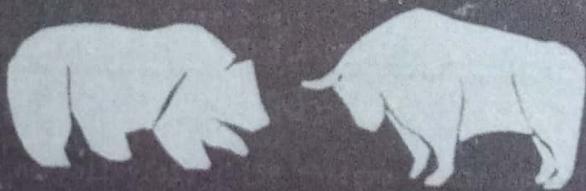


PYTHON PARA FINANZAS QUANT



Pandas DataFrames

t[1]



Juan Pablo Pisano
@johnGalt_is_www

Impreso en:
laimprentadigital.com.ar

Pandas.....	9
Importación de datos usando Pandas.....	9
Importando un XLS con pandas	11
Configurando la salida.....	13
Opciones de Formato.....	14
Ver información general del dataFrame	15
El indice de un DataFrame.....	16
Seteo del índice	16
Creación de nuevas columnas en un DataFrame	19
Creación de una columna con un valor constante	19
Creación de una columna a partir de otras columnas	20
Creación de una columna iterando el DataFrame	21
Guardado de un DataFrame a un Excel.....	22
Creación de un DataFrame de cero	23
Creación de un DataFrame a partir de una lista de listas	23
Creación de DataFrames a partir de Diccionarios.....	24
Creación a Partir de una lista de diccionarios	25
Creación a partir de otro DataFrame	26
Ejercicios.....	27
Lectura de DataFrame y sus atributos.....	29
Acceso a datos de la tabla.....	29
Métodos loc e iloc.....	29
Filtrado.....	37
Selección de un rango de fechas.....	37
Selección por filtros condicionales	38
Filtros Avanzados	38
Filtros Avanzados con OPERADORES lógicos.....	41
Ordenamiento de Tabla.....	43
Tratamiento de los elementos del DataFrame	44
Armado de diccionario partiendo de una fila de un DataFrame	46
Ejercicios.....	48
Respuestas	49
Funciones básicas de Pandas.....	55
pct_change().....	55
diff()	57
eval()	58
Shift().....	61

Datos Perdidos.....	64
Saber si tenemos datos NaN, o cuantos tenemos	64
Promedios Móviles.....	69
Método Rolling.....	69
Método ewm.....	74
Funciones Acumulativas	75
Cummax().....	75
Cummin().....	75
Cumsum().....	76
Cumprod()	77
Funciones de Conteo	79
count(), any() y all().....	79
Ejercicios Propuestos.....	83
Agrupamiento y Resampleo	97
Agrupamiento - GroupBy	97
Agrupamiento y agrupamiento combinado para conteo de un booleano.....	106
Otros tipos de agrupamiento por timestamps.....	108
Agrupamiento con filtros	109
Resampleo.....	111
Material y Archivos Usados	117
Ejercicios.....	117
Respuestas	118
Funciones Estadísticas de Pandas.....	124
Estadísticas básicas de la Tabla - Describe()	124
Repaso de funciones estadísticas	136
Desvío Estándar	137
Ratios de Riesgo básicos	139
Error estandar.....	143
Covarianza	144
Correlación	144
Mas funciones Estadísticas	146
Kurtosis:	146
Sesgo	147
Cuantiles	147
Gráficos con Pandas	148
Ejercicios Propuestos.....	151
Títulos de esta colección.....	162

Pandas

La librería más útil jamás creada, ya verán por qué

Es una librería pensada para trabajar con matrices de datos, es decir arreglos bidimensionales, o, dicho de otra forma: tablas

Es dentro de programación y Python en sí, lo más parecido que van a ver a una planilla Excel, ya que vamos a trabajar acá con datos agrupados como en una tabla de Excel, con filas y columnas

Importación de datos usando Pandas

importando la librería Pandas en la primera línea ya en la segunda podemos levantar los datos de cualquier CSV o XLS vemos esto en ejemplos sencillos, para eso vamos a utilizar el archivo AAPL.csv y AAPL.xls que preparé para este módulo y lo pueden descargar de <http://bit.ly/39Am64T> (<http://bit.ly/39Am64T>)

