case_1.4

April 16, 2020

1 How are returns and volatility related for energy stocks and the broader market?

1.1 Introduction

Business Context. You are an analyst at a large bank focused on natural resource stock investments. You recently conducted an analysis of the following energy stocks and how their trading volume is related to their volatility:

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)

Your boss was quite pleased with your previous analysis, and now wants you to conduct additional analysis so he can figure out how to size potential positions in these stocks... i.e. what percentage of the investment portfolio should be dedicated to each of these stocks. Specifically, he wants you to look at daily returns and volatility for each stock as well as for the broader market (i.e. not just the energy sector).

This is important because high volatility implies higher risk, and your boss would like to know if the potential returns of these high-volatility energy stocks compensate him for the added risk. Additionally, because his performance is measured or benchmarked against the broader market, he wants to understand whether these stocks generally outperform the broader market.

Business Problem. Based on the above context, your boss has posed the following question to you: **"What is the relationship between daily volatility and returns for these stocks, and what is the relationship between daily returns for these stocks and the broader stock market?"**

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. You will proceed by: (1) conducting preliminary cleaning of the data; (2) creating additional features required for our analysis; (3) labelling the data into volatility groups, or regimes, and determining how volatility is related to returns; and finally (4) comparing these returns against those of the broader market.

[1]:
```
# Import libraries required for this case
import pandas as pd
```

case_1.1

April 3, 2020

1 Pre-Case Setup: Python, Jupyter Notebook, Git

1.1 Python and Jupyter

Python is a general purpose programming language that allows for both simple and complex data analysis. Python is incredibly versatile, allowing analysts, consultants, engineers, and managers to obtain and analyze information for insightful decision-making.

The Jupyter Notebook is an open-source web application that allows for Python code development. Jupyter further allows for inline plotting and provides useful mechanisms for viewing data that make it an excellent resource for a variety of projects and disciplines.

The following section will outline how to install and begin working with Python and Juypter.

1.2 Setting up the Python Environment

Instruction guides for Windows and macOS are included below. Follow the one that corresponds with your operating system.

1.2.1 Windows Install:

Step 1: Open browser and go to https://www.anaconda.com/distribution/

Step 2: Click on "Windows" and then "Download" for Python 3.7 64-bit installer

Step 3: Run the downloaded file found in the downloads section from Step 2

Step 4: Click through the install prompts

Step 5: Go to menu (or Windows Explorer), find the Anaconda3 folder, and double-click to run

1.2.2 macOS Install:

Step 1: Open browser and go to https://www.anaconda.com/distribution/

Step 2: Click on "macOS" and then "Download" for Python 3.7 64-bit installer

Step 3: Run the downloaded file found in the downloads section from Step 2

Step 4: Click through the install prompts

Step 5: In Finder (or Launchpad), browse to the Anaconda3 folder to find the Jupyter program, and double-click to run

1.3 File Management with Python and Jupyter

It is common practice to have a main folder where all projects will be located (e.g. "jupyter_research"). The following are guidelines you can use for Python projects to help keep your code organized and accessible:

1. Create subfolders for each Jupyter-related project
2. Group related .ipynb files together in the same folder
3. It can be useful to create a "Data" folder within individual project folders if using a large number of related data files

You should now be set up and ready to begin coding in Python!

1.4 Setting up Git, GitHub, and cloning a repository

Git is a free and open source distributed version-control system. It is very useful for both simple software projects and large multi-functional projects that span thousands of files. The main website for Git documentation is https://git-scm.com/. We will go over a brief introduction to Git here to get up and running with the software, in addition to understanding how its version control system operates. We'll create a new repository and also cover how to clone a repository from GitHub. GitHub is an online platform that hosts repositories for users to interact with on their own projects and collaborative projects.

The following steps outline how to get started with Git, sign up for a GitHub account, create a repository, and clone a repository to your computer:

Step 1: Open your browser and go to https://git-scm.com/downloads and download Git for your operating system

Step 2: Create a new repository by browsing to the folder on your computer that you'd like to work under. From this folder, run the command: git init

Step 3: Open your browser and go to https://github.com/ and sign up for a Github account

Step 4: On GitHub, navigate to the main page of the repository you are interested in using. Note that when you create a repository on GitHub, it is a remote repository. We will clone a repository to create a local copy on your computer and we can then sync between the two locations when needed. Hence, we will have a local copy of the repository and a remote copy of the repository.

Step 5: Under the repository name on Github, click clone or download

Step 6: In the Clone with HTTPs section, copy the clone URL link

Step 7: Open your command line and navigate to the current working directory to the location where you want the cloned directory to be made. Type and run the command: git clone https://github.com/YOUR-USERNAME/YOUR-REPOSITORY

You should have now created your local clone repository. Let's cover some of the utilities available with Git.

1.5 Understanding the Git workflow

Git is a distributed version-control system. The Git workflow is built upon three objects managed by Git: 1. Your working directory 2. The index (staging area) 3. The HEAD (points to your last commit)

The central workflow with Git is to locally edit files on your staging area (updating the index) and commit the changes to your working directory (therefore updating the HEAD) when you are ready to save this version of your work to your repository. While you are editing the files in the staging area, you can of course save these files as you progress, however the files on your working directory will not have changed since you need to commit the changes. Hence, you essentially have two versions of files, one that you are editing, and one that is a snapshot from the previous commit.

Let's go through a simple example of committing a change to some text file `hello.txt`.

Step 1: Create a text file hello.txt on your local computer in the folder you will be using Git from.

Step 2: Edit the file to add one line to the text file. Say you add "Hello World" as the first line. Save the file.

Step 3: Update your staging area (i.e. update the index). To do this, use the command: `git add hello.txt`

(If you have multiple files and you'd like to add all the files with changes in the folder to the staging area you can use the command: `git add *`)

Step 4: To commit the changes on the staging area to the working directory, use the command: `git commit -m "Commit message"`

(You should update the commit message to provide useful information. Do not neglect this as it can be very useful to have informative commit messages when debugging unexpected issues with your files)

Step 5: Step 4 commits the file to the HEAD, but the remote repository will not update, so you've updated the HEAD of your local working copy. To push the changes to your remote repository, use the command: `git push origin master`

Furthermore, useful commands to check the status and history of your repository include: 1. Command to check repository status: `git status` 2. Command to check history of repository: `git log`

This covers the basics of Git. While it takes considerable practice to master, we've covered the cloning, adding, committing, and pushing features that are the backbone of the Git version control software.

We're ready to begin the case.

2 Identifying Expansion Opportunities for Luxury Commercial Airline Flights

2.1 Case Introduction

Business Context. You are an employee for GrowthAir, a growing commercial airline company. In the past few years, GrowthAir has expanded luxury flight services to locations across the globe. Following your team's excellent performance in identifying new business opportunities last year, you have been tasked with identifying the top countries to further expand GrowthAir's luxury flight service.

Business Problem. Your manager has asked you to answer the following question: "In which countries should GrowthAir expand its luxury flight service?"

Analytical Context. The relevant data is a series of success estimates (i.e. probabilities of success) that your internal marketing research teams have come up with. Using your ability to conduct data analysis in Python, you will embark on summarizing the available success estimates to produce a concise recommendation to your boss.

2.2 Fundamentals of Python

Python is an interpreted, high-level general programming language that was first released in 1991. Python allows users to easily manipulate data and store values in what are known as objects. Everything in Python is an object and has a type. For example, if a user aims to store the integer 5 in an object named `my_int`, this can be accomplished by the statement, `my_int = 5`. This statement tells Python to assign the integer value of 5 to the **variable** `my_int` (called a variable here because it can change value). `my_int` is a Python object, and has type `int`.

Similar to how Excel distinguishes different data types (such as Text, Number, Currency, Scientific), Python offers a variety of data types. Here are a few common data types:

1. Integers, type `int`: `my_int = 1`
2. Float type `float`: `my_float = 25.5`
3. Strings, type `str`: `my_string = 'Hello'`

Here we see (1) integers and (2) floats store numeric data. The difference between the two is that floats store decimal variables, whereas the integer type can only store integer variables. (3) is the string type. Strings are used to store textual data in Python. This case will use string variables to store country names. They are often used to store identifiers such as person names, city names, and more.

There are other data types available in Python; however, these are the three fundamental types that you will see across almost every Python program. Always keep in mind that **every** object in Python has a type.

Now that we've covered the fundamentals of Python, let's take a look at GrowthAir's proprietary company data on country success estimates.

2.3 Exploring company data on success estimates

Let's take a look at a common data structure used to hold your company's proprietary data on estimates of probability of success for global expansion projects by country. The `success_estimates` variable below is a Python dictionary, which is being assigned certain data using the `'='` assignment operator. Each estimate here is a number (float) between 0 and 1, inclusive, which represents the probability that expanding to that country will be successful.

Python's dictionary type stores key-value pairs that allow users to quickly access information for a particular key. By specifying a key, the user can return the value corresponding to the given key. Python's syntax for dictionaries uses curly braces {},

```
user_dictionary = {'Key1':Value1, 'Key2':Value2, 'Key3':Value3}
```

The `success_estimates` dictionary has keys which are strings, and values which are of type list. A list is an incredibly useful data structure in Python that can store any number of Python objects, and are denoted by the use of square brackets `[]`. In `success_estimates` below, the list contains float types. Lists are versatile and can be expanded by adding new elements to the end of the list (the right-most side is considered the end of the list). Moreover, list elements (i.e. the objects in the list) can be accessed easily using integer indices. Interestingly, lists can also store other lists (called a lists of lists). This makes them a powerful tool for holding complex data sets.

Let's take a look at the `success_estimates` data:

[1]:
```
# Data on probability of expansion success by country estimates
success_estimates = {'Australia': [0.6, 0.33, 0.11, 0.14],
                     'France': [0.66, 0.78, 0.98, 0.2],
                     'Italy': [0.6],
                     'Brazil': [0.22, 0.22, 0.43],
                     'USA': [0.2, 0.5, 0.3],
                     'England': [0.45],
                     'Canada': [0.25, 0.3],
                     'Argentina': [0.22],
                     'Greece': [0.45, 0.66, 0.75, 0.99, 0.15, 0.66],
                     'Morocco': [0.29],
                     'Tunisia': [0.68, 0.56],
                     'Egypt': [0.99],
                     'Jamaica': [0.61, 0.65, 0.71],
                     'Switzerland': [0.73, 0.86, 0.84, 0.51, 0.99],
                     'Germany': [0.45, 0.49, 0.36]}
```

Python easily allows you to print the elements stored in any variable to the screen using the `print()` statement:

[2]:
```
print(success_estimates)
```

```
{'Australia': [0.6, 0.33, 0.11, 0.14], 'France': [0.66, 0.78, 0.98, 0.2],
'Italy': [0.6], 'Brazil': [0.22, 0.22, 0.43], 'USA': [0.2, 0.5, 0.3], 'England':
[0.45], 'Canada': [0.25, 0.3], 'Argentina': [0.22], 'Greece': [0.45, 0.66, 0.75,
0.99, 0.15, 0.66], 'Morocco': [0.29], 'Tunisia': [0.68, 0.56], 'Egypt': [0.99],
```

```
'Jamaica': [0.61, 0.65, 0.71], 'Switzerland': [0.73, 0.86, 0.84, 0.51, 0.99],
'Germany': [0.45, 0.49, 0.36]}
```

Notice the re-ordering of the dictonary elements when we print compared to the order in which we originally defined the dictionary. This is a key aspect of dictionary data types – they are unordered! (This is very different compared to list data types, which are ordered. More on this later.)

