case_1.2

April 3, 2020

1 How are trading volume and volatility related for energy stocks?

1.1 Introduction

Business Context. You are an analyst at a large bank focused on natural resource stock investments. Natural resources are vital for a variety of industries in our economy. Recently, your division has taken interest in the following stocks:

1. Dominion Energy Inc.
2. Exelon Corp.
3. NextEra Energy Inc.
4. Southern Co.
5. Duke Energy Corp.

These stocks are all part of the energy sector, an important but volatile sector of the stock market. While high volatility increases the chance of great gains, it also makes it more likely to have large losses, so risk must be carefully managed with high-volatility stocks.

Because your firm is quite large, there must be enough trading volume (average amount of shares transacted per day) so that it can easily transact in these stocks. Otherwise, this effect compounded with the stocks' naturally high volatility could make these too risky for the bank to invest in.

Business Problem. Given that both low trading volume and high volatility present risks to your investments, your team lead asks you to investigate the following: **"How is the volatility of energy stocks related to their average daily trading volume?"**

Analytical Context. The data you've been given is in the Comma Separated Value (CSV) format, and comprises price and trading volume data for the above stocks. This case begins with a brief overview of this data, after which you will: (1) learn how to use the Python library `pandas` to load the data; (2) use `pandas` transform this data into a form amenable for analysis; and finally (3) use `pandas` to analyze the above question and come to a conclusion. As you may have guessed, `pandas` is an enormously useful library for data analysis and manipulation.

1.2 Importing packages to aid in data analysis

External libraries (a.k.a. packages) are code bases that contain a variety of pre-written functions and tools. This allows you to perform a variety of complex tasks in Python without having to "reinvent the wheel" build everything from the ground up. We will use two core packages: `pandas` and `numpy`.

`pandas` is an external library that provides functionality for data analysis. Pandas specifically offers a variety of data structures and data manipulation methods that allow you to perform complex tasks with simple, one-line commands.

`numpy` is a package that we will use later in the case that offers numerous mathematical operations. Together, `pandas` and `numpy` allow you to create a data science workflow within Python.

Let's import both packages using the `import` keyword. We will rename `pandas` to `pd` and `numpy` to `np` using the `as` keyword. This allows us to use the short name abbreviation when we want to reference any function that is inside either package. The abbreviations we chose are standard across the data science industry and should be followed unless there is a very, very good reason not to.

```
[1]: # Import the Pandas package
     import pandas as pd

     # Import the NumPy package
     import numpy as np
```

Now that these packages are loaded into Python, we can use their contents. Let's first take a look at `pandas` as it has a variety of features we will use to load and analyze our stock data.

1.3 Fundamentals of pandas

At the core of the `pandas` library are two fundamental data structures/objects: 1. `Series` 2. `DataFrame`

A `Series` object stores single-column data along with an **index**. An index is just a way of "numbering" the `Series` object. For example, in this case study, the indices will be dates, while the single-column data may be stock prices or daily trading volume.

A `DataFrame` object is a two-dimensional tabular data structure with labeled axes. It is conceptually helpful to think of a DataFrame object as a collection of Series objects. Namely, think of each column in a DataFrame as a single Series object, where each of these Series objects shares a common index - the index of the DataFrame object.

Below is the syntax for creating a Series object, followed by the syntax for creating a DataFrame object. Note that DataFrame objects can also have a single-column – think of this as a DataFrame consisting of a single Series object:

```
[2]: # Create a simple Series object
     simple_series = pd.Series(index=[0,1,2,3], name='Volume',␣
      ↪data=[1000,2600,1524,98000])
     simple_series
```

```
[2]: 0     1000
     1     2600
     2     1524
     3    98000
     Name: Volume, dtype: int64
```

By changing `pd.Series` to `pd.DataFrame`, and adding a columns input list, a DataFrame object can be created:

```
[3]: # Create a simple DataFrame object
simple_df = pd.DataFrame(index=[0,1,2,3], columns=['Volume'],␣
 ↪data=[1000,2600,1524,98000])
simple_df
```

```
[3]:    Volume
     0    1000
     1    2600
     2    1524
     3   98000
```

DataFrame objects are more general compared to Series objects. Let's create a two column DataFrame object:

```
[4]: # Create another DataFrame object
another_df = pd.DataFrame(index=[0,1,2,3], columns=['Date','Volume'],␣
 ↪data=[[20190101,1000],[20190102,2600],[20190103,1524],[20190104,98000]])
another_df
```

```
[4]:        Date  Volume
     0  20190101    1000
     1  20190102    2600
     2  20190103    1524
     3  20190104   98000
```

Notice how a list of lists was used to specify the data in the `another_df` DataFrame. Each element of the list corresponds to a row in the DataFrame, so the list has 4 elements because there are 4 indices. Each element of the list of lists has 2 elements because the DataFrame has two columns.

1.4 Using pandas to analyze stock data

Recall that we have CSV files that include data for each of the following stocks:

1. Dominion Energy Inc. (Stock Symbol: D)
2. Exelon Corp. (Stock Symbol: EXC)
3. NextEra Energy Inc. (Stock Symbol: NEE)
4. Southern Co. (Stock Symbol: SO)
5. Duke Energy Corp. (Stock Symbol: DUK)

The available data for each stock includes:

1. **Date:** The day of the year
2. **Open:** The stock opening price of the day
3. **High:** The highest observed stock price of the day
4. **Low:** The lowest observed stock price of the day
5. **Close:** The stock closing price of the day

6. **Adj Close:** The adjusted stock closing price for the day (adjusted for splits and dividends)
7. **Volume:** The volume of the stock traded over the day

To get a better sense of the available data, let's first take a look at just the data for Dominion Energy, listed on the New York Stock Exchange under the symbol D. You are given a CSV file that contains the company's stock data, `D.csv`. Pandas allows easy loading of CSV files through the use of the method `pd.read_csv()`:

[5]:
```
# Load a file as a DataFrame and assign to df
df = pd.read_csv('D.csv')
```

The contents of the file `D.csv` are now stored in the DataFrame object `df`.