Tal como en la librería CSV algo que le vamos a tener que indicar a Pandas es el carácter que usamos de separador ya que recuerden que en realidad CSV es un formato de texto plano, si no se aclara este dato por default se toma la coma "," aunque si se guarda como CSV directo desde Excel es probable que se les guarde con ":" como separador, esto dependerá de las configuraciones regionales y ese tipo de cosas del office

```
import pandas as pd
data = pd.read_csv('AAPL.csv', sep=',')
data
```

	timestamp	open	high	low	close	adjusted_close	volume
0	6/3/2020	282.00	290.82	281.23	289.03	289.0300	56544246
1	5/3/2020	295.52	299.55	291.41	292.92	292.9200	46893219
2	4/3/2020	296.44	303.40	293.13	302.74	302.7400	54794568
3	3/3/2020	303.67	304.00	285.80	289.32	289.3200	79868852
4	2/3/2020	282.28	301.44	277.72	298.81	298.8100	85349339
...
5028	10/3/2000	121.69	127.94	121.00	125.75	3.8979	2219700
5029	9/3/2000	120.87	125.00	118.25	122.25	3.7894	2470700
5030	8/3/2000	122.87	123.94	118.56	122.00	3.7816	2421700
5031	7/3/2000	126.44	127.44	121.12	122.87	3.8086	2437600
5032	6/3/2000	126.00	129.13	125.00	125.69	3.8960	1880000

5033 rows × 7 columns

Veamos qué pasa si importo otro CSV que tiene punto y coma en lugar de coma y no se lo indico

```
import pandas as pd
data = pd.read_csv('SPY.csv')
data
```

	timestamp;open;high;low;close;adjusted_close;volume
0	19/3/2020;239.25;247.38;232.22;240.51;240.51;2...
1	18/3/2020;236.25;248.37;228.02;240;240;324845381
2	17/3/2020;245.04;256.17;237.07;252.8;252.8;260...
3	16/3/2020;241.18;256.9;237.36;239.85;239.85;29...
...	...
5028	23/3/2000;149.1562;153.4687;149.1562;152.6562;...
5029	22/3/2000;149.5625;150.8437;148.6875;150.0937;...
5030	21/3/2000;145.5312;149.75;144.5;149.1875;102.7...
5031	20/3/2000;146.875;147.3437;144.7812;146.1875;1...

5032 rows × 1 columns

Como verán en el ejemplo al no indicarle nada buscó la coma como separador, y al no encontrar ninguna coma me trajo todas las columnas como si fuese una sola columna del tipo string con todo el choclo de datos y sus ";" intercalados entre dato y dato

Obviamente que para que me lo importe bien tengo que poner `sep=";"` y listo, veamoslo:

```
import pandas as pd
data = pd.read_csv('SPY.csv', sep=";")
data
```

	timestamp	open	high	low	close	adjusted_close	vol
0	19/3/2020	239.2500	247.3800	232.2200	240.5100	240.5100	28812
1	18/3/2020	236.2500	248.3700	228.0200	240.0000	240.0000	32484
2	17/3/2020	245.0400	256.1700	237.0700	252.8000	252.8000	26056
3	16/3/2020	241.1800	256.9000	237.3600	239.8500	239.8500	29501
...
5028	23/3/2000	149.1562	153.4687	149.1562	152.6562	105.1870	1165
5029	22/3/2000	149.5625	150.8437	148.6875	150.0937	103.4213	826
5030	21/3/2000	145.5312	149.7500	144.5000	149.1875	102.7969	1361
5031	20/3/2000	146.8750	147.3437	144.7812	146.1875	100.7297	1250

5032 rows × 7 columns

Importando un XLS con pandas

El caso de importar un XLS como dataframes es un poco más completo porque en este caso el objeto XLS es más completo que un archivo CSV ya que por ejemplo tenemos múltiples hojas de cálculo en el mismo libro o archivo, por lo tanto, como se imaginarán el nombre de la hoja que queremos abrir vamos a tener que pasárselo como argumento para que pandas sepa que hoja queremos tomar los datos

