Disclaimer: Notebooks translated to Spanish are distributed as an optional aid to assist in your learning and comprehension. We make no guarantees that the translations are completely accurate nor that the translated code blocks will run properly.

¿Cómo se relacionan la cantidad de transacciones bursátiles y la volatilidad de las acciones de las compañías de energía?

Objetivos (2 min)

Al final de este caso, habremos introducido la librería `pandas` en Python. También habrá ganado experiencia con la librería `numpy`, sabrá cómo leer archivos de datos y hallar estadísticas descriptivas.

También debería empezar a desarrollar una mentalidad adecuada para investigar la librería por su propia cuenta, a través de la documentación u otros recursos como StackOverflow. La auto-investigación de la documentación existente es una parte crucial del desarrollo profesional en el campo de los datos.

Introducción (5 min)

Contexto empresarial. Usted es un analista de un gran banco enfocado en inversiones en acciones de recursos naturales. Los recursos naturales son vitales para una variedad de industrias en nuestra economía. Recientemente, su división se ha interesado en las siguientes acciones:

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

Todas estas acciones forman parte del sector de la energía, un sector importante pero volátil del mercado de valores. Si bien la alta volatilidad aumenta la posibilidad de grandes ganancias, también hace más probable que se produzcan grandes pérdidas, por lo que el riesgo debe manejarse cuidadosamente con las acciones de alta volatilidad.

Dado que su empresa es bastante grande, debe haber un volumen de transacciones suficiente (cantidad media de acciones negociadas por día) para que la compañía pueda operar fácilmente en estas acciones. De lo contrario, este efecto, sumado a la alta volatilidad natural de las acciones, podría hacer que éstas sean demasiado arriesgadas para que el banco invierta en ellas.

Problema empresarial. Dado que tanto el bajo volumen de operaciones como la alta volatilidad presentan riesgos para sus inversiones, el jefe de su equipo le pide que investigue lo siguiente: **"¿Cómo se relaciona la volatilidad de las acciones de energía con su volumen de transacciones diario promedio?"**

Contexto analítico. Los datos que se le han dado están en el formato Valores Separados por Comas (Comma Separated Value, CSV), y comprenden los datos de precios y volumen de comercio de las acciones mencionadas. Este caso comienza con un breve resumen de estos datos, después de lo cual: (1) aprenderá a utilizar la librería pandas de Python para cargar los datos; (2) utilizará `pandas` para llevar estos datos a una forma que facilite su análisis; y, finalmente, (3) utilizará `pandas` para analizar la pregunta expresada más arriba y llegar a una conclusión. Como habrá adivinado, `pandas` es una librería enormemente útil para el análisis y la manipulación de datos.

Importando paquetes para ayudar en el análisis de datos

Las librerías externas (también conocidas como paquetes) son repositorios de código que contienen una variedad de funciones y herramientas preescritas. Esto permite realizar una variedad de tareas complejas en

Python sin tener que "reinventar la rueda", sin tener que construir todo desde cero. Usaremos dos paquetes básicos: `pandas` y `numpy`.

`pandas` es una librería externa que provee funcionalidad para el análisis de datos. Pandas ofrece específicamente una variedad de estructuras de datos y métodos de manipulación de datos que permiten realizar tareas complejas con comandos simples de una línea.

`numpy` es un paquete que usaremos más adelante en un caso que requerirá numerosas operaciones matemáticas. Juntos, pandas y numpy permiten crear un flujo de trabajo de ciencia de datos dentro de Python. `numpy` es en muchos sentidos fundamental para `pandas`, proporcionando operaciones vectorizadas, mientras que `pandas` proporciona abstracciones de nivel superior construidas sobre `numpy`.

Vamos a importar ambos paquetes usando la palabra clave `import`. Cambiaremos el nombre de `pandas` a `pd` y `numpy` a `np` usando la palabra clave `as`. Esto nos permite usar la abreviatura del nombre corto cuando queramos referirnos a cualquier función que esté dentro de cualquiera de los paquetes. Las abreviaturas que elegimos son estándar en toda la industria de la ciencia de los datos y deben seguirse a menos que haya una muy buena razón para no hacerlo.

[0]:
```
# Importar el paquete Pandas
import pandas as pd

# Importar el paquete NumPy
import numpy as np
```

Ahora que estos paquetes están cargados en Python, podemos usar su contenido. Primero echemos un vistazo a `pandas` ya que tiene una variedad de características que usaremos para cargar y analizar nuestros datos de acciones en la bolsa.

Fundamentos de `pandas` (8 min)

`pandas` es una librería de Python que facilita una amplia gama de análisis y manipulación de datos. Antes, vimos estructuras básicas de datos en Python como listas y diccionarios. Aunque podemos construir una tabla de datos básicos (similar a una hoja de cálculo de Excel) usando listas anidadas en Python, se hace bastante difícil trabajar con ellas. Por el contrario, en `pandas` la estructura de datos tabular conocida como `DataFrame`, es una excelente opción, pues permite manipular fácilmente los datos pensando en ellos en términos de filas y columnas.

Si alguna vez ha usado u oído hablar de R o SQL antes, `pandas` trae algunas funcionalidades de cada uno de estos a Python, permitiéndole estructurar y filtrar datos más eficientemente que con Python puro. Esta eficiencia se ve reflejada de dos maneras distintas:

- Los scripts escritos usando `pandas` a menudo corren más rápido que los scripts escritos en Python puro.
- Los scripts escritos usando `pandas` a menudo contendrán muchas menos líneas de código que su equivalente escrito en Python puro.

En el núcleo de la librería `pandas` hay dos estructuras de datos y objetos fundamentales: 1. Series 2. DataFrame

Un objeto `Series` almacena datos de una sola columna junto con un **índice**. Un índice es sólo una forma de "numerar" el objeto `Series`. Por ejemplo, en este estudio de caso, los índices serán fechas, mientras que los datos de una sola columna pueden ser los precios de las acciones o la cantidad diaria de transacciones.