Now intuitively, we would like to recommend that the business put effort into the country with the highest success estimate. But what does this mean when there are multiple success estimates for some countries, and only one for others? We will explore this next.

2.4 Interacting with dictionaries and lists

Taking a careful look at the `success_estimates` dictionary, you notice some countries only have one success estimate, while others have many. For example, England has only one estimate contained in its list [0.45], while Jamaica has three estimates contained in its list [0.61, 0.65, 0.71]. Let's zoom in on Jamaica and take a look at some summary statistics of the estimates.

In Python, the dictionary type has built-in methods (functions, which we will discuss later) to access the dictionary keys and values. These methods are called by typing `.keys()` or `.values()` after the dictonary object. We will change the return type of calling `.keys()` and `.values()` to a list by using the `list()` method.

[3]:
```
# Look at the keys...
list(success_estimates.keys())
```

[3]:
```
['Australia',
 'France',
 'Italy',
 'Brazil',
 'USA',
 'England',
 'Canada',
 'Argentina',
 'Greece',
 'Morocco',
 'Tunisia',
 'Egypt',
 'Jamaica',
 'Switzerland',
 'Germany']
```

[4]:
```
# ...and their corresponding values
list(success_estimates.values())
```

[4]:
```
[[0.6, 0.33, 0.11, 0.14],
 [0.66, 0.78, 0.98, 0.2],
 [0.6],
```

```
 [0.22, 0.22, 0.43],
 [0.2, 0.5, 0.3],
 [0.45],
 [0.25, 0.3],
 [0.22],
 [0.45, 0.66, 0.75, 0.99, 0.15, 0.66],
 [0.29],
 [0.68, 0.56],
 [0.99],
 [0.61, 0.65, 0.71],
 [0.73, 0.86, 0.84, 0.51, 0.99],
 [0.45, 0.49, 0.36]]
```

We will make use of the access to keys and values of a dictionary later in the case when comparing across numerous countries' estimates. For now, just remember that you can access a dictionary's full list of keys or values simply by calling built-in methods.

We'd also like to check if a country name is one the keys in the dictionary. Python allows us to check if a key is in a dictionary through the use of the `in` keyword. The statement `key in dictionary` will return a boolean type of `True` if the key is one of the keys in the dictionary and `False` otherwise. Let's take a look at how this works.

[5]:
```
print('Checking if Morocco key is present:')
print('Morocco' in success_estimates)

print('Checking if Japan key is present:')
print('Japan' in success_estimates)
```

```
Checking if Morocco key is present:
True
Checking if Japan key is present:
False
```

We'd now like to access the value corresponding to a specific key in the `success_estimates` dictionary. Simply type the value name in square brackets adjacent to the dictionary name. For example, `success_estimates['Jamaica']` will return Jamaica's list of estimates:

[6]:
```
success_estimates['Jamaica']
```

[6]:
```
[0.61, 0.65, 0.71]
```

If you would like to store the result in a variable to be used later, use the assignment operator `'='`:

[7]:
```
jamaica_list = success_estimates['Jamaica']
```

You can then view the contents of the list via the `print()` method:

[8]:
```
print(jamaica_list)
```

```
[0.61, 0.65, 0.71]
```

Here, you'll see that the order of the elements in the Jamaica list is the same as what was originally defined above. This is because lists are ordered objects. In fact, you can access elements of a list by an index. In Python, indices start at 0 (for the first element of a given list) and increment by 1 for each successive element. For example, let's print each element of the Jamaica list:

[9]:
```
# Each successive print statement will print on a new line
print(jamaica_list[0]) # prints the first element of the list
print(jamaica_list[1]) # prints the second element of the list
print(jamaica_list[2]) # prints the third element of the list
```

```
0.61
0.65
0.71
```

Python also offers a simple way to determine the length of a list: the `len()` method. We expect the length of `jamaica_list` to be 3 since it has three elements:

[10]:
```
len(jamaica_list) # returns the length of the list
```

[10]: 3

2.4.1 Exercise 1:

Print the length of the success estimate lists for France, Greece, and Morocco.

Answer. One possible solution is shown below:

[11]:
```
print('Number of estimates for France:')
print(len(success_estimates['France']))

print('Number of estimates for Greece:')
print(len(success_estimates['Greece']))

print('Number of estimates for Morocco:')
print(len(success_estimates['Morocco']))
```

```
Number of estimates for France:
4
Number of estimates for Greece:
6
Number of estimates for Morocco:
1
```

2.4.2 Exercise 2:

Which of the following would be useful to store project success estimates if they were available at a regional level instead of a country level?

(a) List

(b) Dictionary

(c) Float

(d) String

Answer. (b).

(a) Python lists store data in a an ordered manner, where each element of a list can be of any type. There is no mapping between keys and values in Python lists, rather list elements are accessed using integer indices. Lists are commonly used to group related data together for further processing. In this case, the regional level data would still consist of key-value pairs (where the keys are the region names), so lists are not ideal.

(b) B is the correct answer since dictionaries use a key-value pair to represent data. They are fast to access and are unordered. Dictionaries are especially useful when data is being accessed by a commonly used identifier, such as country names used in this case, or region names as indicated in this example.

(c) Floats are used to represent numeric data, hence are not used for mapping keys to values.

(d) Strings are used to represent textual data. While they are often used as the keys in a dictionary, the strings themselves are not built for storing key-value pairs of any kind.

Now that we're familiar with using lists and know that lists are ordered data structures while dictionaries are unordered data structures, let's begin to compare success estimates across countries.

2.5 Calculating a country-specific average success estimate

Continuing our analysis on Jamaica, the list contains three numbers, [0.61, 0.65, 0.71]. Recall these number are of type `float` in Python, which stores numeric decimal values. One logical way to summarize these estimates so that they can be compared across countries is to use the arithmetic average. Let's use basic arthimetic operators to calculate the average success estimate for Jamaica, storing the result in a new variable `avg_jamaica`:

[12]:
```
avg_jamaica = (0.61 + 0.65 + 0.71) / 3
print(avg_jamaica)
```

```
0.6566666666666666
```

We see the average probability of success estimate for Jamaica is approximately 0.657. However, we produced this estimate by hand-coding the values. If we were to do this for every country, it would take quite a long time. So we'd like to use a more automated way of producing the average.

To produce an average we can utilize a function. Functions operate on data and variables in Python to perform a desired action. Functions may have both inputs and outputs, just like familiar mathematical operators like addition, subtraction, multiplication, and division (which each have two inputs and one output). While functions in Python may still be for a mathematical purpose, such as squaring an integer, Python allows for more abstract function behaviour, such as printing to the screen. In this case, the `print()` function will print its input to the screen.

Let's use Python's built-in mathematical functions `sum()`, `min()`, and `max()` to calculate Jamaica's average success estimate, minimum success estimate, and maximum success estimate, respectively:

[13]:
```
country_name = 'Jamaica'
jamaica_list = success_estimates[country_name] # list of the estimates for␣
↪Jamaica
print(jamaica_list)
```

```
[0.61, 0.65, 0.71]
```

[14]:
```
avg_jamaica = sum(jamaica_list) / len(jamaica_list)
min_jamaica = min(jamaica_list)
max_jamaica = max(jamaica_list)
print("Country:",country_name,", Average:",avg_jamaica)
print("Country:",country_name,", Min:",min_jamaica)
print("Country:",country_name,", Max:",max_jamaica)
```

```
Country: Jamaica , Average: 0.6566666666666666
Country: Jamaica , Min: 0.61
Country: Jamaica , Max: 0.71
```

As expected, we get the same average result of approximately 0.657. Note that we could also have rounded the results to two decimal places using the `round()` method. This can improve readability.

[15]:
```
avg_jamaica = round(sum(jamaica_list) / len(jamaica_list),2)
min_jamaica = round(min(jamaica_list),2)
max_jamaica = round(max(jamaica_list),2)
print("Country:",country_name,", Average:",avg_jamaica)
print("Country:",country_name,", Min:",min_jamaica)
print("Country:",country_name,", Max:",max_jamaica)
```

```
Country: Jamaica , Average: 0.66
Country: Jamaica , Min: 0.61
Country: Jamaica , Max: 0.71
```

Functions in Python are a very powerful tool to increase productivity and perform more complex tasks.

2.5.1 Exercise 3:

Write a script to calculate the average success for every country. Output (using `print()`) each country's average success estimate to the screen. The print statements should output each country on a new line, for example:

```
Country: France , Average: 0.655
Country: Brazil , Average: 0.29
```

[16]:
```
# One possible solution
```

```
print("Country:",'France',", Average:",sum(success_estimates['France']) /␣
 ↪len(success_estimates['France']))
print("Country:",'Brazil',", Average:",sum(success_estimates['Brazil']) /␣
 ↪len(success_estimates['Brazil']))
print("Country:",'Argentina',", Average:",sum(success_estimates['Argentina']) /␣
 ↪len(success_estimates['Argentina']))
print("Country:",'Germany',", Average:",sum(success_estimates['Germany']) /␣
 ↪len(success_estimates['Germany']))
print("Country:",'Australia',", Average:",sum(success_estimates['Australia']) /␣
 ↪len(success_estimates['Australia']))
print("Country:",'Canada',", Average:",sum(success_estimates['Canada']) /␣
 ↪len(success_estimates['Canada']))
print("Country:",'Greece',", Average:",sum(success_estimates['Greece']) /␣
 ↪len(success_estimates['Greece']))
print("Country:",'USA',", Average:",sum(success_estimates['USA']) /␣
 ↪len(success_estimates['USA']))
print("Country:",'Switzerland',", Average:
 ↪",sum(success_estimates['Switzerland']) /␣
 ↪len(success_estimates['Switzerland']))
print("Country:",'Tunisia',", Average:",sum(success_estimates['Tunisia']) /
 ↪len(success_estimates['Tunisia']))
print("Country:",'Italy',", Average:",sum(success_estimates['Italy']) /␣
 ↪len(success_estimates['Italy']))
print("Country:",'Egypt',", Average:",sum(success_estimates['Egypt']) /␣
 ↪len(success_estimates['Egypt']))
print("Country:",'Jamaica',", Average:",sum(success_estimates['Jamaica']) /␣
 ↪len(success_estimates['Jamaica']))
print("Country:",'Morocco',", Average:",sum(success_estimates['Morocco']) /␣
 ↪len(success_estimates['Morocco']))
print("Country:",'England',", Average:",sum(success_estimates['England']) /␣
 ↪len(success_estimates['England']))
```

```
Country: France , Average: 0.655
Country: Brazil , Average: 0.29
Country: Argentina , Average: 0.22
Country: Germany , Average: 0.4333333333333333
Country: Australia , Average: 0.29500000000000004
Country: Canada , Average: 0.275
Country: Greece , Average: 0.61
Country: USA , Average: 0.3333333333333333
Country: Switzerland , Average: 0.7859999999999999
Country: Tunisia , Average: 0.6200000000000001
Country: Italy , Average: 0.6
Country: Egypt , Average: 0.99
Country: Jamaica , Average: 0.6566666666666666
Country: Morocco , Average: 0.29
```

```
Country: England , Average: 0.45
```

2.6 Systematically determine the average success estimate for all of the countries

The end goal of this analysis is a recommendation for where global expansion opportunities should be considered. To reach a conclusion, it'd be ideal to have the average success probability for each country.