There are several common methods and attributes available to take a peek at the data and get a sense of it:

1. `DataFrame.head()` -> returns the column names and first 5 rows by default
2. `DataFrame.tail()` -> returns the column names and last 5 rows by default
3. `DataFrame.shape` -> returns (num_rows, num_columns)
4. `DataFrame.columns` -> returns index of columns
5. `DataFrame.index` -> returns index of rows

Using `df.head()` and `df.tail()` we can take a look at the data contents. Unless specified otherwise, Pandas Series and DataFrame objects have indicies starting at 0 and increase monotonically upward along the integers.

[6]:
```
# Look at the head of the DataFrame (i.e. the top rows of the DataFrame)
df.head()
```

[6]:
```
         Date       Open       High        Low      Close  Adj Close   Volume
0  2014-07-28  69.750000  71.059998  69.750000  70.879997  57.963978  1806400
1  2014-07-29  70.669998  70.980003  69.930000  69.930000  57.187099  2231100
2  2014-07-30  70.000000  70.660004  68.400002  68.970001  56.402020  2588900
3  2014-07-31  68.629997  68.849998  67.580002  67.639999  55.314388  3266900
4  2014-08-01  67.330002  68.410004  67.220001  67.589996  55.273487  2601800
```

[7]:
```
# Look at the tail of the DataFrame (i.e. the top rows of the DataFrame)
df.tail()
```

[7]:
```
            Date       Open       High        Low      Close  Adj Close  \
1254  2019-07-22  76.879997  76.930000  75.779999  76.260002  76.260002
1255  2019-07-23  76.099998  76.199997  75.269997  75.430000  75.430000
1256  2019-07-24  75.660004  75.720001  74.889999  75.180000  75.180000
1257  2019-07-25  75.150002  75.430000  74.610001  74.860001  74.860001
1258  2019-07-26  74.730003  75.349998  74.610001  75.150002  75.150002

       Volume
1254  2956500
1255  3175600
1256  3101900
```

```
1257    3417200
1258    3076500
```

Thus, we see there are 1259 data entries (each with 7 data points) for Dominion Energy. The shape of a DataFrame is accessed using the `shape` attribute:

```
[8]: # Determine the shape of the two-dimensional structure, that is (num_rows,␣
      ↪num_columns)
     df.shape
```

```
[8]: (1259, 7)
```

It's important to note that `DataFrame.columns` and `DataFrame.index` return an index object instead of a list. To cast an index to a list for further list manipulation, we use the `list()` method:

```
[9]: # List of the column names of the DataFrame
     list(df.columns)
```

```
[9]: ['Date', 'Open', 'High', 'Low', 'Close', 'Adj Close', 'Volume']
```

```
[10]: # List of the column names of the DataFrame
      list(df.index)[0:20] # only showing first 20 index values so reduce screen␣
       ↪output
```

```
[10]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

1.5 Creating additional variables relevant to stock volatility

Oftentimes, the data provided to you will not be sufficient to achieve your goal. You may have to add additional variables or data features to assist you. Recall that our original question concerned the relationship between stock trading volume and volatility. Therefore, our DataFrame must have features related to both of these quantities.

It can be helpful to think about adding columns to DataFrames as adding adjacent columns one-by-one in Excel. Here is an example of how to do it:

```
[11]: # Add a new column named "Symbol"
      df['Symbol'] = 'D'
      df.head()
```

```
[11]:          Date       Open       High        Low      Close  Adj Close   Volume  \
      0  2014-07-28  69.750000  71.059998  69.750000  70.879997  57.963978  1806400
      1  2014-07-29  70.669998  70.980003  69.930000  69.930000  57.187099  2231100
      2  2014-07-30  70.000000  70.660004  68.400002  68.970001  56.402020  2588900
      3  2014-07-31  68.629997  68.849998  67.580002  67.639999  55.314388  3266900
      4  2014-08-01  67.330002  68.410004  67.220001  67.589996  55.273487  2601800
```

```
  Symbol
0      D
1      D
2      D
3      D
4      D
```

```
[12]: # We can access a column by using [] brackets and the column name
      df['Volume'].head() # added .head() to suppress output
```

```
[12]: 0    1806400
      1    2231100
      2    2588900
      3    3266900
      4    2601800
      Name: Volume, dtype: int64
```

```
[13]: # Add a new column named "Volume_Millions", which is calculated from the Volume␣
       ↪column currently in df
      df['Volume_Millions'] = df['Volume'] / 1000000.0 # divide every row in␣
       ↪df['Volume'] by 1 million, store in new column
      df.head()
```

```
[13]:          Date       Open       High        Low      Close  Adj Close   Volume  \
      0  2014-07-28  69.750000  71.059998  69.750000  70.879997  57.963978  1806400
      1  2014-07-29  70.669998  70.980003  69.930000  69.930000  57.187099  2231100
      2  2014-07-30  70.000000  70.660004  68.400002  68.970001  56.402020  2588900
      3  2014-07-31  68.629997  68.849998  67.580002  67.639999  55.314388  3266900
      4  2014-08-01  67.330002  68.410004  67.220001  67.589996  55.273487  2601800

        Symbol  Volume_Millions
      0      D           1.8064
      1      D           2.2311
      2      D           2.5889
      3      D           3.2669
      4      D           2.6018
```

```
[14]: # Take a look at the updated DataFrame shape. Two new columns have been added.
      df.shape
```

```
[14]: (1259, 9)
```

As discussed, we need to have a feature in our DataFrame that is related to volatility. Because this currently does not exist, we must create it from the already available features. Recall that volatility is the standard deviation of daily returns over a period of time, so let's create a feature for daily returns:

```
[15]: df['VolStat'] = (df['High'] - df['Low']) / df['Open']
      df['Return'] = (df['Close'] / df['Open']) - 1.0
```

Here we see the power of `pandas`. We can simply perform mathematical operations on columns of DataFrames just as if the DataFrames were single variables themselves.

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

```
[16]:          Date       Open       High        Low      Close  Adj Close   Volume  \
      0  2014-07-28  69.750000  71.059998  69.750000  70.879997  57.963978  1806400
      1  2014-07-29  70.669998  70.980003  69.930000  69.930000  57.187099  2231100
      2  2014-07-30  70.000000  70.660004  68.400002  68.970001  56.402020  2588900
      3  2014-07-31  68.629997  68.849998  67.580002  67.639999  55.314388  3266900
      4  2014-08-01  67.330002  68.410004  67.220001  67.589996  55.273487  2601800