Un objeto `DataFrame` es una estructura de datos tabular bidimensional con ejes etiquetados. Es conceptualmente útil pensar en un objeto `DataFrame` como una colección de objetos `Series`. Es decir, piense en cada

columna de un DataFrame como un único objeto `Series`, donde cada uno de estos objetos `Series` comparte un índice común (el índice del objeto `DataFrame`).

A continuación se muestra la sintaxis para crear un objeto `Series`, seguida de la sintaxis para crear un objeto `DataFrame`. Ten en cuenta que los objetos DataFrame también pueden tener una sola columna - piensa en esto como un DataFrame que consiste en un solo objeto Series:

[0]:
```
# Crear un objeto Series simple
simple_series = pd.Series(
    index=[0, 1, 2, 3], name="Volumen", data=[1000, 2600, 1524, 98000]
)
simple_series
```

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

Cambiando `pd.Series` a `pd.DataFrame`, y añadiendo una lista de columnas, se puede crear un objeto `DataFrame`:

[0]:
```
# Crear un objeto DataFrame simple
simple_df = pd.DataFrame(
    index=[0, 1, 2, 3], columns=["Volumen"], data=[1000, 2600, 1524, 98000]
)
simple_df
```

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

Los objetos del DataFrame son más generales que los objetos de la Serie, y un DataFrame puede contener muchos objetos de la Serie, cada uno como una columna diferente. Vamos a crear un objeto DataFrame de dos columnas:

[0]:
```
# Crear otro objeto DataFrame
otro_df = pd.DataFrame(
    index=[0, 1, 2, 3],
    columns=["Fecha", "Volumen"],
    data=[[20190101, 1000], [20190102, 2600], [20190103, 1524], [20190104, 98000]],
)
otro_df
```

[0]:
```
      Fecha  Volumen
0  20190101     1000
1  20190102     2600
2  20190103     1524
3  20190104    98000
```

Fíjese en cómo una lista de listas fue usada para especificar los datos en el `otro_df` DataFrame. Cada elemento de la lista corresponde a una fila en el DataFrame, por lo que la lista tiene 4 elementos porque hay

4 índices. Cada elemento de la lista de listas tiene 2 elementos porque el DataFrame tiene dos columnas.

Usando Pandas para analizar los datos de las acciones (10 min)

Recordemos que tenemos archivos CSV que incluyen datos de cada una de las siguientes acciones:

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

Los datos disponibles para cada acción incluyen:

1. **Date:** El día del año
2. **Open:** El precio de apertura de las acciones en ese día
3. **High:** El precio más alto observado de las acciones en ese día
4. **Low:** El precio más bajo observado de las acciones en ese día
5. **Close:** El precio de cierre de las acciones en ese día
6. **Adj Close:** El precio de cierre de las acciones ajustado en ese día (ajustado por desdoblamientos y dividendos)
7. **Volume:** La cantidad de acciones negociadas durante el día

Para tener una mejor idea de los datos disponibles, veamos primero sólo los datos de Dominion Energy, que cotiza en la Bolsa de Nueva York con el símbolo D. Se le da un archivo CSV que contiene los datos de las acciones de la compañía, `D.`. `pandas` permite una fácil carga de los archivos CSV mediante el uso del método pd.read_csv():

[0]:
```
# Cargar un archivo como un DataFrame y asignarlo a df
df = pd.read_csv("data/D.csv")
```

El contenido del archivo `D.csv` está ahora almacenado en el objeto DataFrame `df`.

Hay varios métodos y atributos comunes disponibles para revisar los datos y obtener una idea general acerca de ellos:

1. `DataFrame.head()` -> devuelve los nombres de las columnas y las primeras 5 filas por defecto
2. `DataFrame.tail()` -> devuelve los nombres de las columnas y las últimas 5 filas por defecto
3. `DataFrame.shape` -> devuelve el número de filas y el número de columnas
4. `DataFrame.columns` -> devuelve el índice de las columnas
5. `DataFrame.index` -> devuelve el índice de las filas

En su tiempo libre, por favor revise la documentación de Pandas y explore los parámetros de estos métodos así como otros métodos. La familiaridad con esta librería mejorará drásticamente su productividad como científico de datos.

Usando `df.head()` y `df.tail()` podemos dar un vistazo al contenido de los datos. A menos que se especifique lo contrario, los objetos Series y DataFrame tienen índices enteros que comienzan en 0 y aumentan monotónicamente.

[0]:
```
# Recupere la cabeza del DataFrame (es decir, las filas superiores del DataFrame)
df.head()
```

[0]:

	Date	Open	High	Low	Close	Adj Close	Volume
0	2014-07-28	69.750000	71.059998	69.750000	70.879997	57.963978	1806400
1	2014-07-29	70.669998	70.980003	69.930000	69.930000	57.187099	2231100
2	2014-07-30	70.000000	70.660004	68.400002	68.970001	56.402020	2588900

```
3  2014-07-31  68.629997  68.849998  67.580002  67.639999  55.314388  3266900
4  2014-08-01  67.330002  68.410004  67.220001  67.589996  55.273487  2601800
```

[0]:
```
# Recupere la cola del DataFrame (es decir, las filas inferiores del DataFrame)
df.tail()
```

[0]:
```
            Date       Open       High        Low      Close  Adj Close  \
1254  2019-07-22  76.879997  76.930000  75.779999  76.260002  76.260002
1255  2019-07-23  76.099998  76.199997  75.269997  75.430000  75.430000
1256  2019-07-24  75.660004  75.720001  74.889999  75.180000  75.180000
1257  2019-07-25  75.150002  75.430000  74.610001  74.860001  74.860001
1258  2019-07-26  74.730003  75.349998  74.610001  75.150002  75.150002

       Volume
1254  2956500
1255  3175600
1256  3101900
1257  3417200
1258  3076500
```

Así, vemos que hay 1259 entradas de datos (cada una con 7 puntos de datos) para Dominion Energy. Se accede a la "forma" de un DataFrame usando el atributo `shape`:

[0]:
```
# Determinar la "forma" de la estructura bidimensional, es decir (número de filas,␣
↪número de columnas)
df.shape
```