Si no le pasamos el argumento `sheet_name` nos va a traer lo que encuentre en la primera hoja del libro

Veamos la importación de los precios de AAPL tal como hicimos cuando importamos el CSV, pero importando lo mismo de un Excel

```
data = pd.read_excel('AAPL.xlsx', sheet_name='Hoja1')
data
```

	timestamp	open	high	low	close	adjusted_close	volume
0	2020-03-06	282.00	290.82	281.23	289.03	289.0300	56544246
1	2020-03-05	295.52	299.55	291.41	292.92	292.9200	46893219
2	2020-03-04	296.44	303.40	293.13	302.74	302.7400	54794568
3	2020-03-03	303.67	304.00	285.80	289.32	289.3200	79868852
4	2020-03-02	282.28	301.44	277.72	298.81	298.8100	85349339
...
5028	2000-03-10	121.69	127.94	121.00	125.75	3.8979	2219700
5029	2000-03-09	120.87	125.00	118.25	122.25	3.7894	2470700
5030	2000-03-08	122.87	123.94	118.56	122.00	3.7816	2421700
5031	2000-03-07	126.44	127.44	121.12	122.87	3.8086	2437600
5032	2000-03-06	126.00	129.13	125.00	125.69	3.8960	1880000

5033 rows × 7 columns

Vamos a traer la misma tabla, pero indicando que el índice sea la fecha en lugar de un índice numérico, para ello solo usamos el atributo `index_col`

Y como podrán ver, indicando `index_col="nombre_columna_indice"` al traerme la tabla me muestra la columna indicada como índice

```
data = pd.read_excel('AAPL.xlsx', sheet_name='Hoja1', index_col="timestamp")
data
```

	open	high	low	close	adjusted_close	volume
timestamp						
2020-03-06	282.00	290.82	281.23	289.03	289.0300	56544246
2020-03-05	295.52	299.55	291.41	292.92	292.9200	46893219
2020-03-04	296.44	303.40	293.13	302.74	302.7400	54794568
2020-03-03	303.67	304.00	285.80	289.32	289.3200	79868852
...
2000-03-09	120.87	125.00	118.25	122.25	3.7894	2470700
2000-03-08	122.87	123.94	118.56	122.00	3.7816	2421700
2000-03-07	126.44	127.44	121.12	122.87	3.8086	2437600
2000-03-06	126.00	129.13	125.00	125.69	3.8960	1880000

5033 rows × 6 columns

También podemos traer solamente una determinada cantidad de columnas que necesitemos, para ello usamos el atributo `usecols` y le pasamos la lista de columnas que queremos traer como argumento

```
columnas = ["timestamp","open","close"]
data = pd.read_excel('AAPL.xlsx', sheet_name='Hoja1', index_col="timestamp",
usecols = columnas)
data
```

	open	close
timestamp		
2020-03-06	282.00	289.03
2020-03-05	295.52	292.92
2020-03-04	296.44	302.74
...
2000-03-08	122.87	122.00
2000-03-07	126.44	122.87
2000-03-06	126.00	125.69

5033 rows × 2 columns

Configurando la salida

Como habrán notado a esta altura, cuando mandamos a imprimir o mandamos a pantalla una variable que es un DataFrame con muchas filas, por defecto Pandas nos muestra 10 filas en total, las primeras 5 y las últimas 5 filas, algo similar hace con las columnas y claramente esto es un valor de default que se puede configurar, por lo general es útil este valor, pero a veces vamos a preferir más o menos, por lo tanto veamos como configurar estas cosas

En el siguiente ejemplo vamos a limitar la cantidad de filas o de columnas que quiero que me muestra en pantalla

Para ello usamos el objeto `pd` que es un objeto de pandas vacío antes de llenarlo con la tabla del Excel