To achieve this, we will use a control flow element in Python - the for loop. The `for` loop allows one to execute the same statements over and over again (i.e. looping). This saves a significant amount of time coding repetitive tasks and aids in code readability. The general structure of a for loop is:

```
for iterator_variable in some_sequence:
    statements(s)
```

The for loop iterates over `some_sequence` and performs `statements(s)` at each iteration. That is, at each iteration the `iterator_variable` is updated to the next value in `some_sequence`. As a concrete example, consider the loop:

```
for i in [1,2,3,4]:
    print(i*i)
```

Here, the for loop will print to the screen four times; that is it will print `1` on the first iteration of the loop, `4` on the second iteration, `9` on the third, and `16` on the fourth. Hence, the for loop statement will iterate over all the elements of the list `[1,2,3,4]`, and at each iteration it updates the iterator variable `i` to the next value in the list `[1,2,3,4]`.

Let's use a for loop on our country data by getting a list of all the keys in `success_estimates`:

[17]:
```
# Get all the keys from the success_estimates dictionary
country_name_list = list(success_estimates.keys())
print(country_name_list)
```

```
['Australia', 'France', 'Italy', 'Brazil', 'USA', 'England', 'Canada',
'Argentina', 'Greece', 'Morocco', 'Tunisia', 'Egypt', 'Jamaica', 'Switzerland',
'Germany']
```

Here we loop through all the elements in `country_name_list`, extract the corresponding value from `success_estimates` (which will be of type list), and subsequently take the mean of the list. Detailed printing will guide you through the for loop execution.

[18]:
```
# Loop through all countries and calculate their mean success estimate
for i in country_name_list:
    print('--Begin one iteration of loop--')
    print('Element of country_name_list, placeholder i = ' + i)
    print('Access value from dict success_estimates[i]: ', success_estimates[i])
    print('Average of list from success_estimates[i]: ',␣
 ↪sum(success_estimates[i]) / len(success_estimates[i]))
    print('--Go to next iteration of loop--')
```

```
--Begin one iteration of loop--
Element of country_name_list, placeholder i = Australia
Access value from dict success_estimates[i]:  [0.6, 0.33, 0.11, 0.14]
Average of list from success_estimates[i]:  0.29500000000000004
--Go to next iteration of loop--
--Begin one iteration of loop--
Element of country_name_list, placeholder i = France
Access value from dict success_estimates[i]:  [0.66, 0.78, 0.98, 0.2]
Average of list from success_estimates[i]:  0.655
--Go to next iteration of loop--
--Begin one iteration of loop--
Element of country_name_list, placeholder i = Italy
Access value from dict success_estimates[i]:  [0.6]
Average of list from success_estimates[i]:  0.6
--Go to next iteration of loop--
--Begin one iteration of loop--
Element of country_name_list, placeholder i = Brazil
Access value from dict success_estimates[i]:  [0.22, 0.22, 0.43]
Average of list from success_estimates[i]:  0.29
--Go to next iteration of loop--
--Begin one iteration of loop--
Element of country_name_list, placeholder i = USA
Access value from dict success_estimates[i]:  [0.2, 0.5, 0.3]
Average of list from success_estimates[i]:  0.3333333333333333
--Go to next iteration of loop--
--Begin one iteration of loop--
Element of country_name_list, placeholder i = England
Access value from dict success_estimates[i]:  [0.45]
Average of list from success_estimates[i]:  0.45
--Go to next iteration of loop--
--Begin one iteration of loop--
Element of country_name_list, placeholder i = Canada
Access value from dict success_estimates[i]:  [0.25, 0.3]
Average of list from success_estimates[i]:  0.275
--Go to next iteration of loop--
--Begin one iteration of loop--
Element of country_name_list, placeholder i = Argentina
Access value from dict success_estimates[i]:  [0.22]
Average of list from success_estimates[i]:  0.22
--Go to next iteration of loop--
--Begin one iteration of loop--
Element of country_name_list, placeholder i = Greece
Access value from dict success_estimates[i]:  [0.45, 0.66, 0.75, 0.99, 0.15,
0.66]
Average of list from success_estimates[i]:  0.61
--Go to next iteration of loop--
--Begin one iteration of loop--
Element of country_name_list, placeholder i = Morocco
```

```
Access value from dict success_estimates[i]:  [0.29]
Average of list from success_estimates[i]:  0.29
--Go to next iteration of loop--
--Begin one iteration of loop--
Element of country_name_list, placeholder i = Tunisia
Access value from dict success_estimates[i]:  [0.68, 0.56]
Average of list from success_estimates[i]:  0.6200000000000001
--Go to next iteration of loop--
--Begin one iteration of loop--
Element of country_name_list, placeholder i = Egypt
Access value from dict success_estimates[i]:  [0.99]
Average of list from success_estimates[i]:  0.99
--Go to next iteration of loop--
--Begin one iteration of loop--
Element of country_name_list, placeholder i = Jamaica
Access value from dict success_estimates[i]:  [0.61, 0.65, 0.71]
Average of list from success_estimates[i]:  0.6566666666666666
--Go to next iteration of loop--
--Begin one iteration of loop--
Element of country_name_list, placeholder i = Switzerland
Access value from dict success_estimates[i]:  [0.73, 0.86, 0.84, 0.51, 0.99]
Average of list from success_estimates[i]:  0.7859999999999999
--Go to next iteration of loop--
--Begin one iteration of loop--
Element of country_name_list, placeholder i = Germany
Access value from dict success_estimates[i]:  [0.45, 0.49, 0.36]
Average of list from success_estimates[i]:  0.4333333333333333
--Go to next iteration of loop--
```

Let's take a closer look at the above `for` loop. The `country_name_list` has 15 countries which the `for` loop is iterating over. The `for` loop uses a placeholder variable, denoted `i` in this case, to store the element of `country_name_list` that each loop iteration corresponds to. Namely, for the first iteration of the `for` loop, `i = 'Brazil'`. For the second iteration, `i = 'Canada'`. And so on until the loop reaches the final element of `country_name_list`, which it then completes and exits the looping process.

Why is this looping process useful? Well, we've performed the same calculation statements 15 times while only writing the code once! Notice that for each iteration, the corresponding value from `success_estimates` is accessed, and the mean of the returned list is calculated. The `for` loop process also enhances code readability.

2.6.1 Exercise 4:

Write a for loop to instead calculate the minimum and maximum of each country's list of success estimates, printing each out consecutively as in the for loop example above.

Answer. One possible solution is shown below:

[19]:
```
for i in country_name_list:
    print('Country',i,', Min: ', min(success_estimates[i]))
    print('Country',i,', Max: ', max(success_estimates[i]))
```

```
Country Australia , Min:  0.11
Country Australia , Max:  0.6
Country France , Min:  0.2
Country France , Max:  0.98
Country Italy , Min:  0.6
Country Italy , Max:  0.6
Country Brazil , Min:  0.22
Country Brazil , Max:  0.43
Country USA , Min:  0.2
Country USA , Max:  0.5
Country England , Min:  0.45
Country England , Max:  0.45
Country Canada , Min:  0.25
Country Canada , Max:  0.3
Country Argentina , Min:  0.22
Country Argentina , Max:  0.22
Country Greece , Min:  0.15
Country Greece , Max:  0.99
Country Morocco , Min:  0.29
Country Morocco , Max:  0.29
Country Tunisia , Min:  0.56
Country Tunisia , Max:  0.68
Country Egypt , Min:  0.99
Country Egypt , Max:  0.99
Country Jamaica , Min:  0.61
Country Jamaica , Max:  0.71
Country Switzerland , Min:  0.51
Country Switzerland , Max:  0.99
Country Germany , Min:  0.36
Country Germany , Max:  0.49
```

2.6.2 Exercise 5:

Using the for loop, write code to determine the country with the largest range of success estimates (that is, the largest difference between the smallest and largest estimate for a country).

Answer. One possible solution is shown below:

[20]:
```
for i in country_name_list:
    country_range = max(success_estimates[i]) - min(success_estimates[i])
    print('Country: ', i, ", Range: ", country_range)

# Visualizally analyzing the output, we see Greece has the largest range
```

```
Country:  Australia , Range:  0.49
Country:  France , Range:  0.78
Country:  Italy , Range:  0.0
Country:  Brazil , Range:  0.21
Country:  USA , Range:  0.3
Country:  England , Range:  0.0
Country:  Canada , Range:  0.04999999999999999
Country:  Argentina , Range:  0.0
Country:  Greece , Range:  0.84
Country:  Morocco , Range:  0.0
Country:  Tunisia , Range:  0.12
Country:  Egypt , Range:  0.0
Country:  Jamaica , Range:  0.09999999999999998
Country:  Switzerland , Range:  0.48
Country:  Germany , Range:  0.13
```

2.7 Using list comprehensions to determine the number of estimates for each country

Moving forward, we are interested in knowing the number of success estimates available for each country. Python offers a concise way to achieve this goal through the use of list comprehensions.

List comprehensions allow one to concisely build a list. Let's take a look at how this works.

[21]:
```
key_name_list = [i for i in success_estimates] # loop over each item i in␣
↪success_estimates and put i in the list
key_name_list
```

[21]:
```
['Australia',
 'France',
 'Italy',
 'Brazil',
 'USA',
 'England',
 'Canada',
 'Argentina',
 'Greece',
 'Morocco',
 'Tunisia',
 'Egypt',
 'Jamaica',
 'Switzerland',
 'Germany']
```

Here we see that we've looped over each key of the dictionary success_estimates (hence each country), and extracted the country name, all in one line of code. We can also access the values of each key in success_estimates.

[22]:
```
value_name_list = [success_estimates[i] for i in success_estimates] # loop over
↪each item i in success_estimates and put success_estimates[i] in the list
value_name_list
```

[22]:
```
[[0.6, 0.33, 0.11, 0.14],
 [0.66, 0.78, 0.98, 0.2],
 [0.6],
 [0.22, 0.22, 0.43],
 [0.2, 0.5, 0.3],
 [0.45],
 [0.25, 0.3],
 [0.22],
 [0.45, 0.66, 0.75, 0.99, 0.15, 0.66],
 [0.29],
 [0.68, 0.56],
 [0.99],
 [0.61, 0.65, 0.71],
 [0.73, 0.86, 0.84, 0.51, 0.99],
 [0.45, 0.49, 0.36]]
```

In the list comprehension above, each value of `i` is a country name and the value is returned when `success_estimates[i]` is called. We see the list comprehension is an effective and concise way to write a for loop that creates a list.

We can the use this to quickly determine how many success estimates are available for each country.