[0]:
```
(1259, 7)
```

Es importante notar que `DataFrame.columns` y `DataFrame.index` devuelven un objeto índice en lugar de una lista. Para convertir un índice en una lista para manipularla, usamos el método `list()`:

[0]:
```
# Lista de los nombres de las columnas del DataFrame
list(df.columns)
```

[0]:
```
['Date', 'Open', 'High', 'Low', 'Close', 'Adj Close', 'Volume']
```

[0]:
```
# Lista de los nombres de las columnas del DataFrame
list(df.index)[0:20] # sólo muestra los primeros 20 valores de índice para no ocupar␣
↪demasiado espacio en la pantalla
```

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

Creando variables adicionales relevantes para entender la volatilidad de las acciones (7 min)

A menudo, los datos que se le proporcionen no serán suficientes para lograr su objetivo. Es posible que tenga que añadir variables o características de datos adicionales para ayudarse. Recuerde que nuestra pregunta original se refería a la relación entre la cantidad de acciones negociadas y la volatilidad. Por lo tanto, nuestro DataFrame debe tener características relacionadas con ambas cantidades.

Puede ser útil pensar en agregar columnas a los DataFrames como si se estuvieran añadiendo columnas adyacentes una por una en Excel. Aquí hay un ejemplo de cómo hacerlo:

[0]:
```
# Agregar una nueva columna llamada "Symbol"
df["Symbol"] = "D"
df.head()
```

[0]:

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

[0]:
```
# Podemos acceder a una columna usando corchetes [] y el nombre de la columna
df['Volume'].head() # añadimos .head() para no mostrar demasiadas filas
```

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

[0]:
```
# Agregar una nueva columna llamada "Volume_Millions" (la cantidad de transacciones␣
↪contadas en millones), que se calcula a partir de la columna Volume actualmente en df
# Dividir cada fila en df['Volume'] por 1 millón, almacenar en una nueva columna
df["Volume_Millions"] = df["Volume"] / 1000000.0
df.head()
```

[0]:

	Date	Open	High	Low	Close	Adj Close	Volume	Symbol	Volume_Millions
0	2014-07-28	69.750000	71.059998	69.750000	70.879997	57.963978	1806400	D	1.8064
1	2014-07-29	70.669998	70.980003	69.930000	69.930000	57.187099	2231100	D	2.2311
2	2014-07-30	70.000000	70.660004	68.400002	68.970001	56.402020	2588900	D	2.5889
3	2014-07-31	68.629997	68.849998	67.580002	67.639999	55.314388	3266900	D	3.2669
4	2014-08-01	67.330002	68.410004	67.220001	67.589996	55.273487	2601800	D	2.6018

[0]:
```
# Revise a la forma actualizada del DataFrame. Se han añadido dos nuevas columnas.
df.shape
```

[0]: (1259, 9)

Como se ha dicho, necesitamos tener una característica en nuestro DataFrame que esté relacionada con la volatilidad. Debido a que no existe actualmente, debemos crearla a partir de las características ya disponibles. Recordemos que la volatilidad es la desviación estándar de los rendimientos diarios durante un período de tiempo, así que vamos a crear una característica para los rendimientos diarios:

[0]:
```
df["VolStat"] = (df["High"] - df["Low"]) / df["Open"]
df["Return"] = (df["Close"] / df["Open"]) - 1.0
```

Aquí vemos el poder de **pandas**. Podemos simplemente realizar operaciones matemáticas en las columnas de los DataFrames como si los DataFrames fueran variables individuales.

[0]:
```
df.head()
```

[0]:

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

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

Ahora tenemos características relevantes a la pregunta original, y podemos proceder al análisis. Un primer paso comúnmente usado en el análisis de datos es revisar la la distribución de los datos disponibles. Haremos esto a continuación.

Entendiendo la distribución de los datos a través de estadísticas de resumen (12 min)

Agrupemos las estadísticas resumidas de las cinco empresas del sector de la energía que se están estudiando. Afortunadamente, los objetos DataFrame y Series ofrecen un sinfín de métodos estadísticos de resumen de datos:

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

A continuación, usamos cada método en la columna `Volume_Millions`. Fíjese en lo simples que son las funciones para aplicar al DataFrame. Simplemente escriba el nombre del DataFrame, seguido de un `.` y luego el nombre del método que quiera calcular. Hemos elegido seleccionar una sola columna `Volume_Millions` del DataFrame `df`, pero podríamos haber llamado fácilmente estos métodos en el DataFrame completo en lugar de una sola columna:

[0]:
```
# Calcular el mínimo de la columna Volume_Millions
df["Volume_Millions"].min()
```

[0]: 0.73839999999999995

```
[0]: # Calcular la mediana de la columna Volume_Millions
     df["Volume_Millions"].median()
```

[0]: 2.6957

```
[0]: # Calcula el promedio de la columna Volume_Millions
     df["Volume_Millions"].mean()
```

[0]: 3.0881293089753776

```
[0]: # Calcular el máximo de la columna Volume_Millions
     df["Volume_Millions"].max()
```

[0]: 14.587400000000001

También nos gustaría explorar la distribución de los datos a un nivel más granular para ver cómo la distribución se ve más allá de las simples estadísticas resumidas presentadas anteriormente. Para ello, podemos utilizar el método `quantile()`. El método `quantile()` devolverá el valor que representa el percentil dado de todos los datos en estudio (en este caso, de los datos en `Volume_Millions`):

```
[0]: # Calcular el percentil 25
     df['Volume_Millions'].quantile(0.25)
```

```
[0]: # Calcular el percentil 75
     df['Volume_Millions'].quantile(0.75)
```

[0]: 3.61285

¿Existe un método más eficiente para calcular rápidamente todas estas estadísticas resumidas? Sí. Un método increíblemente útil que combina estas estadísticas de resumen y también añade un par más es el método `describe()`:

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

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

A partir de este análisis de la distribución del volumen de negociaciones diarias podemos ver que más de 14 millones de acciones sería un día de comercio muy grande, mientras que por debajo de 2 millones de acciones sería un día de comercio relativamente pequeño.