El objeto `pd` vamos a cambiarle opciones y dentro de las opciones vamos a cambiarle las de `display` y dentro de las de `display` veremos la cantidad máxima de filas y columnas

```
pd.options.display.max_rows = 4
pd.options.display.max_columns = 6

data = pd.read_excel('AAPL.xlsx', sheet_name='Hoja1')
data
```

	timestamp	open	high	...	close	adjusted_close	volume
0	2020-03-06	282.00	290.82	...	289.03	289.0300	56544246
1	2020-03-05	295.52	299.55	...	292.92	292.9200	46893219
...
5031	2000-03-07	126.44	127.44	...	122.87	3.8086	2437600
5032	2000-03-06	126.00	129.13	...	125.69	3.8960	1880000

5033 rows × 7 columns

Si queremos que el número de filas o columnas a mostrar sea ilimitado, es decir que muestre siempre todo, lo configuramos como `None`

```
pd.options.display.max_columns= None
pd.options.display.max_rows= 6
data
```

	timestamp	open	high	low	close	adjusted_close	volume
0	2020-03-06	282.00	290.82	281.23	289.03	289.0300	56544246
1	2020-03-05	295.52	299.55	291.41	292.92	292.9200	46893219
2	2020-03-04	296.44	303.40	293.13	302.74	302.7400	54794568
...
5030	2000-03-08	122.87	123.94	118.56	122.00	3.7816	2421700
5031	2000-03-07	126.44	127.44	121.12	122.87	3.8086	2437600
5032	2000-03-06	126.00	129.13	125.00	125.69	3.8960	1880000

Opciones de Formato

Podemos modificar también el formato de salida más que nada la cantidad de decimales que muestra, a veces puede ser medio denso que muestre muchos decimales y facilita a la vista que muestre solo 2 o 1 o ninguno

Cabe destacar que redondea los valores, es decir que el numero 3.45678 si lo mostramos con 3 decimales lo mostrará como 3.457

Si le indicamos un numero de decimales mayor a la que tienen todas las columnas, no hará nada

```
pd.options.display.precision = 4
data
```

	timestamp	open	high	low	close	adjusted_close	volume
0	2020-03-06	282.00	290.82	281.23	289.03	289.0300	56544246
1	2020-03-05	295.52	299.55	291.41	292.92	292.9200	46893219
2	2020-03-04	296.44	303.40	293.13	302.74	302.7400	54794568
...
5030	2000-03-08	122.87	123.94	118.56	122.00	3.7816	2421700
5031	2000-03-07	126.44	127.44	121.12	122.87	3.8086	2437600
5032	2000-03-06	126.00	129.13	125.00	125.69	3.8960	1880000

5033 rows × 7 columns

Mas opciones de formato

```
pd.options.display.precision = 2
data
```

	timestamp	open	high	low	close	adjusted_close	volume
0	2020-03-06	282.00	290.82	281.23	289.03	289.03	56544246
1	2020-03-05	295.52	299.55	291.41	292.92	292.92	46893219
2	2020-03-04	296.44	303.40	293.13	302.74	302.74	54794568
...
5030	2000-03-08	122.87	123.94	118.56	122.00	3.78	2421700
5031	2000-03-07	126.44	127.44	121.12	122.87	3.81	2437600
5032	2000-03-06	126.00	129.13	125.00	125.69	3.90	1880000

5033 rows × 7 columns

Ver información general del DataFrame

El método info () nos va a devolver la información de la cantidad de columnas y de filas de cada una de ellas con valores válidos y asimismo nos va a devolver los tipos de datos de cada columna y los nombres de columna, también nos va a decir que espacio ocupa en memoria el DataFrame

```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5033 entries, 0 to 5032
Data columns (total 7 columns):
 #   Column      Non-Null Count  Dtype  
 ---  --          -----          ...
 0   timestamp   5033 non-null    datetime64[ns]
 1   open        5033 non-null    float64 
 2   high        5033 non-null    float64 
 3   low         5033 non-null    float64 
 4   close       5033 non-null    float64 
 5   adjusted_close  5033 non-null    float64 
 6   volume      5033 non-null    int64  
dtypes: datetime64[ns](1), float64(5), int64(1)
memory usage: 275.4 KB
```