[23]:
```
# Number of estimates available for each country
[[i,len(success_estimates[i])] for i in success_estimates]
```

[23]:
```
[['Australia', 4],
 ['France', 4],
 ['Italy', 1],
 ['Brazil', 3],
 ['USA', 3],
 ['England', 1],
 ['Canada', 2],
 ['Argentina', 1],
 ['Greece', 6],
 ['Morocco', 1],
 ['Tunisia', 2],
 ['Egypt', 1],
 ['Jamaica', 3],
 ['Switzerland', 5],
 ['Germany', 3]]
```

2.7.1 Exercise 6:

Using list comprehensions, write a script to create a list of lists called `sum_squares_list`, where each element of the list is a two-item list [country name, value]. The value item in the list should be the sum of squares of that country's success estimates. For example, one element of `sum_squares_list` should be for Jamaica, where the two-item list is [Jamaica, 1.2987] (since 1.2987 = 0.61^2 + 0.65^2 + 0.71^2).

Answer. One possible answer is shown below:

[24]:
```
# One possible solution
sum_squares_list = [[i, sum([j**2 for j in success_estimates[i]])] for i in␣
↪success_estimates]
sum_squares_list
```

[24]:
```
[['Australia', 0.5005999999999999],
 ['France', 2.0444],
 ['Italy', 0.36],
 ['Brazil', 0.28169999999999995],
 ['USA', 0.38],
 ['England', 0.2025],
 ['Canada', 0.1525],
 ['Argentina', 0.0484],
 ['Greece', 2.6388],
 ['Morocco', 0.0841],
 ['Tunisia', 0.7760000000000001],
 ['Egypt', 0.9801],
 ['Jamaica', 1.2987],
 ['Switzerland', 3.2183],
 ['Germany', 0.5722]]
```

2.7.2 Exercise 7:

We'd like to determine the spread around the mean success estimate for each country. Using list comprehensions, write a script that subtracts the mean success estimate for a given country from each success estimate for that country. Store the results in a list named `removed_mean_list`. Round values to two decimal places. Your output should produce the following list of lists:

```
[['Australia', [0.3, 0.03, -0.19, -0.16]],
 ['France', [0.01, 0.12, 0.32, -0.46]],
 ['Italy', [0.0]],
 ['Brazil', [-0.07, -0.07, 0.14]],
 ['USA', [-0.13, 0.17, -0.03]],
 ['England', [0.0]],
 ['Canada', [-0.03, 0.02]],
 ['Argentina', [0.0]],
 ['Greece', [-0.16, 0.05, 0.14, 0.38, -0.46, 0.05]],
 ['Morocco', [0.0]],
```

```
 ['Tunisia', [0.06, -0.06]],
 ['Egypt', [0.0]],
 ['Jamaica', [-0.05, -0.01, 0.05]],
 ['Switzerland', [-0.06, 0.07, 0.05, -0.28, 0.2]],
 ['Germany', [0.02, 0.06, -0.07]]]
```

Answer. One possible answer is shown below:

[25]:
```
# One possible solution
removed_mean_list = [[i, [round(j - sum(success_estimates[i])/
 ↪len(success_estimates[i]),2) for j in success_estimates[i]]] for i in␣
 ↪success_estimates]
removed_mean_list
```

[25]:
```
[['Australia', [0.3, 0.03, -0.19, -0.16]],
 ['France', [0.01, 0.12, 0.32, -0.46]],
 ['Italy', [0.0]],
 ['Brazil', [-0.07, -0.07, 0.14]],
 ['USA', [-0.13, 0.17, -0.03]],
 ['England', [0.0]],
 ['Canada', [-0.03, 0.02]],
 ['Argentina', [0.0]],
 ['Greece', [-0.16, 0.05, 0.14, 0.38, -0.46, 0.05]],
 ['Morocco', [0.0]],
 ['Tunisia', [0.06, -0.06]],
 ['Egypt', [0.0]],
 ['Jamaica', [-0.05, -0.01, 0.05]],
 ['Switzerland', [-0.06, 0.07, 0.05, -0.28, 0.2]],
 ['Germany', [0.02, 0.06, -0.07]]]
```

2.8 Reflecting on the country-specific mean success estimate

Based on the above analysis, we see the mean country success estimates vary widely, from the lowest, Canada = 0.275, to the highest, Egypt = 0.99. However, notice that Egypt's mean is calculated from 1 success estimate. Are we confident in trusting a single estimate as a proxy for the average success estimate?

Given that the global expansion project will utilize valuable company resources, we decide it is best to restrict our analysis to countries that have two or more success estimates. To accomplish this task, we will use a control structure in Python known as the if...elif...else statement. The general structure follows.

```
if test_expression_1:
    block1_statement(s)
elif test_expression_2:
    block2_statement2(s)
else:
    block3_statement(s)
```

Here, `test_expression_1` and `test_expression_2` must evaluate to `True` or `False`, a Python boolean type. The boolean type is associated with variables that are either `True` or `False`.

If `test_expression_1` is True, `block1_statement(s)` will execute and the other block statements will not. If `test_expression_1` is False yet `test_expression_2` is True, then `block2_statement2(s)` will execute and the others will not. Finally, if `test_expression_1` and `test_expression_2` are both False, then the else section's `block3_statement(s)` will execute. This conditional structure of an if statement allows one to control the flow of Python code.

Let's use this to filter out the countries that only have one success estimate.

2.9 Selecting only multi-observation countries for global expansion potential

We will use the if statement above to remove countries with less than one success estimate. For convenience of viewing the result, we will store the mean estimates for each country in a new dictionary `country_means`.

[26]:
```
# Get a list of all the country names
country_name_list = list(success_estimates.keys())

# Create an empty dictionary to hold country mean estimates
country_means = {}

# Loop through all countries and calculate their mean success estimate
for i in country_name_list:
    list_country_estimates = success_estimates[i] # list of estimates for a␣
↪country

    # if more than one country estimate, then record the mean estimate,␣
↪otherwise go to next loop iteration
    if len(success_estimates[i]) > 1:
        country_mean_value = sum(list_country_estimates) /␣
↪len(list_country_estimates)
        country_means[i] =  country_mean_value # insert country mean value into␣
↪dict using country name as key
```

Let's format our results, modifying the string output to the screen to use 2 decimals when printing the float type. This is accomplished using the string type's `.format()` functionality. The `{0:s}` and `{1:.2f}` in the string indicate to the `.format()` method to format the first variable it receives as input as a string and replace the `{0:s}` placeholder, and to format the second variable it receives as input as a 2-decimal float and replace the `{1:.2f}` placeholder.

With this formatting, the `country_key` variable will be displayed as a string in place of `{0:s}`, while the `country_means[country_key]` variable will be displayed as a 2-decimal float in place of `{1:.2f}`. This advanced string formatting approach is useful to improve the clarity of the results.

[27]:
```
# Nicely format the result for printing to the screen
for country_key in country_means:
```

```
    print("Country: {0:s}, Avg Success Estimate: {1:.2f}".format(country_key,␣
↪country_means[country_key]))
```

```
Country: Australia, Avg Success Estimate: 0.30
Country: France, Avg Success Estimate: 0.66
Country: Brazil, Avg Success Estimate: 0.29
Country: USA, Avg Success Estimate: 0.33
Country: Canada, Avg Success Estimate: 0.28
Country: Greece, Avg Success Estimate: 0.61
Country: Tunisia, Avg Success Estimate: 0.62
Country: Jamaica, Avg Success Estimate: 0.66
Country: Switzerland, Avg Success Estimate: 0.79
Country: Germany, Avg Success Estimate: 0.43
```

Observing the resulting country means, we notice the country with the largest mean success estimate is Switzerland at 0.79, while the lowest mean success estimate is Canada at 0.28.

2.9.1 Exercise 8:

After reviewing company policy on statistical procedures, you notice the company recommends that all estimates (averages, minimums, maximums) must have at least three values contributing to the summary statistic. Write a for loop and use the `if` statement structure to select and print the average success estimates for the countries satisfying this policy. If the country does not satisfy the policy, print the country name and `"*Does not meet company policy*"`. Each country should appear on a new line.

Answer. One possible answer is shown below:

[28]:
```
# Get a list of all the country names
country_name_list = list(success_estimates.keys())

# Create an empty dictionary to hold country mean estimates
country_means = {}

# Loop through all countries and calculate their mean success estimate
for i in country_name_list:
    list_country_estimates = success_estimates[i] # list of estimates for a␣
↪country

    # if more than one country estimate, then record the mean estimate,␣
↪otherwise go to next loop iteration
    if len(success_estimates[i]) > 2:
        country_mean_value = sum(list_country_estimates) /␣
↪len(list_country_estimates)
        country_means[i] =  country_mean_value # insert country mean value into␣
↪dict using country name as key
```

```
        print("Country: {0:s}, Avg Success Estimate: {1:.2f}".format(i,␣
↪country_mean_value))
    else:
        print("Country: {0:s}, *Does not Meet Company Policy*".format(i))
```

```
Country: Australia, Avg Success Estimate: 0.30
Country: France, Avg Success Estimate: 0.66
Country: Italy, *Does not Meet Company Policy*
Country: Brazil, Avg Success Estimate: 0.29
Country: USA, Avg Success Estimate: 0.33
Country: England, *Does not Meet Company Policy*
Country: Canada, *Does not Meet Company Policy*
Country: Argentina, *Does not Meet Company Policy*
Country: Greece, Avg Success Estimate: 0.61
Country: Morocco, *Does not Meet Company Policy*
Country: Tunisia, *Does not Meet Company Policy*
Country: Egypt, *Does not Meet Company Policy*
Country: Jamaica, Avg Success Estimate: 0.66
Country: Switzerland, Avg Success Estimate: 0.79
Country: Germany, Avg Success Estimate: 0.43
```

2.9.2 Exercise 9:

What is another approach to ameliorate the one-sample problem for some countries? Think in terms of the factors that drive confidence in data-driven business decisions.

(a) Group countries together to larger regions to ensure each region has at least one estimate

(b) Only remove a country if its esimates are very large or very small compared to other estimates

(c) Use a different summary statistic for the analysis other than the average value

(d) Revisit why some countries only have one estimate and see if more data can be sourced for these countries

Answer. (d).

Answer (a) sounds like a good approach, however it will not accomplish the task at hand. The task is to find a specific country for global expansion, not a region. Thus, if one were to regroup countries to regions, they would first have to redefine the problem statement.

Answer (b) is an incorrect method to handle this particular data set since country statistics should be calculated independently of other countries. If one countries estimates depend on another, this could significantly complicate the analysis and would likely require a more complex and costly investigation.

Regarding answer (c), using a different summary statistic will not solve the problem of having one sample for the statistical calculation. We would like to have more confidence in our statistical estimate. Using a different measure of success such as the minimum or maximum estimate will not ameliorate the single-sample size issue.

It is common to revisit the source of the data set for a problem as an iterative analysis is performed. Hence, answer (d) is a correct approach to ameliorate the one-sample size problem. As new insight is gained throughout the analysis, one should always reflect on the data collection process and determine if any new information may help the analysis to move forward. In this case, upon observing some countries having one estimate, it would likely be beneficial to determine why some countries have such small numbers of estimates and if there is a simple, cost-effective method to obtain additional estimates for these countries before moving forward in the analysis.