Además de `describe()`, existe un método value_counts() para hallar la frecuencia de los elementos en datos categóricos. Tenga en cuenta que `value_counts()` es un método de la clase Series y NO de la clase

DataFrame. Esto significa que tiene que aislar una columna específica de un DataFrame antes de llamar a `value_counts()`:

[0]:
```
dicc_datos = {
    "números": [1, 2, 3, 4, 5, 6, 7, 8],
    "colores": ["rojo", "rojo", "rojo", "azul", "azul", "verde", "azul", "verde"],
}
categorico_df = pd.DataFrame(data=dicc_datos)

categorico_df
```

[0]:

	números	colores
0	1	rojo
1	2	rojo
2	3	rojo
3	4	azul
4	5	azul
5	6	verde
6	7	azul
7	8	verde

[0]:
```
#¿Por qué no funciona esto? (descomente la expresión que sigue)
#categorico_df.value_counts()
```

[0]:
```
# Sólo los objetos Series pueden llamar a este método (descomente la siguiente␣
↪expresión)
#categorico_df['colores'].value_counts()
```

[0]:
```
azul     3
rojo     3
verde    2
Name: colores, dtype: int64
```

Ejercicio 1 (5 min):

Determinar los percentiles 25, 50 y 75 para las columnas `Open`, `High`, `Low`, y `Close` de `df`.

Respuesta.

Agregando datos de múltiples compañías (18 min)

Hasta ahora, sólo hemos estado mirando los datos de una de nuestras cinco compañías. Vamos a combinar los cinco archivos CSV para analizar las cinco compañías juntas. Esto también reducirá la cantidad de trabajo de programación requerido ya que el código será compartido entre las cinco compañías.

Una forma de lograr esta tarea de agregación es usar el método pd.concat() de `pandas`. Una entrada en este método puede ser una lista de DataFrames que quiera concatenar. Usaremos un bucle `for` sobre los símbolos de las acciones para cargar el archivo CSV correspondiente y luego pegar el resultado a una lista que luego se agregará usando `pd.concat()`. Miremos a cómo se hace esto.

[0]:
```
# Cargar cinco archivos en un DataFrame
print("Definición de los símbolos de las acciones")
```

```
simbolos_a_cargar = ["D", "EXC", "NEE", "SO", "DUK"]
lista_de_df = []

# Bucle sobre los símbolos
print(" --- Inicie el bucle sobre los símbolos --- ")
for simbolo in simbolos_a_cargar:
    print("Procesando el símbolo: " + simbolo)
    temp_df = pd.read_csv("data/" + simbolo + ".csv")
    temp_df["Volume_Millions"] = temp_df["Volume"] / 1000000.0

    # Agregar una nueva columna con el nombre del símbolo para distinguirlo en el␣
 ↪DataFrame final
    temp_df["Symbol"] = simbolo
    lista_de_df.append(temp_df)

# Usando un salto de línea al final de esta cadena de caracteres por estética
print(" --- Bucle completo sobre los símbolos --- \n")

# Combinando en un solo DataFrame usando el concat
print("Agregando los datos")
agr_df = pd.concat(lista_de_df, axis=0)

# Agregando estadísticas relevantes para este análisis de retorno y volatilidad
print("Calculando las características importantes")
agr_df["VolStat"] = (agr_df["High"] - agr_df["Low"]) / agr_df["Open"]
agr_df["Return"] = (agr_df["Close"] / agr_df["Open"]) - 1.0

print("Forma del DataFrame agr_df (filas, columnas): ")
print(agr_df.shape)

print("Cabeza del DataFrame agr_df: ")
agr_df.head()
```

```
Definición de los símbolos de las acciones
 --- Inicie el bucle sobre los símbolos ---
Procesando el símbolo: D
Procesando el símbolo: EXC
Procesando el símbolo: NEE
Procesando el símbolo: SO
Procesando el símbolo: DUK
 --- Bucle completo sobre los símbolos ---

Agregando los datos
Calculando las características importantes
Forma del DataFrame agr_df (filas, columnas):
(6295, 11)
Cabeza del DataFrame agr_df:
```

[0]:

	Date	Open	High	Low	Close	Adj Close	Volume	\
0	2014-07-28	69.750000	71.059998	69.750000	70.879997	57.963978	1806400	
1	2014-07-29	70.669998	70.980003	69.930000	69.930000	57.187099	2231100	

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

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

Después del bucle `for`, hemos agregado y añadido las características relevantes que identificamos en la sección anterior. Luego imprimimos la cabeza del DataFrame agregado para tener una idea del formato de los datos, y también hemos impreso la forma del DataFrame. Esto es para comprobar que nuestro DataFrame final es más o menos lo que esperamos. Noten que el DataFrame agregado tiene el mismo número de columnas que los datos originales de la tabla de la acción (D), sin embargo el número de filas se ha quintuplicado. Esto tiene sentido, porque cada símbolo adicional contiene 1259 entradas de datos, por lo que cinco símbolos conducen a un total de `1259*5 = 6295` filas. Por lo tanto, este resultado pasa nuestro control de calidad.

Ahora, si queremos revertir este proceso y extraer los datos relevantes para un único símbolo de acciones del DataFrame agregado `ag_df`, podemos hacerlo usando el operador `==`, que devuelve `True` cuando dos objetos contienen el mismo valor, y `False` en caso contrario:

[0]:
```
simbolo_DUK_df = agr_df[agr_df["Symbol"] == "DUK"]
simbolo_DUK_df.head()
```

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

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

Mirando el bloque de código de arriba, hemos filtrado las filas que corresponden a cada símbolo. A saber,

[0]:
```
agr_df['Symbol'] == 'DUK'
```

[0]:
```
0       False
1       False
2       False
3       False
4       False
5       False
6       False
7       False
```

```
8       False
9       False
10      False
11      False
12      False
13      False
14      False
15      False
16      False
17      False
18      False
19      False
20      False
21      False
22      False
23      False
24      False
25      False
26      False
27      False
28      False
29      False
        …
1229     True
1230     True
1231     True
1232     True
1233     True
1234     True
1235     True
1236     True
1237     True
1238     True
1239     True
1240     True
1241     True
1242     True
1243     True
1244     True
1245     True
1246     True
1247     True
1248     True
1249     True
1250     True
1251     True
1252     True
1253     True
1254     True
1255     True
1256     True
```

```
1257     True
1258     True
Name: Symbol, Length: 6295, dtype: bool
```

devuelve una serie booleana con el mismo número de filas de `agr_df`, donde cada valor es verdadero o falso dependiendo de si el valor del `Symbol` de una fila específica es igual a `DUK`.