Si simplemente queremos obtener la cantidad de filas y columnas de la tabla podemos usar el atributo shape

```
data.shape
(5033, 7)
```

Si simplemente queríamos acceder a los nombres de las columnas usamos el atributo columns
Para visualizarlo o accederlo de forma más cómoda lo podemos pasar a lista

```
data.columns
```

```
Index(['timestamp', 'open', 'high', 'low', 'close',
       'adjusted_close', 'volume'], dtype='object')
```

```
list(data.columns)
```

```
['timestamp', 'open', 'high', 'low', 'close', 'adjusted_close', 'volume']
```

Así como puedo acceder a las columnas, también del mismo modo podría modificarlas, en lugar de solo verlas, digo, data.columns = ["columna 1", "columna 2",..... etc]

Y de ese modo les estaría cambiando los nombres de las columnas

Si solamente queremos acceder a los tipos de datos de cada columna usamos el atributo `dtypes`
 Para visualizarlo o accederlo más cómodo lo podemos pasar a diccionario

```
data.dtypes
```

```
timestamp      datetime64[ns]
open           float64
high           float64
...
close          float64
adjusted_close float64
volume         int64
Length: 7, dtype: object
```

```
dict(data.dtypes)
```

```
{'timestamp': dtype('M8[ns]'),
 'open': dtype('float64'),
 'high': dtype('float64'),
 'low': dtype('float64'),
 'close': dtype('float64'),
 'adjusted_close': dtype('float64'),
 'volume': dtype('int64')}
```

El índice de un DataFrame

Como iremos viendo más adelante el índice de una tabla no es solo la primera columna, sino una columna muy especial que será la que el motor de python use para ordenar la tabla, para buscar u valor, para resamplear una muestra y muchas otras cosas que iremos viendo, digamos que es la columna más importante de todas y por lo general son siempre valores únicos e irrepetibles, es decir que podríamos referirnos a una fila simplemente pidiéndole por el valor en el índice, ya que son todos valores únicos

Seteo del índice

Por lo general se usan dos tipos de índices usuales

El índice de Fecha

Un índice numérico creciente

Vemos que nuestro DataFrame viene con un índice numérico (arranca de 0 la primera fila que encuentra)

```
import pandas as pd
data = pd.read_excel('AAPL.xlsx', sheet_name='Hoja1')
data
```

	timestamp	open	high	low	close	adjusted_close	volume
0	2020-03-06	282.00	290.82	281.23	289.03	289.03	56544246
1	2020-03-05	295.52	299.55	291.41	292.92	292.92	46893219
2	2020-03-04	296.44	303.40	293.13	302.74	302.74	54794568
...
5030	2000-03-08	122.87	123.94	118.56	122.00	3.78	2421700
5031	2000-03-07	126.44	127.44	121.12	122.87	3.81	2437600
5032	2000-03-06	126.00	129.13	125.00	125.69	3.90	1880000

5033 rows × 7 columns

Pero le vamos a cambiar ese índice por el valor de la columna que tiene la fecha que en nuestro caso se llama timestamp

```
data.set_index('timestamp', inplace=True)
```

	open	high	low	close	adjusted_close	volume
timestamp						
2020-03-06	282.00	290.82	281.23	289.03	289.03	56544246
2020-03-05	295.52	299.55	291.41	292.92	292.92	46893219
2020-03-04	296.44	303.40	293.13	302.74	302.74	54794568
...

5033 rows × 6 columns

Ese atributo que vemos ahí "inplace = True" lo que hace es asignarle el seteo en la misma línea a la variable "data" sin tener que poner "data = "

Ya que es equivalente a poner: (sin el atributo inplace)

```
Data = data.set_index('timestamp')
```

	open	high	low	close	adjusted_close	volume
timestamp						
2020-03-06	282.00	290.82	281.23	289.03	289.03	56544246
2020-03-05	295.52	299.55	291.41	292.92	292.92	46893219
2020-03-04	296.44	303.40	293.13	302.74	302.74	54794568
...