2.10 Putting it all together

We've used for loops and control structures to calculate partial summary statistics for each of the countries. Let's put it all together to obtain a recommendation on which country we should choose to expand the luxury flight services.

2.10.1 Exercise 10:

Write code to print each country name and summary statistics. Each line should show one country and the corresponding summary statistics: Min Estimate (float), Average Estimate (float), Max Estimate (float), Number of Estimates (int), Meets Company Policy of at least 3 estimates (bool). For example, the line for France would appear as:

```
Country: France , Min: 0.2 , Average: 0.655 , Max: 0.98 , NumEst: 4 , MeetsPolicy: True
```

Answer. One possible solution is shown below:

[29]:
```
country_name_list = list(success_estimates.keys())
for i in country_name_list:
    min_stat = min(success_estimates[i])
    mean_stat = sum(success_estimates[i]) / len(success_estimates[i])
    max_stat = max(success_estimates[i])
    len_stat = len(success_estimates[i])
    meets_policy = len_stat > 2
    print('Country:',i,', Min:',min_stat,', Average:',mean_stat,', Max:
 ↪',max_stat,', NumEst:',len_stat,', MeetsPolicy:',meets_policy)
```

```
Country: Australia , Min: 0.11 , Average: 0.29500000000000004 , Max: 0.6 ,
NumEst: 4 , MeetsPolicy: True
Country: France , Min: 0.2 , Average: 0.655 , Max: 0.98 , NumEst: 4 ,
MeetsPolicy: True
Country: Italy , Min: 0.6 , Average: 0.6 , Max: 0.6 , NumEst: 1 , MeetsPolicy:
False
Country: Brazil , Min: 0.22 , Average: 0.29 , Max: 0.43 , NumEst: 3 ,
MeetsPolicy: True
Country: USA , Min: 0.2 , Average: 0.3333333333333333 , Max: 0.5 , NumEst: 3 ,
MeetsPolicy: True
Country: England , Min: 0.45 , Average: 0.45 , Max: 0.45 , NumEst: 1 ,
MeetsPolicy: False
Country: Canada , Min: 0.25 , Average: 0.275 , Max: 0.3 , NumEst: 2 ,
```

```
MeetsPolicy: False
Country: Argentina , Min: 0.22 , Average: 0.22 , Max: 0.22 , NumEst: 1 ,
MeetsPolicy: False
Country: Greece , Min: 0.15 , Average: 0.61 , Max: 0.99 , NumEst: 6 ,
MeetsPolicy: True
Country: Morocco , Min: 0.29 , Average: 0.29 , Max: 0.29 , NumEst: 1 ,
MeetsPolicy: False
Country: Tunisia , Min: 0.56 , Average: 0.6200000000000001 , Max: 0.68 , NumEst:
2 , MeetsPolicy: False
Country: Egypt , Min: 0.99 , Average: 0.99 , Max: 0.99 , NumEst: 1 ,
MeetsPolicy: False
Country: Jamaica , Min: 0.61 , Average: 0.6566666666666666 , Max: 0.71 , NumEst:
3 , MeetsPolicy: True
Country: Switzerland , Min: 0.51 , Average: 0.7859999999999999 , Max: 0.99 ,
NumEst: 5 , MeetsPolicy: True
Country: Germany , Min: 0.36 , Average: 0.4333333333333333 , Max: 0.49 , NumEst:
3 , MeetsPolicy: True
```

Now that we have summary statistics for a variety of country estimates, let's construct our actionable business recommendation.

2.11 Conclusions

From the analysis, Switzerland is the country with the largest chance of success for global expansion with an estimated success rate of 0.79. The summary statistic for Switzerland was calculated with an adequate number of estimates according to company policy. Thus, it is recommended that management explore opportunities in Switzerland for luxury flight services.

In addition, other countries that should be closely monitored for luxury flight services are Jamaica and France, where each held a 0.66 average success estimate. If there are additional resources in the future for further luxury service expansion, these countries may be apt choices.

2.12 Takeaways

In this case, we've learned the foundations of Python. Through identifying global expansion opportunities for an airline company we've covered fundamental data types, control structures, and a useful Python workflow to analyze a given set of data. You also learned about various summary statistics and how much confidence you can have in drawing conclusions from them.

Building on this knowledge, you can use these Python tools as both a foundation and a framework to build more complex projects and solve critical business problems. Python continues to be an outstanding tool to perform data-driven analysis and deliver key business insights.

[ ]:

```
import numpy as np
import matplotlib.pyplot as plt
```

1.2 Preliminary cleaning of the data

Before we can proceed with data analysis and modeling, we first need to determine if the relevant data is adequate to proceed as-is, or if it needs further cleaning. In this case, we have received a Comma Separated Value (CSV) file that includes the following data:

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
8. **Symbol:** The symbol for that particular stock

One very common problem that arises in datasets is missing values. Let's see how to identify whether or not our dataset has this problem, and how to deal with missing values.

[2]:
```
# Load and view head of DataFrame
raw_df = pd.read_csv('EnergySectorData.csv')
raw_df.head()
```

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

      Volume Symbol
0  1806400.0      D
1  2231100.0      D
2  2588900.0      D
3  3266900.0      D
4  2601800.0      D
```

Let's begin by determining if we have missing values. We can use the `pandas` DataFrame method `isnull()` to check for `NaN` values in `raw_df` (i.e. check for null values):

[3]:
```
# Check if there are missing values (NaNs)
raw_df.isnull().sum()
```

[3]:
```
Date          0
Open          2
```

```
High          14
Low            7
Close          7
Adj Close      7
Volume        22
Symbol         0
dtype: int64
```

Here we see we have some missing values. Let's instead use the `mean()` method to determine what percent of each column we are missing.

[4]:
```
raw_df.isnull().mean()
```

[4]:
```
Date         0.000000
Open         0.000319
High         0.002231
Low          0.001116
Close        0.001116
Adj Close    0.001116
Volume       0.003506
Symbol       0.000000
dtype: float64
```

We see here that we are missing less than 0.5% of the observations in any given column.

1.2.1 Exercise 1:

We do not want any missing values in our analysis. Which of the following options is the WORST option for how to proceed in this case?

(a) Fill in any day's missing value with the previous day's value

(b) Replace the missing values by re-gathering the data

(c) Estimate the missing values by interpolating them from the values of other, similar data points

(d) Remove rows from the dataset that contain missing values

Answer. (a). We have a couple options that are generally acceptable on how to proceed with missing values:

1. One option is to replace the missing values by re-gathering the data. However, this option is often quite expensive in real man-hours, so we will forgo it for now.
2. Another option is to try to estimate the missing values using some reasonable estimation method, interpolating from other data points. However, this can be complicated and given that such a little amount of our data is missing, we will forgo this option.
3. In practice, a regularly chosen option when only a small amount of data is missing is to just remove the rows that have missing data. This option is generally fine to perform so long as

the removed data is an insignificant portion of the data under study. Here we will choose this option as it simplifies the analysis and should not harm any results moving forward.

Answer (a) is problematic because replacing a missing value with the previous day's value doesn't make sense for stocks because stock prices and trading volumes are known to move day-to-day rather than remain unchanged for an extended period of time. Since we will be dealing with daily returns and volatility, this is especially problematic as it defaults any missing day's volatility and return to 0.

Note that these options for cleaning data should be carefully weighed when commencing a new data science study.

Let's clean the missing values by removing them from the data set.

[5]:
```
# Remove NaNs from data
# Drop the missing values
progress_df = raw_df.dropna()
```

1.3 Standardizing dates

We'd like to be able to analyze these stocks together across time. This would be easier is if all the stocks contained non-missing data for the same set of dates. Let's first ascertain if this is the case. One way to do this is to use the `groupby` method to group by `Date`, then use the `count()` function to enumerate how many distinct dates we have. Since there are a total of 1259 rows per symbol, there should be a count of 1259 for each symbol.

[6]:
```
# How many data rows do we have for each Symbol
progress_df.groupby('Symbol').count()
```

[6]:

Symbol	Date	Open	High	Low	Close	Adj Close	Volume
D	1230	1230	1230	1230	1230	1230	1230
DUK	1249	1249	1249	1249	1249	1249	1249
EXC	1239	1239	1239	1239	1239	1239	1239
NEE	1251	1251	1251	1251	1251	1251	1251
SO	1259	1259	1259	1259	1259	1259	1259

Since most symbols do not have a count of 1259 for their `Date` columns, there are clearly some inconsistent values. Some of these duplicates will be missing values (NaNs), so let's enumerate those again first:

[7]:
```
# Check if there are missing values (NaNs)
raw_df.isnull().sum()
```

[7]:
```
Date          0
Open          2
High         14
Low           7
Close         7
```

```
Adj Close     7
Volume       22
Symbol        0
dtype: int64
```

As discussed earlier, we can remove missing values as there are not many samples that are missing, and dropping a small number of dates is not expected to significantly impact the analysis:

[8]:
```
# Drop the missing values
progress_df = raw_df.dropna().copy()
```

[9]:
```
# How many data rows do we have for each Symbol
progress_df.groupby('Symbol').count()
```

[9]:

Symbol	Date	Open	High	Low	Close	Adj Close	Volume
D	1230	1230	1230	1230	1230	1230	1230
DUK	1249	1249	1249	1249	1249	1249	1249
EXC	1239	1239	1239	1239	1239	1239	1239
NEE	1251	1251	1251	1251	1251	1251	1251
SO	1259	1259	1259	1259	1259	1259	1259

Still, we see that different symbols have different numbers of dates. We'd like all the symbols to have the same set of dates for analysis purposes. Let's create a new `clean_df` that corresponds to a DataFrame with the same number of rows for each `Symbol`, where all symbols share the same set of dates:

[10]:
```
set_dates_D = set(progress_df[progress_df['Symbol'] == 'D']['Date'])
set_dates_EXC = set(progress_df[progress_df['Symbol'] == 'EXC']['Date'])
set_dates_NEE = set(progress_df[progress_df['Symbol'] == 'NEE']['Date'])
set_dates_SO = set(progress_df[progress_df['Symbol'] == 'SO']['Date'])
set_dates_DUK = set(progress_df[progress_df['Symbol'] == 'DUK']['Date'])
set_unique_dates = set.
 ↪intersection(set_dates_D,set_dates_EXC,set_dates_NEE,set_dates_SO,set_dates_DUK)
```

[11]:
```
# Filter new DataFrame for only the dates that are present in every symbol (i.e.
 ↪ the overlapping dates)
clean_df = progress_df[progress_df['Date'].isin(set_unique_dates)].copy()
```

[12]:
```
# Let's take a look
clean_df.groupby('Symbol').count()
```

[12]:

Symbol	Date	Open	High	Low	Close	Adj Close	Volume
D	1192	1192	1192	1192	1192	1192	1192
DUK	1192	1192	1192	1192	1192	1192	1192
EXC	1192	1192	1192	1192	1192	1192	1192
NEE	1192	1192	1192	1192	1192	1192	1192