Esta técnica de extracción de filas nos será útil más adelante en este caso cuando realicemos los análisis usando el símbolo de cada acción.

Ejercicio 2 (4 min):

Si sumamos el número de filas de los cinco DataFrames, `D_df`,`NEE_df`,`EXC_df`,`SO_df`, y `DUK_df`, llegaríamos al mismo número de filas que `agr_df`: 6295 filas. Si en su lugar usamos el operador `!=` en las cinco líneas donde filtramos cada símbolo, ¿cuántas filas tendríamos si sumamos todas las filas en los cinco nuevos DataFrames?

(a) 31475

(b) 12590

(c) 25180

(d) 6295

Respuesta.

Ejercicio 3 (5 min):

Escriba código para aplicar un bucle `for` sobre cada uno de los cinco símbolos, extraer sólo las filas correspondientes a cada símbolo, y calcular e imprimir el valor promedio `VolStat` para cada uno de los cinco símbolos.

Respuesta.

Analizando los niveles de volatilidad de cada acción (20 min)

Pandas ofrece la posibilidad de agrupar filas de DataFrames que estén relacionadas entre ellas según los valores de otras filas. Esta útil característica se logra usando el método groupby(). Echemos un vistazo y veamos cómo se puede usar para agrupar filas de manera que cada grupo corresponda a un solo símbolo de acción:

[0]:
```
# Usar el método groupby(), notar que un objeto DataFrameGroupBy es devuelto
agr_df.groupby('Symbol')
```

[0]:
```
<pandas.core.groupby.groupby.DataFrameGroupBy object at 0x7fd1ea24d438>
```

Aquí, el objeto `DataFrameGroupBy` se puede entender más fácilmente como algo que contiene un objeto DataFrame para cada grupo (en este caso, un objeto DataFrame para cada símbolo). Específicamente, cada elemento del objeto es una tupla que contiene el identificador de grupo (en este caso el símbolo), y las filas correspondientes del DataFrame que tienen ese símbolo).

Afortunadamente, `pandas` le permite iterar sobre el objeto `groupby()` para ver lo que hay dentro:

[0]:
```
grp_obj = agr_df.groupby("Symbol") # Datos del grupo en agr_df filtrados por el símbolo

# Haciendo un bucle a través de los grupos
for item in grp_obj:
    print(" ------ ¡Comencemos! ------ ")
    print(type(item)) # Mostrando el tipo de artículo en grp_obj
    print(item[0]) # Símbolo
    print(item[1].head())  # DataFrame con datos del Símbolo
    print(" ------ ¡Listo! ------ ")
```

```
 ------ ¡Comencemos! ------
<class 'tuple'>
D
         Date       Open       High        Low      Close  Adj Close   Volume  \
0  2014-07-28  69.750000  71.059998  69.750000  70.879997  57.963978  1806400
1  2014-07-29  70.669998  70.980003  69.930000  69.930000  57.187099  2231100
2  2014-07-30  70.000000  70.660004  68.400002  68.970001  56.402020  2588900
3  2014-07-31  68.629997  68.849998  67.580002  67.639999  55.314388  3266900
4  2014-08-01  67.330002  68.410004  67.220001  67.589996  55.273487  2601800

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

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

```
4  2014-08-01  31.139999  32.080002  31.100000  31.540001  26.103081  9734300

   Volume_Millions Symbol   VolStat    Return
0           5.6834    EXC  0.023878  0.017192
1           6.2928    EXC  0.019411 -0.014715
2           7.9766    EXC  0.025609 -0.019602
3           9.2361    EXC  0.022308  0.004850
4           9.7343    EXC  0.031471  0.012845
 ------ ¡Listo! ------
 ------ ¡Comencemos! ------
<class 'tuple'>
NEE
         Date       Open       High        Low      Close  Adj Close   Volume  \
0  2014-07-28  98.470001  99.760002  98.099998  99.580002  85.106087  1643000
1  2014-07-29  99.029999  99.389999  97.300003  98.400002  84.097595  1942500
2  2014-07-30  98.160004  98.500000  95.760002  96.339996  82.337006  2844100
3  2014-07-31  95.639999  95.980003  93.800003  93.889999  80.243126  2725200
4  2014-08-01  93.500000  94.919998  93.279999  93.820000  80.183289  2514400

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

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

Combinemos el método `pd.groupby()` con el método `describe()` y apliquémoslo a cada símbolo para analizar la distribución de las características relacionadas con la volatilidad para cada símbolo.

[0]:
```
grp_obj = agr_df.groupby("Symbol") # Datos del grupo en agr_df por el símbolo

# Bucle a través de los grupos
for item in grp_obj:
    print("------Símbolo: ", item[0])
```

```
    grp_df = item[1]
    df_relevante = grp_df[["VolStat"]]
    print(df_relevante.describe())
```

```
------Símbolo:  D
           VolStat
count  1259.000000
mean      0.014836
std       0.006548
min       0.003640
25%       0.010246
50%       0.013528
75%       0.017920
max       0.062350
------Símbolo:  DUK
           VolStat
count  1259.000000
mean      0.014534
std       0.007047
min       0.003548
25%       0.010075
50%       0.012922
75%       0.017653
max       0.117492
------Símbolo:  EXC
           VolStat
count  1259.000000
mean      0.017722
std       0.008129
min       0.005230
25%       0.011868
50%       0.015931
75%       0.021752
max       0.093156
------Símbolo:  NEE
           VolStat
count  1259.000000
mean      0.014881
std       0.006544
min       0.004454
25%       0.010309
50%       0.013439
75%       0.017700
max       0.048495
------Símbolo:  SO
           VolStat
count  1259.000000
mean      0.014065
std       0.006109
min       0.003960
25%       0.009786
50%       0.012858
```

```
75%       0.016865
max       0.051847
```

Una observación inmediata a destacar es que el nivel de volatilidad en un día determinado puede variar ampliamente. Esto es evidente a partir de la gran distancia entre el mínimo y el máximo nivel de `VolStat`, que podemos ver usando el método `describe()`. Por ejemplo, el símbolo bursátil D tiene un valor mínimo `VolStat` de 0,003640, mientras que su valor máximo `VolStat` es de 0,062350. Eso es un factor de más de 10 veces en el valor de `VolStat`.