        Symbol  Volume_Millions   VolStat    Return
      0      D           1.8064  0.018781  0.016201
      1      D           2.2311  0.014858 -0.010471
      2      D           2.5889  0.032286 -0.014714
      3      D           3.2669  0.018505 -0.014425
      4      D           2.6018  0.017674  0.003861
```

Now we have features relevant to the original question, and can proceed to the analysis step. A common first step in data analysis is to learn about the distribution of the available data. We will do this next.

1.6 Learning about the data distribution through summary statistics

Let's aggregate summary statistics for the five energy sector companies under study. Fortunately, the DataFrame and Series objects offer a myriad of data summary statistics methods:

1. `min()`
2. `median()`
3. `mean()`
4. `max()`
5. `quantile()`

Below, each method is used on the `Volume_Millions` column. Notice how simple the functions are to apply to the DataFrame. Simply type the name of the DataFrame, followed by a `.` and then the method name you'd like to calculate. We've chosen to select a single column `Volume_Millions` from the DataFrame `df`, but you could have just as easily called these methods on the full DataFrame rather than a single column:

```
[17]: # Calculate the minimum of the Volume_Millions column
      df['Volume_Millions'].min()
```

```
[17]: 0.7384
```

```
[18]: # Calculate the median of the Volume_Millions column
      df['Volume_Millions'].median()
```

```
[18]: 2.6957
```

```
[19]: # Calculate the average of the Volume_Millions column
      df['Volume_Millions'].mean()
```

```
[19]: 3.0881293089753776
```

```
[20]: # Calculate the maximum of the Volume_Millions column
      df['Volume_Millions'].max()
```

```
[20]: 14.5874
```

We'd also like to explore the data distribution at a more granular level to see how the distribution looks beyond the simple summary statistics presented above. For this, we can use the `quantile()` method. The `quantile()` method will return the value which represents the given percentile of all the data under study (in this case, of the `Volume_Millions` data):

```
[21]: # Calculate the 25th percentile
      df['Volume_Millions'].quantile(0.25)
```

```
[21]: 2.0888
```

```
[22]: # Calculate the 75th percentile
      df['Volume_Millions'].quantile(0.75)
```

```
[22]: 3.61285
```

Is there a more efficient method to quickly compute all of these summary statistics? Yes. One incredibly useful method that combines these summary statistics and also adds a couple others is the `describe()` method:

```
[23]: df['Volume_Millions'].describe()
```

```
[23]: count    1259.000000
      mean        3.088129
      std         1.548809
      min         0.738400
      25%         2.088800
      50%         2.695700
      75%         3.612850
      max        14.587400
      Name: Volume_Millions, dtype: float64
```

From this distribution analysis of the daily trading volume, we can see that more than 14 million shares would be a very large trading day, whereas below 2 million shares would be a relatively small trading day.

1.6.1 Exercise 1:

Determine the 25th, 50th, and 75th percentile for the `Open`, `High`, `Low`, and `Close` columns of `df`.

Answer. One possible solution is indicated below:

[24]:
```
# One possible solution
print(df['Open'].describe())
print(df['High'].describe())
print(df['Low'].describe())
print(df['Close'].describe())
```

```
count    1259.000000
mean       73.253765
std         4.187696
min        61.790001
25%        70.220001
50%        73.180000
75%        76.560001
max        85.110001
Name: Open, dtype: float64
count    1259.000000
mean       73.780842
std         4.162946
min        62.840000
25%        70.829998
50%        73.690002
75%        76.954998
max        85.300003
Name: High, dtype: float64
count    1259.000000
mean       72.697911
std         4.177581
min        61.529999
25%        69.685001
50%        72.550003
75%        75.959999
max        83.900002
Name: Low, dtype: float64
count    1259.000000
mean       73.274940
std         4.182135
min        61.750000
25%        70.239998
50%        73.150002
75%        76.510002
```

```
max        84.910004
Name: Close, dtype: float64
```

1.7 Aggregating data from multiple companies

So far, we've only been looking at data from one of our five companies. Let's go ahead and combine all five CSV files to analyze the five companies together. This will also reduce the amount of programming work required since the code will be shared across the five companies.

One way to accomplish this aggregation task is to use the `pd.concat()` method from `pandas`. An input into this method may be a list of DataFrames that you'd like to concatenate. We will use a for loop to loop over each stock symbol, load the corresponding CSV file, and then append the result to a list which is later aggregated using `pd.concat()`. Let's take a look at how this is done.

[25]:
```
# Load five csv files into one dataframe
print("Defining stock symbols")
symbol_data_to_load = ['D','EXC','NEE','SO','DUK']
list_of_df = []

# Loop over all symbols
print(" --- Start loop over symbols --- ")
for i in symbol_data_to_load:
    print("Processing Symbol: " + i)
    temp_df = pd.read_csv(i+'.csv')
    temp_df['Volume_Millions'] = temp_df['Volume'] / 1000000.0
    temp_df['Symbol'] = i # ADD NEW COLUMN WITH SYMBOL NAME TO DISTINGUISH IN␣
 ↪FINAL DATAFRAME
    list_of_df.append(temp_df)

print(" --- Complete loop over symbols --- ")