Otro atributo que podemos pasarle a la función "set_index" es drop, que por default es False, como imaginarán lo que hace el atributo "drop" es borrar la columna del índice anterior

O volvemos atrás de la siguiente forma

```
data = data.reset_index()
```

	timestamp	open	high	low	close	adjusted_close	volume
0	2020-03-06	282.00	290.82	281.23	289.03	289.03	56544246
1	2020-03-05	295.52	299.55	291.41	292.92	292.92	46893219
2	2020-03-04	296.44	303.40	293.13	302.74	302.74	54794568

5033 rows × 7 columns

O también como les decía, podemos aplicar el atributo drop, veámoslo en el ejemplo:

```
data = data.reset_index(drop=True)
```

	open	high	low	close	adjusted_close	volume
0	282.00	290.82	281.23	289.03	289.03	56544246
1	295.52	299.55	291.41	292.92	292.92	46893219
2	296.44	303.40	293.13	302.74	302.74	54794568

5030	122.87	123.94	118.56	122.00	3.78	2421700
5031	126.44	127.44	121.12	122.87	3.81	2437600
5032	126.00	129.13	125.00	125.69	3.90	1880000

Esto de usar el atributo inplace, es muy cómodo en algunas ocasiones en donde concatenamos métodos en la misma línea, ahora parece un sinsentido porque me dirán ¿para que voy a usar ese atributo si directamente puedo poner data=.... y con eso me desligo de tener que usar el atributo

Bueno, son formas de escribir código diferente, si no usamos el atributo inplace, lo que hacemos es sobreescribir toda la variable, en cambio si usamos el atributo lo que hacemos es diferente, en realidad lo que hace el intérprete por detrás y esto se nota en la performance cuando hay varios métodos concatenados o cuando las matrices de datos son pesadas, en fin, por ahora es un detalle menor al que no vamos a prestar importancia pero ténganlo en cuenta

Creación de nuevas columnas en un DataFrame

Creación de una columna con un valor constante

Si bien es muy poco usual hacer este tipo de cosas, puede ser útil saber hacerlo para generar una columna con un determinado valor de default por poner un ejemplo típico

Como vemos en el ejemplo que sigue al setear el índice usamos el atributo inplace=True, esto es para que en la misma línea haga el seteo sin necesidad de asignar el resultado del mismo a la misma variable data, es un seteo con asignación inline como se dice

```
import pandas as pd
data = pd.read_excel('AAPL.xlsx', sheet_name='Hoja1')
data.set_index('timestamp', inplace=True)
data['Precio_COME']=3
data.drop(["high","low"], axis=1).head()
```

	open	close	adjusted_close	volume	Precio_COME
timestamp					
2020-03-06	282.00	289.03	289.03	56544246	3
2020-03-05	295.52	292.92	292.92	46893219	3
2020-03-04	296.44	302.74	302.74	54794568	3
2020-03-03	303.67	289.32	289.32	79868852	3
2020-03-02	282.28	298.81	298.81	85349339	3

Lo del drop(["high","low"], axis=1) es simplemente borrar esas dos columnas para que me entre mejor en la página la tabla y limpiar un poco de datos que no necesito ver y además para ir mostrando herramientas de a poco

drop es borrar obviamente, el listado de columnas va entre corchetes porque es una lista, y luego uso axis=1 indicando que me refiero a columnas, ya que si quiero borrar filas usa axis=0 que es el valor de default de la variable axis

Tengan en cuenta siempre lo siguiente, cuando yo escribo en la Jupyter notebook:

```
data.drop(["high","low"], axis=1).head()
```

Directamente me "da salida" y me imprime la tabla como vimos, pero si lo hago en un IDE sin poner print no me devuelve nada.. pero hay mas, al hacerlo así sin haberle asignado esa salida a "data" si ahora vuelvo a pedir que imprima "data" van a aparecer las columnas borradas nuevamente, porque lo que hice fue solamente ver la salida de la línea pero no se lo asigné a la variable, para ello debería haber escrito:

```
data = data.drop(["high","low"], axis=1).head()
```