Mientras que esto es genial de ver, hay una manera más poderosa de mostrar estos datos en Pandas. Podemos llamar al método `describe()` directamente en el objeto `DataFrameGroupBy`. Esta línea le permite evitar tener que escribir un bucle `for` cada vez que quiera resumir los datos:

[0]:
```
# VolStat
agr_df[["Symbol", "VolStat"]].groupby("Symbol").describe()
```

[0]:

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

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

Estos datos son idénticos a los datos previamente emitidos usando el `for`. La diferencia es que utilizando las características del objeto `DataFrameGroupBy` se nos permite una fácil codificación, resultados rápidos y una salida limpia. Esto ilustra el poder de usar el método `pd.groupby()`: la generación de estadísticas para grupos de interés en sus datos es directa y eficiente de codificar.

Notará mucho este patrón a medida que se familiarice con Python y el análisis de datos. Hay muchas maneras de resolver un problema, pero a menudo una de ellas es sustancialmente más eficiente, tanto en términos de tiempo de ejecución como en términos de líneas de código.

Ejercicio 4 (3 min):

¿Qué ideas puede sacar del resumen estadístico de `VolStat` en términos de niveles de volatilidad?

Respuesta.

Ejercicio 5 (6 min):

Usando `agr_df` y un bucle `for`, escriba un script para determinar el valor medio de `VolStat` para cada símbolo por año.

Respuesta.

Etiquetando los puntos de datos como de alta o baja volatilidad (7 min)

Ahora que hemos determinado que los niveles de volatilidad de cada acción pueden variar ampliamente, el siguiente paso lógico es agrupar los períodos de alta y baja volatilidad para que podamos entonces observar cómo difiere el volumen entre esos períodos de tiempo.

Sin embargo, actualmente no tenemos una columna que identifique cuándo la volatilidad es alta y cuándo es baja. Por lo tanto, debemos crear una nueva columna llamada `VolLevel` usando algún umbral de volatilidad. Por ejemplo, nos gustaría tener un nuevo valor de columna determinado por:

```
si VolStat > umbral:
    VolLevel = 'HIGH'
de lo contrario:
    VolLevel = 'LOW'
```

Aquí definiremos los niveles de baja volatilidad como cualquier `VolStat` por debajo del percentil 50 (es decir, por debajo del nivel mediano de volatilidad para ese símbolo). Cada valor de percentil debe ser calculado por cada símbolo para asegurar que cada símbolo sea analizado individualmente.

Echemos un vistazo a cómo podemos realizar esta tarea usando la funcionalidad `groupby()` y el método `quantile()`, que devuelve el percentil para una serie de datos dada:

[0]:
```
# Determinar umbrales inferiores de volatilidad para cada símbolo
volstat_umbrales = agr_df.groupby("Symbol")["VolStat"].quantile(0.5) # percentil 50␣
 ↪(mediana)
print(volstat_umbrales)
```

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

Como nos gustaría etiquetar los períodos de alta y baja volatilidad por cada símbolo, haremos uso del método np.where() en la biblioteca `numpy`. Este método toma una entrada y comprueba una condición lógica: si la condición es verdadera, devolverá su segundo argumento, mientras que si la condición es falsa, devolverá su tercer argumento. Esto es muy similar a cómo funciona el método `SI.ERROR()` de Microsoft Excel (puede ser útil pensar de esta manera para aquellos familiarizados con Excel). Hagamos un bucle con cada símbolo y etiquetemos cada día como de alta y baja volatilidad:

[0]:
```
# Loop a través de los símbolos
print("Definición de los símbolos de las acciones")
lista_de_simbolos = ["D", "EXC", "NEE", "SO", "DUK"]
lista_de_df = []

# Bucle sobre todos los símbolos
print(" --- Bucle sobre los símbolos --- ")
for i in simbolos_a_cargar:
    print("Etiquetando el régimen de volatilidad para el símbolo: " + i)
```

```
    temp_df = agr_df[agr_df["Symbol"] == i].copy() # hacer una copia del DataFrame␣
↪para asegurarnos de no modificar agr_df sin querer
    volstat_t = volstat_umbrales.loc[i]
    temp_df["VolLevel"] = np.where(temp_df["VolStat"] < volstat_t, "LOW", "HIGH") #␣
↪Etiqueta del régimen de volatilidad
    lista_de_df.append(temp_df)

print(" --- Bucle completo sobre los símbolos --- ")

print("Agregando los datos")
df_con_etiquetas = pd.concat(lista_de_df)
```

```
Definición de los símbolos de las acciones
 --- Bucle sobre los símbolos ---
Etiquetando el régimen de volatilidad para el símbolo: D
Etiquetando el régimen de volatilidad para el símbolo: EXC
Etiquetando el régimen de volatilidad para el símbolo: NEE
Etiquetando el régimen de volatilidad para el símbolo: SO
Etiquetando el régimen de volatilidad para el símbolo: DUK
 --- Bucle completo sobre los símbolos ---
Agregando los datos
```

[0]:
```
df_con_etiquetas.head()
```

[0]:

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

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

Ahora hemos añadido una columna `VolLevel` que identifica si cada símbolo está en un período de alta o baja volatilidad en un día determinado. Como sabemos que el banco requerirá un mayor volumen de operaciones para operar en períodos de alta volatilidad, veamos ahora el promedio de volumen diario negociado para los días de alta y baja volatilidad.