```
SO      1192  1192  1192  1192   1192       1192    1192
```

Now we see that each symbol has the same number of unique dates. Let's write a quick verification program to ensure the resulting `clean_df` does indeed have the same dates for every symbol.

1.3.1 Exercise 2:

Write code to ensure that each of the symbols share the same set of unique dates. (Hint: use the `set()` method.)

Answer. One possible solution is given below:

[13]:
```
# One possible solution
check_set_dates_D = set(clean_df[clean_df['Symbol'] == 'D']['Date'])
check_set_dates_EXC = set(clean_df[clean_df['Symbol'] == 'EXC']['Date'])
check_set_dates_NEE = set(clean_df[clean_df['Symbol'] == 'NEE']['Date'])
check_set_dates_SO = set(clean_df[clean_df['Symbol'] == 'SO']['Date'])
check_set_dates_DUK = set(clean_df[clean_df['Symbol'] == 'DUK']['Date'])

print(check_set_dates_D == check_set_dates_EXC)
print(check_set_dates_D == check_set_dates_NEE)
print(check_set_dates_D == check_set_dates_SO)
print(check_set_dates_D == check_set_dates_DUK)
```

```
True
True
True
True
```

Now that we've completed the preliminary cleaning of the data, let's move forward with determining the relationships between: (1) stock returns and volatility, and (2) stock returns and broader market returns.

1.4 Adding additional variables required for our analysis

Recall that the original question requires us to investigate both daily stock returns and the volatility of those returns. This means that important measures of interest are:

1. Daily (open to close) stock return
2. Volatility of daily stock return

Why are each of these important?

1. Volatility: Gives insight into amount of price movement in any given day. Volatility is directly related to the level of risk involved in holding the stock.
2. Return: Gives us an idea of the return on investment over a period of time.

Let's calculate these statistics and add them to the DataFrame `clean_df`:

[14]:
```
clean_df['VolStat'] = (clean_df['High'] - clean_df['Low']) / clean_df['Open']
clean_df['Return'] = (clean_df['Close'] / clean_df['Open']) - 1.0
clean_df['Volume_Millions'] = clean_df['Volume'] / 1000000.0 # Volume in␣
↪Millions (added for convenience)
clean_df.head()
```

[14]:

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

	Volume	Symbol	VolStat	Return	Volume_Millions
0	1806400.0	D	0.018781	0.016201	1.8064
1	2231100.0	D	0.014858	-0.010471	2.2311
2	2588900.0	D	0.032286	-0.014714	2.5889
3	3266900.0	D	0.018505	-0.014425	3.2669
4	2601800.0	D	0.017674	0.003861	2.6018

Here we see that we've added three columns to `clean_df`, namely `VolStat`, `Return`, and `Volume_Millions` (the last one is just for convenience, as the values in the `Volume` column are quite large).

Since we are looking to analyze the relationship between daily volatility and returns, an additional column that makes sense to add is one which says `True` when the daily return is positive, and `False` when the daily return negative. We can then group days into positive and negative return cohorts and compare the average volatility on those days. We will name this column `ReturnFlag`.

We can accomplish this using an **anonymous function**; that is, a function that is defined but not named:

```
lambda arguments: expression
```

The `lambda` keyword tells Python that we are using an anonymous function. Next, the `arguments` are the name we give to the inputs. It can be `x`, or `y`, or whatever the user would like to call it. In this case we will use the name `row` for the input argument name as the input will indeed be a row of a DataFrame. The `expression` is what is then applied to the `arguments`; this is the function.

Let's take a look at how we can use anonymous functions to create the `ReturnFlag` feature:

[15]:
```
clean_df['ReturnFlag'] = clean_df.apply(lambda row: True if row['Return'] > 0␣
↪else False, axis=1) # Volume in Millions
clean_df.head()
```

[15]:

	Date	Open	High	Low	Close	Adj Close	\
0	2014-07-28	69.750000	71.059998	69.750000	70.879997	57.963978	
1	2014-07-29	70.669998	70.980003	69.930000	69.930000	57.187099	
2	2014-07-30	70.000000	70.660004	68.400002	68.970001	56.402020	
3	2014-07-31	68.629997	68.849998	67.580002	67.639999	55.314388	

```
4  2014-08-01  67.330002  68.410004  67.220001  67.589996  55.273487

      Volume Symbol   VolStat    Return  Volume_Millions  ReturnFlag
0  1806400.0      D  0.018781  0.016201           1.8064        True
1  2231100.0      D  0.014858 -0.010471           2.2311       False
2  2588900.0      D  0.032286 -0.014714           2.5889       False
3  3266900.0      D  0.018505 -0.014425           3.2669       False
4  2601800.0      D  0.017674  0.003861           2.6018        True
```

Notice that the `apply()` method takes in an anonymous function and applies it to the rows of the DataFrame through the use of the second argument `axis`. `axis=0` applies the function to columns, whereas `axis=1` applies the function to rows.

So what is happening in the following statement?

```
clean_df['ReturnFlag'] = clean_df.apply(lambda row: True if row['Return] > 0 else False, axis=
```

1. `pandas` recogized through the `apply` method that it is operating on the `clean_df` DataFrame
2. The `apply` method takes a function as input that will be applied to the DataFrame `clean_df`
3. Given the second argument of `apply` is `axis=1` the input into the anonymous function is a single row
4. For every row, `row['Return']` returns the `Return` value for that row, and it is subsequently passed through the if statement, returning True if greater than zero and False otherwise
5. The new value is stored in the column `clean_df['ReturnFlag']`

1.4.1 Exercise 3:

Using `apply()` and `lambda`, write code to create a new column named `YYYY` to `clean_df`, where the new column is the year of the observation as a string. For instance if the row `Date` value is 2014-07-28, then the value in the new column for the year would be '2014'. Recall that you can access the first 4 characters of some string `my_string` using `my_string[:4]`.

Answer. One possible solution is given below:

[16]:
```
# One possible solution
clean_df['YYYY'] = clean_df.apply(lambda row: row['Date'][:4], axis=1)
```

Let's move forward with labelling volatility regimes present in the data – these regimes are useful for breaking down the stock return analysis by periods of low, medium, and high volatility. It will allow for more granular analysis than just looking at overall averages without a breakdown.

1.4.2 Exercise 4:

Using `apply()` and `lambda`, write a script to create a new column in `clean_df` named `AvgDailyPrice` that calculates an average daily price based on whether or not the daily volume is over 5 million. Set the value of the new column to (Open + High + Low + Close)/4 if the volume is larger than 5 million, or set the value to (High + Low + Close)/3 if the volume is less than or equal to 5 million.

Answer. One possible solution is given below:

[17]:
```
# One possible solution
clean_df['AvgDailyPrice'] = clean_df.apply(lambda row:
 ↪(row['Open']+row['High']+row['Low']+row['Close'])/4 if row['Volume'] >␣
 ↪5000000 else (row['High']+row['Low']+row['Close'])/3, axis=1)
clean_df.head()
```

[17]:

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

	Volume	Symbol	VolStat	Return	Volume_Millions	ReturnFlag	YYYY	\
0	1806400.0	D	0.018781	0.016201	1.8064	True	2014	
1	2231100.0	D	0.014858	-0.010471	2.2311	False	2014	
2	2588900.0	D	0.032286	-0.014714	2.5889	False	2014	
3	3266900.0	D	0.018505	-0.014425	3.2669	False	2014	
4	2601800.0	D	0.017674	0.003861	2.6018	True	2014	

	AvgDailyPrice
0	70.563332
1	70.280001
2	69.343336
3	68.023333
4	67.740000

1.5 Labelling energy sector volatility regimes

Similar to Case 1.2, we'd like to label periods of low and high volatilty in a new column called `VolLevel` for each Symbol using some lower and upper bound values. For example, in the case of the Symbol D we'd like to have a new column with value determined by:

```
if VolStrat > upper_threshold_dict['D']:
    VolLevel = '3_HIGH'
elif VolStrat < lower_threshold_dict['D']:
    VolLevel = '1_LOW'
else:
    VolLevel = '2_MEDIUM'
```

Namely, this labelling should be applied to each row, and the threshold values should correspond to the Symbol for that row. We will group by this column and see if we can find any new insights in the data.

[18]:
```
# Determine lower bounds (we choose to use 25th percentile)
```

```
lower_threshold_dict = clean_df.groupby('Symbol')['VolStat'].quantile(0.25).
 ↪to_dict() # 25th percentile bound
lower_threshold_dict
```

[18]:
```
{'D': 0.010240046986389077,
 'DUK': 0.010018315803797114,
 'EXC': 0.011881680089172456,
 'NEE': 0.010258642787424582,
 'SO': 0.009734019893739424}
```

[19]:
```
# Determine upper bounds (we choose to use 75th percentile)
upper_threshold_dict = clean_df.groupby('Symbol')['VolStat'].quantile(0.75).
 ↪to_dict() # 75th percentile bound
upper_threshold_dict
```

[19]:
```
{'D': 0.017960914526108228,
 'DUK': 0.017598380774085175,
 'EXC': 0.021801523265676366,
 'NEE': 0.01768021802425081,
 'SO': 0.016830447068579304}
```

Again, our goal is to label low, medium, and high volatility periods. Let's define a new column called `VolLevel` for each Symbol using some lower and upper bound values. Let's define a custom function that will be applied to each row to achieve this goal.

[20]:
```
# Our custom function, input is a row from the agg_df, and the output is a␣
 ↪string, either LOW, MEDIUM, or HIGH
def my_custom_row_function(row):
    row_symbol = row['Symbol']    # the Symbol value in the row
    row_volstat = row['VolStat']  # the VolStat value in the row

    lower_threshold = lower_threshold_dict[row_symbol] # Dictionary of {string:
 ↪float}
    upper_threshold = upper_threshold_dict[row_symbol] # Dictionary of {string:
 ↪float}

    # The function decision, return value depending on low, medium, or high␣
 ↪volatility
    if row_volstat > upper_threshold:
        return '3_HIGH'
    elif row_volstat < lower_threshold:
        return '1_LOW'
    else:
        return '2_MEDIUM'
```

Let's now apply the function to each row of the DataFrame `clean_df` using a `lambda` statement. We store the returned values in the new column `VolLevel`:

[21]:
```
# Apply my_custom_row_function to the Pandas DataFrame, row by row (axis=1)
clean_df['VolLevel'] = clean_df.apply(lambda row: my_custom_row_function(row),␣
 ↪axis=1)
clean_df.head()
```

[21]:

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

	Volume	Symbol	VolStat	Return	Volume_Millions	ReturnFlag	YYYY	\
0	1806400.0	D	0.018781	0.016201	1.8064	True	2014	
1	2231100.0	D	0.014858	-0.010471	2.2311	False	2014	
2	2588900.0	D	0.032286	-0.014714	2.5889	False	2014	
3	3266900.0	D	0.018505	-0.014425	3.2669	False	2014	
4	2601800.0	D	0.017674	0.003861	2.6018	True	2014	

	AvgDailyPrice	VolLevel
0	70.563332	3_HIGH
1	70.280001	2_MEDIUM
2	69.343336	3_HIGH
3	68.023333	3_HIGH
4	67.740000	2_MEDIUM

While the workflow here may seem complex at first, the ability to apply custom functions, group by certain features, and construct summary statistics will prove invaluable as you progress onto more advanced analyses.

1.5.1 Exercise 5:

Using `clean_df` and a `lambda` statement within `apply()`, write a function `new_custom_function()` and a script to add a new column to the DataFrame (call it `EnhancedVolLevel`) that operates in a similar manner as VolLevel but instead gives five volatility level categories using the following logic to determine the label for the volatility level:

```
if VolStrat > 90th percentile:
    VolLevel = '5_VERY_HIGH'
elif VolStrat > 75th percentile:
    VolLevel = '4_HIGH'
elif VolStrat > 25th percentile:
    VolLevel = '3_MEDIUM'
elif VolStrat > 10th percentile:
    VolLevel = '2_LOW'
else:
    VolLevel = '1_VERY_LOW'
```

Remember each percentile should be calculated by symbol. Use these new labels to see if there are any patterns between volatility levels and the direction of returns. Produce the DataFrame to be able to run the following command:

```
clean_df.groupby(['Symbol','EnhancedVolLevel'])['ReturnFlag'].mean()
```

Answer. One possible solution is given below:

[22]:
```
# One possible solution
def new_custom_function(row):
    row_symbol = row['Symbol']    # the Symbol value in the row
    row_volstat = row['VolStat']  # the VolStat value in the row

    very_lower_threshold = very_lower_threshold_dict[row_symbol] # Dictionary␣
 ↪of {string:float}
    lower_threshold = lower_threshold_dict[row_symbol] # Dictionary of {string:
 ↪float}
    upper_threshold = upper_threshold_dict[row_symbol] # Dictionary of {string:
 ↪float}
    very_upper_threshold = very_upper_threshold_dict[row_symbol] # Dictionary␣
 ↪of {string:float}

    # The function decision, return value depending on very low, low, medium,␣
 ↪high, or very high volatility
    if row_volstat > very_upper_threshold:
        return '5_VERY_HIGH'
    elif row_volstat > upper_threshold:
        return '4_HIGH'
    elif row_volstat > lower_threshold:
        return '3_MEDIUM'
    elif row_volstat > very_lower_threshold:
        return '2_LOW'
    else:
        return '1_VERY_LOW'

# Find thresholds
very_upper_threshold_dict = clean_df.groupby('Symbol')['VolStat'].quantile(0.
 ↪90).to_dict() # 90th percentile bound
upper_threshold_dict = clean_df.groupby('Symbol')['VolStat'].quantile(0.75).
 ↪to_dict() # 75th percentile bound
lower_threshold_dict = clean_df.groupby('Symbol')['VolStat'].quantile(0.25).
 ↪to_dict() # 25th percentile bound
very_lower_threshold_dict = clean_df.groupby('Symbol')['VolStat'].quantile(0.
 ↪10).to_dict() # 10th percentile bound

# Calculate and add new column
clean_df['EnhancedVolLevel'] = clean_df.apply(lambda row:␣
 ↪new_custom_function(row), axis=1)
```

```
print(clean_df.head())

# Result
clean_df.groupby(['Symbol','EnhancedVolLevel'])['ReturnFlag'].mean()
```

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

      Volume Symbol   VolStat    Return  Volume_Millions  ReturnFlag  YYYY  \
0  1806400.0      D  0.018781  0.016201           1.8064        True  2014
1  2231100.0      D  0.014858 -0.010471           2.2311       False  2014
2  2588900.0      D  0.032286 -0.014714           2.5889       False  2014
3  3266900.0      D  0.018505 -0.014425           3.2669       False  2014
4  2601800.0      D  0.017674  0.003861           2.6018        True  2014

   AvgDailyPrice  VolLevel EnhancedVolLevel
0      70.563332    3_HIGH           4_HIGH
1      70.280001  2_MEDIUM         3_MEDIUM
2      69.343336    3_HIGH      5_VERY_HIGH
3      68.023333    3_HIGH           4_HIGH
4      67.740000  2_MEDIUM         3_MEDIUM
```

[22]:
```
Symbol  EnhancedVolLevel
D       1_VERY_LOW          0.508333
        2_LOW               0.573034
        3_MEDIUM            0.558725
        4_HIGH              0.528090
        5_VERY_HIGH         0.408333
DUK     1_VERY_LOW          0.550000
        2_LOW               0.522472
        3_MEDIUM            0.558725
        4_HIGH              0.511236
        5_VERY_HIGH         0.433333
EXC     1_VERY_LOW          0.533333
        2_LOW               0.533708
        3_MEDIUM            0.508389
        4_HIGH              0.533708
        5_VERY_HIGH         0.500000
NEE     1_VERY_LOW          0.566667
        2_LOW               0.550562
        3_MEDIUM            0.567114
        4_HIGH              0.522472
        5_VERY_HIGH         0.500000
```

```
SO      1_VERY_LOW          0.516667
        2_LOW               0.511236
        3_MEDIUM            0.530201
        4_HIGH              0.500000
        5_VERY_HIGH         0.516667
Name: ReturnFlag, dtype: float64
```

We see that volatility and stock returns do not exhibit any strong patterns in terms of the average return direction (positive or negative) for a given volatility regime.

1.6 Comparing stock returns to broader market returns

Now, let's look into the second part of your boss's question: what is the relationship between broader market returns and the returns of these five energy stocks? The S&P 500 Index is a stock index comprised of about 500 large-capitalization public US companies. The index is often used as a representation of the US stock market. If we can determine whether or not there exists a strong relationship between these 5 energy stocks' returns and those of the S&P 500 Index, we can determine if there are signficant idiosyncratic characteristics at play among the energy sector stock returns, or if the returns are solely driven by the broader market.

Market returns for the tradable S&P 500 index ETF (exchange-traded fund) are available in `SPY.csv` (the "stock" symbol of the ETF is SPY). Let's load the data and append the daily returns onto the cleaned energy sector data:

[23]:
```
# Load file into DataFrame
market_df = pd.read_csv('SPY.csv')
```

[24]:
```
market_df['Symbol'] = 'SPY' # add column for symbol
market_df['Return'] = (market_df['Close'] / market_df['Open']) - 1.0 #␣
 ↪calculate return
market_df['VolStat'] = (market_df['High'] - market_df['Low']) /␣
 ↪market_df['Open']
market_df.head()
```

[24]:

	Date	Open	High	Low	Close	Adj Close	Volume	Symbol	Return	VolStat
0	2014-08-18	196.800003	197.449997	196.690002	197.360001	178.729111	75424000	SPY	0.002846	0.003862
1	2014-08-19	197.839996	198.539993	197.440002	198.389999	179.661896	59135000	SPY	0.002780	0.005560
2	2014-08-20	198.119995	199.160004	198.080002	198.919998	180.141846	72763000	SPY	0.004038	0.005451
3	2014-08-21	199.089996	199.759995	198.929993	199.500000	180.667160	67791000	SPY	0.002059	0.004169
4	2014-08-22	199.339996	199.690002	198.740005	199.190002	180.386368	76107000	SPY	-0.000752	0.004766

We'd like to merge the market returns in `market_df` and the energy stock data in `clean_df`. This can be accomplished using `pd.merge()` - a versatile method to join together DataFrames.

For those of you familiar with SQL, merging and joining DataFrames can be accomplished in much the same way that SQL accomplishes these tasks (if you are not familiar with SQL, don't worry – we will cover it in later cases). In this case, we'd like to use the intersection of dates in `clean_df` and `market_df` dates as the indices in the merge (i.e. in SQL parlance, we will perform an `inner` merge):

[25]:
```
# Merge inner (merge market_df onto clean_df using the dates of clean_df as the␣
↪keys)
merged_df = pd.merge(clean_df, market_df[['Date','Return']], how='inner',␣
↪on='Date', suffixes=('','_SPY'))
```

[26]:
```
# Check how many dates are in the intersection
merged_df.groupby('Symbol')['Date'].count()
```

[26]:
```
Symbol
D      1178
DUK    1178
EXC    1178
NEE    1178
SO     1178
Name: Date, dtype: int64
```

[27]:
```
merged_df.head()
```

[27]:

	Date	Open	High	Low	Close	Adj Close	\
0	2014-08-18	68.970001	69.250000	68.559998	68.680000	56.164867	
1	2014-08-18	31.930000	32.099998	31.730000	31.820000	26.593485	
2	2014-08-18	96.589996	97.180000	95.889999	96.139999	82.166077	
3	2014-08-18	43.529999	43.720001	43.279999	43.380001	34.210857	
4	2014-08-18	72.300003	72.650002	71.919998	72.089996	58.080738	

	Volume	Symbol	VolStat	Return	Volume_Millions	ReturnFlag	YYYY	\
0	1375600.0	D	0.010004	-0.004205	1.3756	False	2014	
1	4037400.0	EXC	0.011588	-0.003445	4.0374	False	2014	
2	1098200.0	NEE	0.013355	-0.004659	1.0982	False	2014	
3	2979800.0	SO	0.010108	-0.003446	2.9798	False	2014	
4	1826000.0	DUK	0.010097	-0.002905	1.8260	False	2014	

	AvgDailyPrice	VolLevel	EnhancedVolLevel	Return_SPY
0	68.829999	1_LOW	2_LOW	0.002846
1	31.883333	1_LOW	2_LOW	0.002846
2	96.403333	2_MEDIUM	3_MEDIUM	0.002846
3	43.460000	2_MEDIUM	3_MEDIUM	0.002846
4	72.219999	2_MEDIUM	3_MEDIUM	0.002846

1.6.1 Exercise 6:

Using only Symbol D in `clean_df` and `market_df`, use `pd.merge()` to determine how many dates are in `clean_df` that are not in `market_df`. Additionally, how many dates are in `market_df` that are not in `clean_df`? (Hint: isnull() may be useful to simplify the solution.)