Creación de una columna a partir de otras columnas

También se puede crear una columna nueva partiendo de los valores de otra u otras columnas
Veamos unos ejemplos

Creamos una columna que muestre el precio medio entre apertura, cierre, máximo y mínimo

```
data = pd.read_excel('AAPL.xlsx', sheet_name='Hoja1')
data.set_index('timestamp', inplace=True)

data['precio_medio'] = (data.open + data.close + data.low + data.high) / 4
data.drop(['high', 'low'], axis=1).head()
```

	open	close	adjusted_close	volume	precio_medio
timestamp					
2020-03-06	282.00	289.03	289.03	56544246	285.7700
2020-03-05	295.52	292.92	292.92	46893219	294.8500
2020-03-04	296.44	302.74	302.74	54794568	298.9275
2020-03-03	303.67	289.32	289.32	79868852	295.6975
2020-03-02	282.28	298.81	298.81	85349339	290.0625

Armemos ahora una columna que muestre el volumen diario en millones de usd en lugar de nominales, usando el precio medio de cada rueda

```
data = pd.read_excel('AAPL.xlsx', sheet_name='Hoja1')
data.set_index('timestamp', inplace=True)

data['precio_medio'] = (data.open + data.close + data.low + data.high) / 4
data['vol_mln_usd'] = round((data.volume * data.precio_medio) / 1000000)
data.drop(['open', 'high', 'low', 'close'], axis=1).head()
```

	adjusted_close	volume	precio_medio	vol_mln_usd
timestamp				
2020-03-06	289.03	56544246	285.7700	16159.0
2020-03-05	292.92	46893219	294.8500	13826.0
2020-03-04	302.74	54794568	298.9275	16380.0
2020-03-03	289.32	79868852	295.6975	23617.0
2020-03-02	298.81	85349339	290.0625	24757.0

Creación de una columna iterando el DataFrame

Si bien es un recurso muy usado, en realidad es poco eficiente cuando ya estamos trabajando con Pandas, pero por ahora no nos vamos a ocupar de la eficiencia, aunque no quería dejar de mencionarlo, por lo general cualquier tipo de cálculo que queramos hacer partiendo de datos de una columna para generar otra columna, se va a poder hacer con funciones específicas de Pandas que ya iremos viendo

Veamos a modo de ejemplo como podría generar una nueva columna al DataFrame iterando el mismo

Vamos a crear una columna que informe el color de la vela de cada rueda

Recordamos que la vela es verde cuando el precio de cierre es mayor al de apertura y es roja cuando pasa lo opuesto

En el ejemplo verán que dentro del bucle FOR nos referimos a "locaciones" .loc [fila, columna] ya lo veremos más adelante en detalle, lo puse acá para mostrarles el ejemplo

Pero como se darán cuenta loc[] nos permite acceder a determinados lugares o locaciones del DataFrame, es como el equivalente a las celdas del Excel, cuando ponían "B5" se referían a la columna "B" y la fila "5", bueno, acá en pandas usaremos algo así como .loc [5,"B"] "digamos"

```
data = pd.read_excel('AAPL.xlsx', sheet_name='Hoja1')

for i in range(len(data)):
    if data.loc[i,'close'] >= data.loc[i, 'open']:
        data.loc[i,'color_vela']="verde"
    else:
        data.loc[i,'color_vela']="roja"
data.drop(["high","low"], axis=1).head(8)
```

	timestamp	open	close	adjusted_close	volume	color_vela
0	2020-03-06	282.00	289.03	289.03	56544246	verde
1	2020-03-05	295.52	292.92	292.92	46893219	roja
2	2020-03-04	296.44	302.74	302.74	54794568	verde
3	2020-03-03	303.67	289.32	289.32	79868852	roja
4	2020-03-02	282.28	298.81	298.81	85349339	verde
5	2020-02-28	257.26	273.36	273.36	106721230	verde
6	2020-02-27	281.10	273.52	273.52	80151381	roja
7	2020-02-26	286.53	292.65	292.65	49678431	verde