¿El volumen de operaciones diarias se ve afectado por el nivel de volatilidad? (15 min)

Para explorar la relación entre el nivel de volatilidad y el volumen de operaciones diarias, agrupemos por `VolLevel` y miremos `Volume` promedio para los grupos de alta y baja volatilidad:

[0]:
```
df_con_etiquetas.groupby(["Symbol", "VolLevel"])[["Volume_Millions"]].mean()
```

[0]:

		Volume_Millions
Symbol	VolLevel	

```
D       HIGH            3.538901
        LOW             2.636641
DUK     HIGH            3.760172
        LOW             2.825710
EXC     HIGH            7.090384
        LOW             5.031123
NEE     HIGH            2.361096
        LOW             1.707347
SO      HIGH            6.148537
        LOW             4.417179
```

Ejercicio 6 (4 min):

¿Qué tendencia puede observar inmediatamente en relación con los regímenes de volatilidad?

Respuesta.

Ejercicio 7 (10 min):

Escriba el código para agrupar los períodos de tiempo en regímenes de baja, media y alta volatilidad, donde:

```
si VolStat > (percentil 75 de VolStat para el símbolo dado):
    VolLevel = 'HIGH'
o si VolStat > (percentil 25de VolStat para el símbolo dado):
    VolLevel = 'MEDIUM'
de lo contrario:
    VolLevel = 'LOW'
```

Cree un DataFrame `final_df` agrupado por `Symbol`, mostrando el `Volumen` medio para cada categoría de `VolLevel`.

Respuesta.

Como puede ver arriba, usamos `loc` para indexar el objeto DataFrame. Esta es sólo una de las muchas maneras diferentes de rebanar el objeto DataFrame. Recomendamos mirar en loc vs iloc ya que ambas serán útiles para todos los científicos de datos.

Graficando la volatilidad a través del tiempo (45 min)

Ya hemos respondido satisfactoriamente a nuestra pregunta original. Sin embargo, no es necesario solamente analizar los datos en formato tabular. Python contiene una funcionalidad que le permite analizar sus datos visualmente también.

Usaremos la funcionalidad de `pandas` sobre la librería estándar de graficación de Python, matplotlib. Vamos a importar la librería e instruir a Jupyter que muestre los gráficos en línea (es decir, mostrar los gráficos en la pantalla del cuaderno para que podamos verlos mientras ejecutamos el código):

[0]:
```
# importar la librería esencial de graficación en Python
import matplotlib.pyplot as plt

# Instruir a Jupyter que grafique en el cuaderno
%matplotlib inline
```

Antes de graficar, necesitamos convertir la columna `Date` en `agr_df` en un objeto similar a los objetos de tipo `datetime`, la representación interna de Python para datos de fechas. `pandas` ofrece el método to_datetime() para convertir una cadena que representa un formato de fecha dado en un objeto parecido a un `datetime`. Instruimos a `pandas` que use `formato='%Y-%m-%d'`, ya que nuestras fechas están en este formato, donde %Y indica el año numérico, %m indica el mes numérico y %d indica el día numérico. Si nuestras fechas estuvieran en otro formato, modificaríamos este valor de entrada apropiadamente.

[0]:
```
# Para convertir una cadena en una fecha y hora
# La llamamos DateTime, pero puede ser cualquier otro nombre
agr_df["DateTime"] = pd.to_datetime(agr_df["Date"], format="%Y-%m-%d")

# Usar esta columna como índice para facilitar el graficado
agr_df = agr_df.set_index(["DateTime"])
agr_df.head()
```

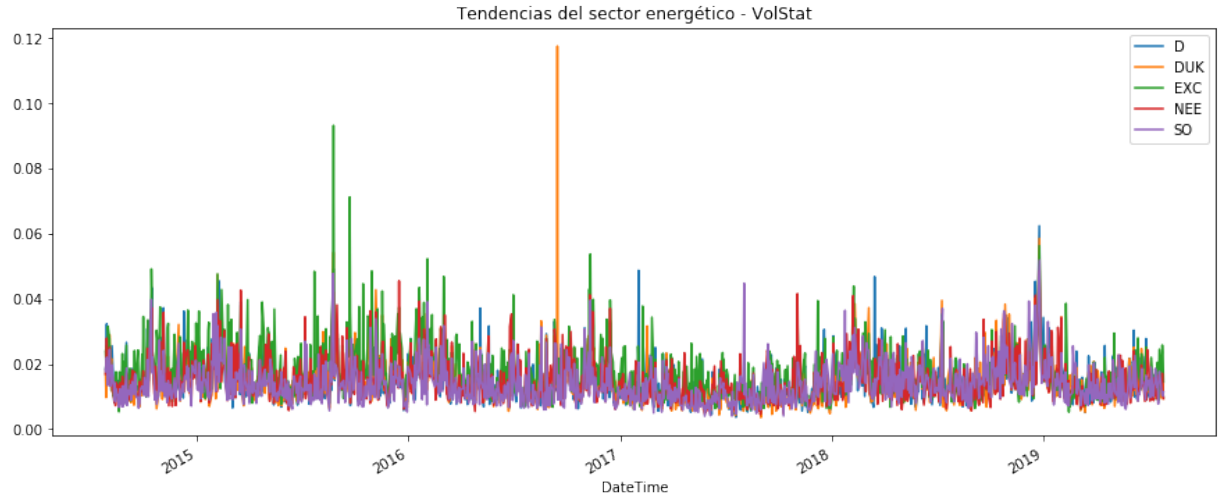
[0]:

DateTime	Date	Open	High	Low	Close	Adj Close	Volume	Volume_Millions	Symbol	VolStat	Return	YYYY
2014-07-28	2014-07-28	69.750000	71.059998	69.750000	70.879997	57.963978	1806400	1.8064	D	0.018781	0.016201	2014
2014-07-29	2014-07-29	70.669998	70.980003	69.930000	69.930000	57.187099	2231100	2.2311	D	0.014858	-0.010471	2014
2014-07-30	2014-07-30	70.000000	70.660004	68.400002	68.970001	56.402020	2588900	2.5889	D	0.032286	-0.014714	2014
2014-07-31	2014-07-31	68.629997	68.849998	67.580002	67.639999	55.314388	3266900	3.2669	D	0.018505	-0.014425	2014
2014-08-01	2014-08-01	67.330002	68.410004	67.220001	67.589996	55.273487	2601800	2.6018	D	0.017674	0.003861	2014

Ahora estamos listos para ver directamente la volatilidad a través del tiempo. Agrupemos por símbolos y tracemos el valor de `VolStat` a través del tiempo. La serie de tiempo de cada símbolo será etiquetada con un color diferente por defecto:

[0]:
```
# Mirando los regímenes de volatilidad
fig, ax = plt.subplots(figsize=(15, 6))
agr_df.groupby("Symbol")["VolStat"].plot(
    ax=ax, legend=True, title="Tendencias del sector energético - VolStat"
)
```

[0]:
```
Symbol
D      AxesSubplot(0.125,0.2;0.775x0.68)
DUK    AxesSubplot(0.125,0.2;0.775x0.68)
EXC    AxesSubplot(0.125,0.2;0.775x0.68)
NEE    AxesSubplot(0.125,0.2;0.775x0.68)
SO     AxesSubplot(0.125,0.2;0.775x0.68)
Name: VolStat, dtype: object
```



Observamos que los períodos de gran volatilidad tienden a "aglomerarse"; es decir, los períodos de gran volatilidad no se distribuyen de manera uniforme y aleatoria a lo largo del tiempo, sino que tienden a producirse en ráfagas muy concentradas. Esta es una conclusión interesante que no podríamos obtener sólo mirando los datos en formato tabular. En casos futuros profundizaremos en las numerosas capacidades gráficas de Python y en cómo integrarlas en el flujo de trabajo de la ciencia de los datos.

Ejercicio 8 (10 min):

Escriba un script para encontrar e imprimir el mes que tiene el mayor volumen de operaciones diarias promedio para cada símbolo. Incluya también el valor del volumen promedio correspondiente a ese mes. Por ejemplo, el símbolo D tiene su mayor promedio de volumen de operaciones diarias de 6,437 millones en diciembre de 2018.

Respuesta.

Ejercicio 9 (10 min):

Hasta ahora hemos mirado la volatilidad agrupada por símbolo de la acción o por año y mes. Como nuestros datos cubren varios años, también es interesante agrupar los datos por mes calendario, ignorando el componente anual (por ejemplo, promediando todos los eneros). Esto nos permite ver si algunos puntos del año, en promedio, son más susceptibles a patrones de comercio volátiles.

Agrupe los datos por mes (ignorando el año), e identifique:

- El mes con, en promedio, la mayor volatilidad
- El mes con, en promedio, la menor volatilidad
- Cualquier patrón general que note durante todo el año

Respuesta.

Ejercicio 10 (15 min):

El punto final que nos interesa es ver los días en que:

- El retorno es alto

- El volumen de transacciones es bajo

Esto indica los días en los que el precio se movió sustancialmente pero sin cambiar mucho de manos.

Los umbrales que nos interesan son: * Volumen bajo: cualquier día con un volumen de operaciones en el 25º percentil inferior * Alto retorno: cualquier día en que el retorno esté en el percentil 75 de la opción

Escriba el código necesario para: * Calcular y añadir una variable "High/Low" para el Nivel de Volumen (el 'Low' es por debajo del percentil 25) * Calcule y añada una variable "High/Low" para el Retorno (el 'High' está por encima del percentil 75)

Describa lo que vea en términos de: * ¿Cuántas filas caen en nuestra definición de "bajo volumen"? * ¿Cuántas filas caen en nuestra definición de "alto rendimiento"? * ¿Cuántas filas caen en la combinación de la definición de alto retorno con la de bajo volumen? * ¿Cuáles son las 20 filas con el mayor retorno pero con bajo volumen? ¿Qué nota sobre estas operaciones?

(pista, puede usar el método sort_values() en `pandas` para ordenar un DataFrame por una columna específica.)

Respuesta.

Conclusiones (3 min)

Habiendo completado el análisis de los datos de las acciones del sector energético, hemos identificado una serie de pautas interesantes que relacionan la volatilidad con el volumen de transacciones. Específicamente, encontramos que los períodos de alta volatilidad también muestran un volumen muy alto. Esta tendencia es consistente en todos los símbolos.

También vimos que cada acción exhibía "agrupación de volatilidad", o sea, que los períodos de alta volatilidad tienden a agruparse. Cada una de las acciones experimentó una alta volatilidad en momentos relativamente similares, lo que sugiere que algún factor de mercado más general puede estar afectando al sector energético.

Puntos clave (5 min)

En este caso, hemos aprendido los fundamentos de la librería `pandas` en Python. Ahora sabemos cómo:

1. Leer datos desde archivos CSV
2. Agregar y manipular datos usando Pandas.
3. Analizar estadísticas de resumen y reunir información de tendencias a lo largo del tiempo.
4. Usar `matplotlib` para crear gráficos para hacer análisis visual.

En el futuro estaremos usando `pandas` consistentemente como marco para el análisis de datos (junto con otras herramientas) y así construir proyectos más complejos y resolver problemas críticos de negocios. Es fundamental que se familiarice con `pandas` tanto como sea posible y es imperativo que continúe investigando nuevos componentes de esta librería después de la finalización de este programa. Lo que hemos enseñado aquí son sólo los fundamentos esenciales de Pandas; todavía hay una gran cantidad de poder en la librería que usted descubrirá y utilizará más adelante en su desarrollo como profesional de los datos.

Recomendamos encarecidamente volver a repasar este caso algunas veces más y estudiarlo de principio a fin sin ayudas ni respuestas. Debe conocer los diversos métodos de DataFrame y Series que hemos introducido aquí, así como la forma de llevar a cabo operaciones comunes en los datos, como la búsqueda de percentiles, antes de considerar que ha "dominado" este material.