Answer. One possible solution is given below:

[28]:
```
# One possible solution

outer_df = pd.merge(clean_df[clean_df['Symbol'] == 'D'][['Date','Return']],␣
 ↪market_df[['Date','Return']], how='outer', on='Date', suffixes=('','_SPY'))
print('--- outer merge HEAD ---')
print(outer_df.head())
print('--- outer merge TAIL ---')
print(outer_df.tail())
print('--- outer merge NaN count ---')
outer_df.isnull().sum()

# Answer to: For Symbol D, how many dates are in clean_df that are not in␣
 ↪market_df? -> 14
# Answer to: For Symbol D, how many dates are in market_df that are not in␣
 ↪clean_df? -> 81
```

```
--- outer merge HEAD ---
         Date    Return  Return_SPY
0  2014-07-28  0.016201         NaN
1  2014-07-29 -0.010471         NaN
2  2014-07-30 -0.014714         NaN
3  2014-07-31 -0.014425         NaN
4  2014-08-01  0.003861         NaN
--- outer merge TAIL ---
            Date  Return  Return_SPY
1268  2019-08-12     NaN   -0.006518
1269  2019-08-13     NaN    0.016716
1270  2019-08-14     NaN   -0.014476
1271  2019-08-15     NaN   -0.000807
1272  2019-08-16     NaN    0.008273
--- outer merge NaN count ---
```

[28]:
```
Date           0
Return        81
Return_SPY    14
dtype: int64
```

1.6.2 Exercise 7:

Use `market_df` and `clean_df` to create a new DataFrame `modified_clean_df` that is the same as `clean_df` but with a new column named `MeanMonthSPYReturn`. Each value in the new column should be the mean monthly return for SPY for the given month of each row's `Date`. The output for the first and last rows of the `modified_clean_df` DataFrame should produce the following output from head(1) and tail(1):

```
print(modified_clean_df[['Date','Symbol','YYYYMM','MeanMonthSPYReturn']].head(1))
```

```
            Date       Symbol YYYYMM MeanMonthSPYReturn
            2014-08-01 D      201408 0.001443
```

```
print(modified_clean_df[['Date','Symbol','YYYYMM','MeanMonthSPYReturn']].tail(1))
```

```
            Date       Symbol YYYYMM MeanMonthSPYReturn
            2019-07-26 DUK    201907 0.000167
```

Answer. One possible solution is given below:

[29]:
```
# One possible solution
# Add YYYYMM to market_df and clean_df
market_df['YYYYMM'] = market_df.apply(lambda row: row['Date'][:4] +␣
 ↪row['Date'][5:7], axis=1)
clean_df['YYYYMM'] = clean_df.apply(lambda row: row['Date'][:4] + row['Date'][5:
 ↪7], axis=1)

# Calculate modified_market_df DataFrame for SPY
modified_market_df = market_df.groupby(['Symbol','YYYYMM']).mean().reset_index()
modified_market_df = modified_market_df.rename(columns={'Return':
 ↪'MeanMonthSPYReturn'}) # rename column

# Inner merge modified_market_df DataFrame and clean_df by YYYYMM
modified_clean_df = pd.merge(clean_df,␣
 ↪modified_market_df[['YYYYMM','MeanMonthSPYReturn']], how='inner',␣
 ↪on='YYYYMM')

# Print results
print(modified_clean_df[['Date','Symbol','YYYYMM','MeanMonthSPYReturn']].
 ↪head(1))
print(modified_clean_df[['Date','Symbol','YYYYMM','MeanMonthSPYReturn']].
 ↪tail(1))
```

```
         Date Symbol  YYYYMM  MeanMonthSPYReturn
0  2014-08-01      D  201408            0.001443
            Date Symbol  YYYYMM  MeanMonthSPYReturn
5939  2019-07-26    DUK  201907            0.000167
```

1.7 Breaking down returns based on the broader market return

Now we can begin our granular analysis of how broader market returns are related to single stock returns.

Let's begin by breaking down broader market returns into quantiles. This quantile analysis approach we will take is commonly employed in data analysis to determine how the magnitude of one variable is related to another variable of interest.

We can explore this idea with the `pd.qcut()` method. Namely, `pd.qcut()` will allow us to cut the market returns by quantiles (we will eventually group by quantiles), and therefore will allow us to calculate summary statistics (such as average return) for each quantile.

Let's first extract the returns using the convenience of the `pivot()` method on a DataFrame. Pivoting a DataFrame may be accomplished by specifying:

1. An index to pivot on. In this case we choose `Date`.
2. Columns that we'd like to have after the pivot. In this case we'd like columns that are the 'Symbol.
3. The values that each (row,column) pair will show. In this case we'd like to have the `Return`.

We will pivot `merged_df` using these parameter inputs to output a DataFrame where the rows are the Date, each column is the Symbol, and the values are the open-to-close daily return for the given date and symbol.

[30]:
```
# Extract returns from merged_df, where we use pivot to simplify the task
return_df = merged_df.pivot(index='Date', columns='Symbol', values='Return')
return_df.head()
```

[30]:
```
Symbol              D       DUK       EXC       NEE        SO
Date
2014-08-18 -0.004205 -0.002905 -0.003445 -0.004659 -0.003446
2014-08-19  0.013250  0.009838  0.001567  0.008393  0.005741
2014-08-20  0.001438  0.002480  0.000313  0.005349  0.005508
2014-08-21  0.000718  0.003159  0.010022  0.002147  0.000000
2014-08-22 -0.004297 -0.008302  0.006213 -0.003470 -0.003195
```

Let's merge on the broader market returns from the SPY `market_df` loaded earlier:

[31]:
```
# Let's merge the SPY (broader market) returns by Date onto the return_df␣
 ↪DataFrame
full_df = pd.merge(return_df, market_df[['Date','Return']].set_index('Date'),␣
 ↪left_index=True, right_index=True)
full_df = full_df.rename(columns={'Return':'MarketReturn'}).reset_index()
full_df.head()
```

[31]:
```
         Date         D       DUK       EXC       NEE        SO  MarketReturn
0  2014-08-18 -0.004205 -0.002905 -0.003445 -0.004659 -0.003446      0.002846
1  2014-08-19  0.013250  0.009838  0.001567  0.008393  0.005741      0.002780
2  2014-08-20  0.001438  0.002480  0.000313  0.005349  0.005508      0.004038
```

```
3  2014-08-21  0.000718  0.003159  0.010022  0.002147  0.000000      0.002059
4  2014-08-22 -0.004297 -0.008302  0.006213 -0.003470 -0.003195     -0.000752
```

Through a few simple lines we've created a DataFrame `full_df` where each value is an open-to-close daily return, whether it be for one of the five symbols under study, or for the broader market.

We proceed by utilizing `pd.qcut()` with 10 quantiles. In general, the number of quantiles should be chosen based on how granular of a view you require.

[32]:
```
# Create 10 quantile categories by the market return
num_quantiles = 10
full_df['market_quantile'] = pd.
 ↪qcut(full_df['MarketReturn'],num_quantiles,labels=False)
full_df.head()
```

[32]:
```
         Date         D       DUK       EXC       NEE        SO  MarketReturn  \
0  2014-08-18 -0.004205 -0.002905 -0.003445 -0.004659 -0.003446      0.002846
1  2014-08-19  0.013250  0.009838  0.001567  0.008393  0.005741      0.002780
2  2014-08-20  0.001438  0.002480  0.000313  0.005349  0.005508      0.004038
3  2014-08-21  0.000718  0.003159  0.010022  0.002147  0.000000      0.002059
4  2014-08-22 -0.004297 -0.008302  0.006213 -0.003470 -0.003195     -0.000752

   market_quantile
0                7
1                7
2                7
3                6
4                3
```

Now let's group by `market_quantile` and calculate the mean return for all the symbols:

[33]:
```
# Group by market quantile and calculate the mean return
full_df.groupby('market_quantile').mean()
```

[33]:
```
                         D       DUK       EXC       NEE        SO  \
market_quantile
0                -0.004553 -0.003282 -0.006711 -0.004723 -0.002890
1                -0.002423 -0.001444 -0.002686 -0.001001 -0.000830
2                -0.001612 -0.000617 -0.001087 -0.000693 -0.000061
3                 0.001010  0.000420 -0.001277  0.000728  0.000938
4                 0.001189  0.000615  0.000377  0.000903  0.000444
5                -0.001516 -0.001656 -0.000699 -0.000985 -0.001240
6                 0.001950  0.001233  0.002246  0.002210  0.001040
7                 0.001804  0.000505  0.001006  0.000831  0.000963
8                 0.002148  0.002366  0.004323  0.003042  0.002563
9                 0.005446  0.004366  0.007506  0.006591  0.004721

                 MarketReturn
```

```
market_quantile
0                -0.013363
1                -0.005045
2                -0.002577
3                -0.001154
4                -0.000113
5                 0.000887
6                 0.002055
7                 0.003568
8                 0.005787
9                 0.012087
```

Each value in the above DataFrame output is a mean of daily returns for a given symbol, where the mean is taken across all dates that correspond to the `market_quantile` listed as the index of the output DataFrame. Note that higher quantile numbers indicate higher market returns, while lower quantile numbers indicate lower market returns (in this case, negative market returns).

We see here that the energy stock returns do indeed follow a pattern when the SPY returns are large (higher quantile number) or small (lower quantile number). Namely, stock returns of the individual stocks follow the same pattern as those of the broader market. Hence, the broader market has an effect on the single-stock returns; that is, the magnitude of the broader stock market return is correlated with the magnitude of the return of the individual stocks.

1.7.1 Exercise 8:

We created quantiles for market returns and subsequently calculated the mean return for each of these quantiles. Perform a similar analysis as above, but instead group by quantiles for the market volatility rather than market returns, and calculate the standard deviation of returns for each market volatility quantile category rather than the mean return.

Answer. One possible solution is given below:

[34]:
```
# One possible solution
# Merge market volstat onto returns
new_df = pd.merge(return_df, market_df[['Date','VolStat']].set_index('Date'),␣
 ↪left_index=True, right_index=True)
new_df = new_df.rename(columns={'VolStat':'MarketVolStat'}).reset_index()

# Add on quantiles
num_quantiles = 10
new_df['market_volstat_quantile'] = pd.
 ↪qcut(new_df['MarketVolStat'],num_quantiles,labels=False)

# Calcualte mean by quantile
new_df.groupby('market_volstat_quantile').std()
```

[34]:

market_volstat_quantile	D	DUK	EXC	NEE	SO
0	0.007394	0.007464	0.009186	0.007310	0.008521
1	0.007177	0.007135	0.009354	0.007658	0.006974
2	0.008480	0.008084	0.009745	0.007748	0.007407
3	0.008298	0.008194	0.010211	0.008106	0.007824
4	0.009563	0.008093	0.010409	0.007003	0.008374
5	0.009868	0.008642	0.010350	0.008032	0.008836
6	0.008900	0.009298	0.013011	0.008751	0.009049
7	0.009557	0.009409	0.010587	0.010960	0.008547
8	0.010907	0.011419	0.013512	0.010837	0.010354
9	0.014097	0.013590	0.016814	0.014441	0.012464

market_volstat_quantile	MarketVolStat
0	0.000508
1	0.000262
2	0.000242
3	0.000245
4	0.000301
5	0.000370
6	0.000509
7	0.000674
8	0.001292
9	0.009016

Similar to how stock returns' directions follow those of the broader market, the volatility of stock returns also follow the volatility of returns of the broader market. This warrants further analysis of the root cause of this effect as a future project.

1.8 Conclusions

We've explored stock returns for the five energy sector stocks in terms of their own volatility regimes, and their returns and volatility relative to the broader market. We found that for the stocks under study, there is no strong link between volatility level and the direction of the daily stock return. Moreover, we found that when comparing stocks to the broader market, their returns and volatility levels are amplified when the market returns and volatility levels are high. These findings indicate there is an intrinsic link between returns and volatility, both in the single stock case and the broader market case. This lends a variety of avenues of exploration for follow-up projects.

1.9 Takeaways

In this case, you learned multiple data manipulation tools in `pandas`, including anonymous functions, grouping, merging, quantile cuttting, and pivoting, while making use of data transformation and aggegation analysis techniques that we've previously learned.

`pandas` is an increbily versatile package and can significantly increase productivity and deliver exceptional business insights. These techniques should serve as a strong basis for any future data analyses you may conduct.

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]: