.curb_weight, aggfunc='mean', margins=True,
            margins_name='Average').fillna('-').round(0)
```

Visualizing results with Seaborn

[ ]:
```
# Seaborn's heatmap can visualize the final results of the crosstab
sns.heatmap(pd.crosstab(df.drive_wheels, df.make))
```

[ ]:
```
# This is a more complex customization of a heatmap
sns.heatmap(pd.crosstab([df.make, df.num_doors], [df.body_style, df.
 ↪drive_wheels]), cmap="YlGnBu",
            annot=True, cbar=False)
```

[ ]:
```
sns.heatmap(pd.crosstab(df.make, df.body_style, values=df.curb_weight,␣
 ↪aggfunc='mean').round(0))
```

[ ]:
```
sns.heatmap(pd.crosstab(df.make, [df.body_style, df.drive_wheels],
            values=df.curb_weight, aggfunc='mean', margins=True,␣
 ↪margins_name='Average'),
           cmap="YlGnBu", annot=True, cbar=False, fmt='.0f')
```

[ ]:
```

```