# Combine into a single DataFrame by using concat
print("Aggregating Data")
agg_df = pd.concat(list_of_df, axis=0)

# Add salient statistics for this return and volatility analysis
print('Calculating Salient Features')
agg_df['VolStat'] = (agg_df['High'] - agg_df['Low']) / agg_df['Open']
agg_df['Return'] = (agg_df['Close'] / agg_df['Open']) - 1.0

print("agg_df DataFrame shape (rows, columns): ")
print(agg_df.shape)

print("Head of agg_df DataFrame: ")
agg_df.head()
```

```
Defining stock symbols
 --- Start loop over symbols ---
```

```
Processing Symbol: D
Processing Symbol: EXC
Processing Symbol: NEE
Processing Symbol: SO
Processing Symbol: DUK
 --- Complete loop over symbols ---
Aggregating Data
Calculating Salient Features
agg_df DataFrame shape (rows, columns):
(6295, 11)
Head of agg_df DataFrame:
```

[25]:
```
         Date       Open       High        Low      Close  Adj Close   Volume  \
0  2014-07-28  69.750000  71.059998  69.750000  70.879997  57.963978  1806400
1  2014-07-29  70.669998  70.980003  69.930000  69.930000  57.187099  2231100
2  2014-07-30  70.000000  70.660004  68.400002  68.970001  56.402020  2588900
3  2014-07-31  68.629997  68.849998  67.580002  67.639999  55.314388  3266900
4  2014-08-01  67.330002  68.410004  67.220001  67.589996  55.273487  2601800

   Volume_Millions Symbol   VolStat    Return
0           1.8064      D  0.018781  0.016201
1           2.2311      D  0.014858 -0.010471
2           2.5889      D  0.032286 -0.014714
3           3.2669      D  0.018505 -0.014425
4           2.6018      D  0.017674  0.003861
```

After the for loop, we've aggregated and added the relevant features we identified in the previous section. We then printed the head of the aggregated DataFrame to have a peek at the format of the data, and we've also printed the shape of the DataFrame. This is to sanity check that our final DataFrame is roughly what we expect. Notice the aggregated DataFrame has the same number of columns as the original single stock (D) data, however the number of rows have increased five-fold. This makes sense, because each additional symbol contains 1259 data entries, so five symbols leads to a total of `1259*5 = 6295` rows. So, this passes our sanity check.

Now, if we want to reverse this process and extract the data relevant to a single stock symbol from the aggregated DataFrame `agg_df`, we can do so using the `==` operator, which returns True when two objects contain the same value, and False otherwise:

[26]:
```python
symbol_DUK_df = agg_df[agg_df['Symbol'] == 'DUK']
symbol_DUK_df.head()
```

[26]:
```
         Date       Open       High        Low      Close  Adj Close   Volume  \
0  2014-07-28  73.309998  74.480003  73.230003  74.389999  59.266285  3281100
1  2014-07-29  74.400002  74.480003  73.760002  73.980003  58.939648  2236300
2  2014-07-30  74.029999  74.199997  72.580002  73.050003  58.198696  2782200
3  2014-07-31  72.610001  73.099998  72.059998  72.129997  57.465740  3249000
4  2014-08-01  72.239998  73.370003  72.150002  72.940002  58.111061  3960200
```

```
   Volume_Millions Symbol   VolStat    Return
0           3.2811    DUK  0.017051  0.014732
1           2.2363    DUK  0.009677 -0.005645
2           2.7822    DUK  0.021883 -0.013238
3           3.2490    DUK  0.014323 -0.006611
4           3.9602    DUK  0.016888  0.009690
```

Looking at the code block above, we've filtered out the rows that correspond to each symbol. Namely,

`agg_df['Symbol'] == 'DUK'`

returns a boolean series of the same number of rows of `agg_df`, where each value is `True` or `False` depending on whether a specific row's `Symbol` values is equal to `'DUK'`.

This row extraction technique will be useful to us later in this case when we perform analyses on each individual stock symbol.

1.7.1 Exercise 2:

If we added the number of rows together from the five DataFrames, `D_df`,`NEE_df`,`EXC_df`,`SO_df`, and `DUK_df`, we'd arrive at the same number of rows as `agg_df`: 6295 rows. If we instead used the `!=` operator in the five lines where we filter out each symbol, how many rows would we have if we sum all the rows in the five new DataFrames?

(a) 31475

(b) 12590

(c) 25180

(d) 6295

Answer. (c). By using the `!=` symbol, we filter out one symbol at a time, rather than keep one symbol when we used the `==` operator. Hence, each DataFrame will have 1259*4 = 5036 rows. Since we will have five DataFrames of this size, we will end with a total of 5036*5 = 25180 rows.

1.7.2 Exercise 3:

Write code to write a for loop to loop through each of the five symbols, extract only the rows correpsonding to each symbol, and calculate and print the average `VolStat` value for each of the five symbols.

Answer. One possible solution is indicated below:

[27]:
```
# One possible solution
symbol_list = ['D','EXC','NEE','SO','DUK']

for i in symbol_list:
    print(i)
    symbol_df = agg_df[agg_df['Symbol'] == i]
    symbol_avg_volstat = symbol_df['VolStat'].mean()
    print(symbol_avg_volstat)
```

```
D
0.014835992194363283
EXC
0.017721713893556437
NEE
0.014881084105602231
SO
0.014064780560964833
DUK
0.014534013252371459
```

1.8 Analyzing each stock's volatility levels

pandas offers the ability to group related rows of DataFrames according to the values of other rows. This useful feature is accomplished using the groupby() method. Let's take a look and see how this can be used to group rows so that each group corresponds to a single stock symbol:

[28]:
```
# Use the groupby() method, notice a DataFrameGroupBy object is returned
agg_df.groupby('Symbol')
```

[28]:
```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x7fe21ab72a20>
```

Here, the DataFrameGroupBy object can be most readily thought of as containing a DataFrame object for every group (in this case, a DataFrame object for each symbol). Specifically, each item of the object is a tuple, containing the group identifier (in this case the Symbol), and the corresponding rows of the DataFrame that have that Symbol).

Fortunately, pandas allows you to iterate over the groupby object to see what's inside:

[29]:
```
grp_obj = agg_df.groupby('Symbol') # Group data in agg_df by Symbol

# Loop through groups
for item in grp_obj:
    print(" ------ Loop Begins ------ ")
    print(type(item))      # Showing type of the item in grp_obj
    print(item[0])         # Symbol
    print(item[1].head()) # DataFrame with data for the Symbol
    print(" ------ Loop Ends ------ ")
```

```
 ------ Loop Begins ------
<class 'tuple'>
D
         Date       Open       High        Low      Close  Adj Close   Volume  \
0  2014-07-28  69.750000  71.059998  69.750000  70.879997  57.963978  1806400
1  2014-07-29  70.669998  70.980003  69.930000  69.930000  57.187099  2231100
```

```
2  2014-07-30  70.000000  70.660004  68.400002  68.970001  56.402020  2588900
3  2014-07-31  68.629997  68.849998  67.580002  67.639999  55.314388  3266900
4  2014-08-01  67.330002  68.410004  67.220001  67.589996  55.273487  2601800

   Volume_Millions Symbol   VolStat    Return
0           1.8064      D  0.018781  0.016201
1           2.2311      D  0.014858 -0.010471
2           2.5889      D  0.032286 -0.014714
3           3.2669      D  0.018505 -0.014425
4           2.6018      D  0.017674  0.003861
 ------ Loop Ends ------
 ------ Loop Begins ------
<class 'tuple'>
DUK
         Date       Open       High        Low      Close  Adj Close   Volume  \
0  2014-07-28  73.309998  74.480003  73.230003  74.389999  59.266285  3281100
1  2014-07-29  74.400002  74.480003  73.760002  73.980003  58.939648  2236300
2  2014-07-30  74.029999  74.199997  72.580002  73.050003  58.198696  2782200
3  2014-07-31  72.610001  73.099998  72.059998  72.129997  57.465740  3249000
4  2014-08-01  72.239998  73.370003  72.150002  72.940002  58.111061  3960200

   Volume_Millions Symbol   VolStat    Return
0           3.2811    DUK  0.017051  0.014732
1           2.2363    DUK  0.009677 -0.005645
2           2.7822    DUK  0.021883 -0.013238
3           3.2490    DUK  0.014323 -0.006611
4           3.9602    DUK  0.016888  0.009690
 ------ Loop Ends ------
 ------ Loop Begins ------
<class 'tuple'>
EXC
         Date       Open       High        Low      Close  Adj Close   Volume  \
0  2014-07-28  31.410000  32.130001  31.379999  31.950001  26.442406  5683400
1  2014-07-29  31.940001  32.049999  31.430000  31.469999  26.045147  6292800
2  2014-07-30  31.629999  31.660000  30.850000  31.010000  25.664442  7976600
3  2014-07-31  30.930000  31.490000  30.799999  31.080000  25.722378  9236100
4  2014-08-01  31.139999  32.080002  31.100000  31.540001  26.103081  9734300

   Volume_Millions Symbol   VolStat    Return
0           5.6834    EXC  0.023878  0.017192
1           6.2928    EXC  0.019411 -0.014715
2           7.9766    EXC  0.025609 -0.019602
3           9.2361    EXC  0.022308  0.004850
4           9.7343    EXC  0.031471  0.012845
 ------ Loop Ends ------
 ------ Loop Begins ------
<class 'tuple'>
NEE
```

```
         Date       Open       High        Low      Close  Adj Close   Volume  \
0  2014-07-28  98.470001  99.760002  98.099998  99.580002  85.106087  1643000
1  2014-07-29  99.029999  99.389999  97.300003  98.400002  84.097595  1942500
2  2014-07-30  98.160004  98.500000  95.760002  96.339996  82.337006  2844100
3  2014-07-31  95.639999  95.980003  93.800003  93.889999  80.243126  2725200
4  2014-08-01  93.500000  94.919998  93.279999  93.820000  80.183289  2514400

   Volume_Millions Symbol   VolStat    Return
0           1.6430    NEE  0.016858  0.011272
1           1.9425    NEE  0.021105 -0.006362
2           2.8441    NEE  0.027914 -0.018541
3           2.7252    NEE  0.022794 -0.018298
4           2.5144    NEE  0.017540  0.003422
 ------ Loop Ends ------
 ------ Loop Begins ------
<class 'tuple'>
SO
         Date       Open       High        Low      Close  Adj Close   Volume  \
0  2014-07-28  44.619999  45.430000  44.619999  45.360001  35.349178  5568900
1  2014-07-29  45.470001  45.470001  44.669998  44.860001  34.959522  5499600
2  2014-07-30  45.000000  45.000000  44.009998  44.380001  34.585461  6945200
3  2014-07-31  43.889999  43.889999  43.220001  43.290001  34.139881  5675300
4  2014-08-01  43.340000  43.830002  43.250000  43.320000  34.163548  4193700

   Volume_Millions Symbol   VolStat    Return
0           5.5689     SO  0.018153  0.016585
1           5.4996     SO  0.017594 -0.013415
2           6.9452     SO  0.022000 -0.013778
3           5.6753     SO  0.015265 -0.013670
4           4.1937     SO  0.013383 -0.000461
 ------ Loop Ends ------
```

Let's combine the `pd.groupby()` method with the `describe()` method and apply it to each symbol to analyze the distribution of volatility related features for each symbol.

[30]:
```
grp_obj = agg_df.groupby('Symbol') # Group data in agg_df by Symbol

# Loop through groups
for item in grp_obj:
    print('------Symbol: ', item[0])
    grp_df = item[1]
    relevant_df = grp_df[['VolStat']]
    print(relevant_df.describe())
```

```
------Symbol:  D
           VolStat
count  1259.000000
mean      0.014836
```

```
std       0.006548
min       0.003640
25%       0.010246
50%       0.013528
75%       0.017920
max       0.062350
------Symbol:  DUK
           VolStat
count  1259.000000
mean      0.014534
std       0.007047
min       0.003548
25%       0.010075
50%       0.012922
75%       0.017653
max       0.117492
------Symbol:  EXC
           VolStat
count  1259.000000
mean      0.017722
std       0.008129
min       0.005230
25%       0.011868
50%       0.015931
75%       0.021752
max       0.093156
------Symbol:  NEE
           VolStat
count  1259.000000
mean      0.014881
std       0.006544
min       0.004454
25%       0.010309
50%       0.013439
75%       0.017700
max       0.048495
------Symbol:  SO
           VolStat
count  1259.000000
mean      0.014065
std       0.006109
min       0.003960
25%       0.009786
50%       0.012858
75%       0.016865
max       0.051847
```

One immediate observation of note is that the volatility level on any given day can vary widely.

This is evident from the wide spread between the minimum and maximum `VolStat` levels seen using the `describe()` method. For example, stock symbol D has a minimum `VolStat` value of 0.003640, while its maximum `VolStat` value is 0.062350. That's more than a ten-times increase in the value of `VolStat`!

While this is great to see, there is a more powerful way to display this data in pandas. We can call the `describe()` method directly on the `DataFrameGroupBy` object. This one line allows you to avoid having to write a for loop every time you'd like to summarize data:

```
[31]: # VolStat
      agg_df[['Symbol','VolStat']].groupby('Symbol').describe()
```

```
[31]:        VolStat                                                          \
              count      mean       std       min       25%       50%       75%
      Symbol
      D      1259.0  0.014836  0.006548  0.003640  0.010246  0.013528  0.017920
      DUK    1259.0  0.014534  0.007047  0.003548  0.010075  0.012922  0.017653
      EXC    1259.0  0.017722  0.008129  0.005230  0.011868  0.015931  0.021752
      NEE    1259.0  0.014881  0.006544  0.004454  0.010309  0.013439  0.017700
      SO     1259.0  0.014065  0.006109  0.003960  0.009786  0.012858  0.016865


                  max
      Symbol
      D      0.062350
      DUK    0.117492
      EXC    0.093156
      NEE    0.048495
      SO     0.051847
```

This data is identical to the data previously outputted using the for loop approach. The difference is that utilizing the features of the `DataFrameGroupBy` object allows for easy coding, fast results, and a clean output. This illustrates the power of using the `pd.groupby()` method: generating statistics for groups of interest in your data is straightforward and efficient to code.

1.8.1 Exercise 4:

What are some insights you can draw from the `VolStat` summary statistics in terms of volatility levels?

Answer. Symbol EXC seems to have higher volatility that the other stocks. Perhaps symbol EXC has a different business model or has had a turbulent business environment in recent years. Further analysis is needed to determine the actual cause of the higher volatility relative to the other energy sector stocks.

1.8.2 Exercise 5:

Using `agg_df` and a for loop, write a script to determine the mean value of `VolStat` for each symbol by year.

Answer. One possible solution is shown below:

[32]:
```python
# One possible solution

# Add a column for the year
year_list = []
for i in agg_df['Date']:
    year_list.append(i[:4])

agg_df['YYYY'] = year_list

# Group by symbol, then loop through the group object to group by year and␣
 ↪calculate mean VolStat
grp = agg_df.groupby('Symbol')
for item in grp:
    print('------Symbol: ', item[0])
    grp_df = item[1]
    grp_df.head()
    relevant_df = grp_df[['YYYY','VolStat']]
    print(relevant_df.groupby('YYYY').mean())
```

```
------Symbol:  D
       VolStat
YYYY
2014  0.016510
2015  0.015748
2016  0.015051
2017  0.011246
2018  0.016678
2019  0.014631
------Symbol:  DUK
       VolStat
YYYY
2014  0.014592
2015  0.016215
2016  0.015841
2017  0.010032
2018  0.016472
2019  0.013723
------Symbol:  EXC
       VolStat
YYYY
2014  0.020166
```

```
2015  0.021383
2016  0.019270
2017  0.013746
2018  0.017106
2019  0.014720
------Symbol:  NEE
       VolStat
YYYY
2014  0.015843
2015  0.016274
2016  0.015805
2017  0.011648
2018  0.016043
2019  0.013692
------Symbol:  SO
       VolStat
YYYY
2014  0.013988
2015  0.014625
2016  0.014233
2017  0.010955
2018  0.016859
2019  0.013395
```

1.9 Labelling data points as high or low volatility

Now that we've determined that the volatility levels of each stock can vary widely, the next logical step is to group periods of high and low volatility so that we can then look at how volume differs between those time periods.

However, we don't currently have a column that identifies when volatility is high and when it is low. Therefore, we must create a new column called `VolLevel` using some volatility threshold. For example, we'd like to have a new column value determined by:

```
if VolStat > threshold:
    VolLevel = 'HIGH'
else:
    VolLevel = 'LOW'
```

Here we will define low volatility levels by any `VolStat` below the 50th percentile (i.e. below the median level of volatility for that symbol). Each percentile value must be calculated by symbol to ensure that each symbol is individually analyzed.

Let's take a look how we can accomplish this task using `groupby()` functionality and the `quantile()` method, which returns the percentile for a given series of data:

[33]:
```
# Determine lower thresholds for volatility for each symbol
volstat_thresholds = agg_df.groupby('Symbol')['VolStat'].quantile(0.5) # 50th␣
↪percentile (median)
```

```
print(volstat_thresholds)
```

```
Symbol
D      0.013528
DUK    0.012922
EXC    0.015931
NEE    0.013439
SO     0.012858
Name: VolStat, dtype: float64
```

Since we'd like to label periods of high and low volatility by symbol, we will make use of the `np.where()` method in the `numpy` library. This method takes an input and checks a logical condition: if the condition is true, it will return its second argument, whereas if the condition is false, it will return its third argument. This is very similar to how Microsoft Excel's `IFERROR()` method works (helpful to think of it this way for those familiar with Excel). Let's loop through each symbol and label each day as either high and low volatility:

```
[34]: # Loop through symbols
print("Defining stock symbols")
list_of_symbols= ['D','EXC','NEE','SO','DUK']
list_of_df = []

# Loop over all symbols
print(" --- Loop over symbols --- ")
for i in symbol_data_to_load:
    print("Labelling Volatility regime for Symbol: " + i)
    temp_df = agg_df[agg_df['Symbol'] == i].copy() # make a copy of the␣
 ↪dataframe to ensure not affecting agg_df
    volstat_t = volstat_thresholds.loc[i]

    temp_df['VolLevel'] = np.where(temp_df['VolStat'] < volstat_t, 'LOW',␣
 ↪'HIGH') # Volatility regime label
    list_of_df.append(temp_df)

print(" --- Completed loop over symbols --- ")

print("Aggregating data")
labelled_df = pd.concat(list_of_df)
```

```
Defining stock symbols
 --- Loop over symbols ---
Labelling Volatility regime for Symbol: D
Labelling Volatility regime for Symbol: EXC
Labelling Volatility regime for Symbol: NEE
Labelling Volatility regime for Symbol: SO
Labelling Volatility regime for Symbol: DUK
 --- Completed loop over symbols ---
Aggregating data
```

```
[35]: labelled_df.head()
```

```
[35]:          Date       Open       High        Low      Close  Adj Close   Volume  \
      0  2014-07-28  69.750000  71.059998  69.750000  70.879997  57.963978  1806400
      1  2014-07-29  70.669998  70.980003  69.930000  69.930000  57.187099  2231100
      2  2014-07-30  70.000000  70.660004  68.400002  68.970001  56.402020  2588900
      3  2014-07-31  68.629997  68.849998  67.580002  67.639999  55.314388  3266900
      4  2014-08-01  67.330002  68.410004  67.220001  67.589996  55.273487  2601800

         Volume_Millions Symbol   VolStat    Return  YYYY VolLevel
      0           1.8064      D  0.018781  0.016201  2014     HIGH
      1           2.2311      D  0.014858 -0.010471  2014     HIGH
      2           2.5889      D  0.032286 -0.014714  2014     HIGH
      3           3.2669      D  0.018505 -0.014425  2014     HIGH
      4           2.6018      D  0.017674  0.003861  2014     HIGH
```

We've now added a `VolLevel` column that identifies whether each symbol is in a period of high or low volatility on any given day. Since we know that the bank will require higher trading volume in order to transact in periods of high volatility, let's now take a look at the average daily traded volume for high volatility vs. low volatility days.

1.10 Is daily trading volume affected by the level of volatility?

To explore the relationship between volatility level and daily trading volume, let's group by `VolLevel` and take a look at the average `Volume` for the HIGH and LOW volatility groups:

```
[36]: labelled_df.groupby(['Symbol','VolLevel'])[['Volume_Millions']].mean()
```

HIGH
LOW

```
[36]:                  Volume_Millions
      Symbol VolLevel
      D      HIGH             3.538901
             LOW              2.636641
      DUK    HIGH             3.760172
             LOW              2.825710
      EXC    HIGH             7.090384
             LOW              5.031123
      NEE    HIGH             2.361096
             LOW              1.707347
      SO     HIGH             6.148537
             LOW              4.417179
```

1.10.1 Exercise 6:

What is an immediate trend you notice regarding the volatility regimes?

Answer. Higher volatility is related to a higher daily trading volume. The pattern is consistent across all five symbols, indicating that we've found an interesting aspect of how volatility and trading volume are related.

1.10.2 Exercise 7:

Write code to group time periods into Low, Medium, or High volatility regimes, where:

```
if VolStat > (75th percentile VolStat for given symbol):
    VolLevel = 'HIGH'
elif  VolStat > (25th percentile VolStat for given symbol):
    VolLevel = 'MEDIUM'
else:
    VolLevel = 'LOW'
```

Output a `final_df` DataFrame output grouped by Symbol, showing the mean Volume for each VolLevel category.

Answer. One possible solution is shown below:

[37]:
```
# One possible solution
# Determine lower thresholds for volatility for each symbol
volstat_thresholds_75 = agg_df.groupby('Symbol')['VolStat'].quantile(0.75) #␣
 ↪50th percentile (median)
volstat_thresholds_25 = agg_df.groupby('Symbol')['VolStat'].quantile(0.25) #␣
 ↪50th percentile (median)

# Loop through symbols
print("Defining stock symbols")
list_of_symbols= ['D','EXC','NEE','SO','DUK']
list_of_df = []

# Loop over all symbols
print(" --- Loop over symbols --- ")
for i in symbol_data_to_load:
    print("Labelling Volatility regime for Symbol: " + i)
    temp_df = agg_df[agg_df['Symbol'] == i].copy() # make a copy of the␣
 ↪dataframe to ensure not affecting agg_df
    volstat_t75 = volstat_thresholds_75.loc[i]
    volstat_t25 = volstat_thresholds_25.loc[i]

    temp_df['VolLevel'] = np.where(temp_df['VolStat'] > volstat_t75, 'HIGH',
                                   np.where(temp_df['VolStat'] > volstat_t25,␣
 ↪'MEDIUM','LOW')) # Volatility regime label
    list_of_df.append(temp_df)

print(" --- Completed loop over symbols --- ")

print("Aggregating data")
final_df = pd.concat(list_of_df)
```

```
print(final_df.groupby(['Symbol','VolLevel'])[['Volume_Millions']].mean())
```

```
Defining stock symbols
 --- Loop over symbols ---
Labelling Volatility regime for Symbol: D
Labelling Volatility regime for Symbol: EXC
Labelling Volatility regime for Symbol: NEE
Labelling Volatility regime for Symbol: SO
Labelling Volatility regime for Symbol: DUK
 --- Completed loop over symbols ---
Aggregating data
                 Volume_Millions
Symbol VolLevel
D      HIGH             3.921797
       LOW              2.456831
       MEDIUM           2.986784
DUK    HIGH             4.169937
       LOW              2.603308
       MEDIUM           3.199854
EXC    HIGH             7.904227
       LOW              4.671271
       MEDIUM           5.835034
NEE    HIGH             2.653587
       LOW              1.608036
       MEDIUM           1.937998
SO     HIGH             6.781142
       LOW              4.111777
       MEDIUM           5.120373
```

1.11 Graphing volatility across time

We've now satisfactorily answered our original question. However, you don't need to just analyze data in tabular format. Python contains functionality to allow you to analyze your data visually as well.

We will use `pandas` functionality built on the standard Python plotting library `matplotlib`. Let's import the library and instruct Jupyter to display the plots inline (i.e. display the plots to the notebook screen so we can see them as we run the code):

[38]:
```
# import fundamental plotting library in Python
import matplotlib.pyplot as plt

# Instruct jupyter to plot in the notebook
%matplotlib inline
```

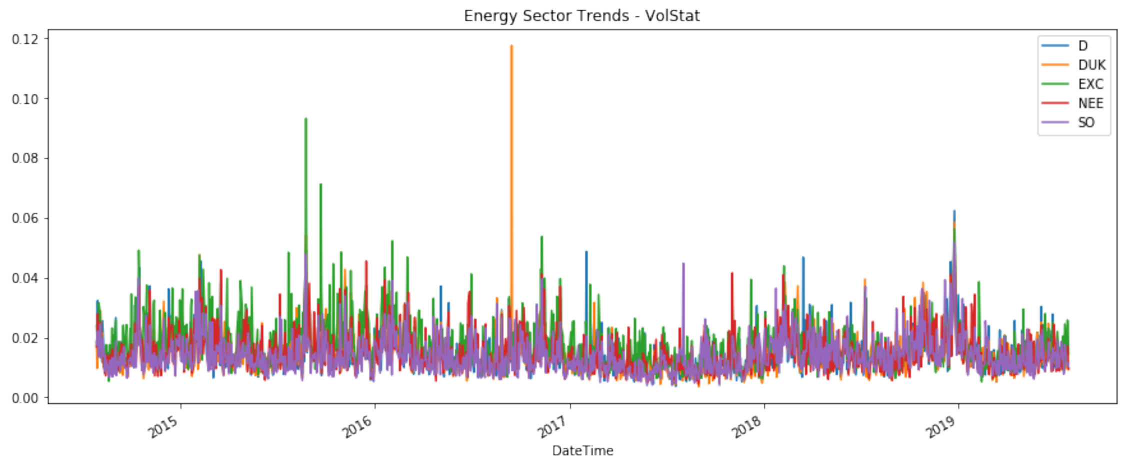
Before we plot, we need to convert the the `Date` column in `agg_df` to a `datetime`-like object, Python's internal data representation of dates. `pandas` offers the `to_datetime()` method to convert a String that represents a given date format into a `datetime`-like object. We instruct `pandas` to use `format='%Y-%m-%d'`, since our dates are in this format, where %Y indicates the numerical year, %m indicates the numerical month and %d indicates the numerical day. If our dates were in another format, we'd modify this input value appropriately.

```
[39]: # To convert a string to a datetime
      agg_df['DateTime'] = pd.to_datetime(agg_df['Date'], format='%Y-%m-%d')

      # Set index as DateTime for plotting purposes
      agg_df = agg_df.set_index(['DateTime'])
      agg_df.head()
```

```
[39]:                   Date       Open       High        Low      Close  Adj Close  \
      DateTime
      2014-07-28  2014-07-28  69.750000  71.059998  69.750000  70.879997  57.963978
      2014-07-29  2014-07-29  70.669998  70.980003  69.930000  69.930000  57.187099
      2014-07-30  2014-07-30  70.000000  70.660004  68.400002  68.970001  56.402020
      2014-07-31  2014-07-31  68.629997  68.849998  67.580002  67.639999  55.314388
      2014-08-01  2014-08-01  67.330002  68.410004  67.220001  67.589996  55.273487

                   Volume  Volume_Millions Symbol   VolStat    Return  YYYY
      DateTime
      2014-07-28  1806400           1.8064      D  0.018781  0.016201  2014
      2014-07-29  2231100           2.2311      D  0.014858 -0.010471  2014
      2014-07-30  2588900           2.5889      D  0.032286 -0.014714  2014
      2014-07-31  3266900           3.2669      D  0.018505 -0.014425  2014
      2014-08-01  2601800           2.6018      D  0.017674  0.003861  2014
```

Now we are ready to look directly at volatility across time. Let's group by symbols and plot the `VolStat` value across time. Each symbol's time series will be labelled a different color by default:

```
[40]: # Look at volatility regimes
      fig, ax = plt.subplots(figsize=(15,6))
      agg_df.groupby('Symbol')['VolStat'].plot(ax=ax, legend=True, title='Energy␣
       ↪Sector Trends - VolStat');
```



We notice that periods of high volatility tend to ”clump” together; that is, periods of high volatility are not uniformly and randomly distributed across time, but tend to occur in highly concentrated bursts. This is an interesting insight that we could not gain by only looking at the data in tabular format. In future cases, you will dig deeper into the numerous graphing capabilities of Python and how to integrate them into your data science workflow.

1.11.1 Exercise 8:

Write a script to find and print the month that has the highest average daily trading volume for each symbol. Also include the average volume value corresponding to that month. For example, symbol D has its highest average daily trading volume of 6.437 million in December 2018.

Answer. One possible solution is shown below:

[41]:
```
# One possible solution

# Add a column for the year
yearmonth_list = []
for i in agg_df['Date']:
    yearmonth_list.append(i[:4]+i[5:7])

agg_df['YYYYMM'] = yearmonth_list

# Group by symbol, then loop through the group object to group by yearmonth and␣
 ↪calculate mean volume traded for each month
grp = agg_df.groupby('Symbol')
for item in grp:
    print('------Symbol: ', item[0])
    grp_df = item[1]
    grp_df.head()
    relevant_df = grp_df[['YYYYMM','Volume_Millions']]
    yearmonth_df = relevant_df.groupby('YYYYMM').mean()

    max_volume = float(yearmonth_df.max())
    print(yearmonth_df[yearmonth_df['Volume_Millions'] == max_volume])
```

```
------Symbol:  D
        Volume_Millions
YYYYMM
201812         6.437421
------Symbol:  DUK
        Volume_Millions
YYYYMM
201809         4.624937
------Symbol:  EXC
        Volume_Millions
YYYYMM
201602          9.66578
------Symbol:  NEE
        Volume_Millions
YYYYMM
201611         3.618452
------Symbol:  SO
        Volume_Millions
YYYYMM
201802         8.456853
```

1.12 Conclusions

Having completed the analysis of the energy sector stock data, we have identified a number of interesting patterns relating volatility to trading volume. Specifically, we found that periods of high volatility also exhibit very high volume. This trend is consistent across all symbols.

We also saw that each stock exhibited ”volatility clustering” – periods of high volatility tend to be clumped together. Each of the stocks experienced high volatility at relatively similar times which suggests some broader market factor may be affecting the energy sector.

1.13 Takeaways

In this case, we’ve learned the foundations of the `pandas` library in Python. We now know how to:

1. Read data from CSV files
2. Aggregate and manipulate data using `pandas`
3. Analyze summary statistics and gather information from trends across time

Going forward, we will be able to use `pandas` as a data analysis framework to build more complex projects and solve critical business problems.