Guardado de un DataFrame a un Excel

Vamos a agregar una columna a la tabla con la que venimos trabajando, y luego guardar el DataFrame modificado, que como se imaginarán en lugar de `read_excel` será algo parecido a `"write_excel"` y si, en este caso es `"to_excel"` el método en cuestión, y no mucha más ciencia, pero veámoslo con un ejemplo rápido

Primero vamos a generar una columna nueva, una manera rápida de hacerlo es tomar al DataFrame como si fuera un diccionario bidimensional y asignarle a la clave `"movimiento_diario"` la diferencia entre el precio de cierre de cada día menos el de apertura

```
import pandas as pd
data = pd.read_excel('AAPL.xlsx', sheet_name='Hoja1')
data['mov_intra'] = data.close - data.open
data.drop(['high', 'low', 'adjusted_close'], axis=1).head()
```

timestamp	open	close	volume	mov_intra
0 2020-03-06	282.00	289.03	56544246	7.03
1 2020-03-05	295.52	292.92	46893219	-2.60
2 2020-03-04	296.44	302.74	54794568	6.30
3 2020-03-03	303.67	289.32	79868852	-14.35
4 2020-03-02	282.28	298.81	85349339	16.53

No sé si se sorprendieron o no de la simpleza con que Pandas nos permite trabajar matrices, la verdad que a mi cuando lo vi me dejó con la boca abierta porque antes los programas no tenían tanta abstracción de código como ahora y no "entendían" de forma tan sencilla lo que queríamos hacer.

Pandas está preparado para trabajar con DataFrames, es decir con tables de datos, por lo que es obvio o presupone que cuando generemos un nuevo atributo lo vamos a querer para toda la tabla, así que de forma tan sencilla como decirle resta lo que tiene en una columna y otra y meterlo en una tercera columna, la librería ya lo entiende y procesa los datos internamente de la manera más eficiente posible para que todo ese cálculo e interacciones se haga con la mejor performance computacional, no sé si notaron pero hace todo muy rápido, después veremos ejemplos más complejos igual.

Bueno, dicho esto entonces vamos a guardar en un Excel nuevo esta tabla con ese agregado que hicimos, para eso como anticipamos vamos a utilizar el método `to_excel` de la librería Pandas

```
data.to_excel('AAPL_Modificado.xlsx', sheet_name='HojaEjemplo')
```

Creación de un DataFrame de cero

Muchas veces nos vamos a ver en la necesidad de ir armando nuestros dataframes a veces de cero e ir llenándolos, y a veces partiendo de listas, diccionarios o estructuras similares, empecemos a ver algunos casos

Creación de un DataFrame a partir de una lista de listas

```
# Supongamos que partimos de esta Lista de datos:
data = [["ALUA",19.15],["BBAR",73.70],["BMA",144.4],[["BYMA",234]]
```

Y ahora quiero armar un DataFrame en la variable "tabla" con esa estructura que tiene tickers y sus precios

Bien, es tan sencillo como esto:

1. Asignamos a la variable "tabla" el objeto pandas.DataFrame
2. Pasamos como primer argumento del DataFrame los datos (variable "data")
3. Le indicamos en el argumento "columns" los nombres de las columnas

```
import pandas as pd
data = [["ALUA",19.15],["BBAR",73.70],["BMA",144.4],[["BYMA",234]]
tabla = pd.DataFrame(data, columns = ['Ticker', 'Precio'])
tabla
```

	Ticker	Precio
0	ALUA	19.15
1	BBAR	73.70
2	BMA	144.40
3	BYMA	234.00

También podemos crear el DataFrame de forma muy sencilla a partir de un diccionario
En este caso ni siquiera hace falta que le pasemos los nombres de las columnas ya que la toma del mismo diccionario

Creación de DataFrames a partir de Diccionarios

```
data = {'Tickers':['ALUA', 'BBAR', 'BMA', 'BYMA'], 'Precios':[19.15,73.7 , 14
4.4, 234]}
tabla = pd.DataFrame(data)
tabla
```

	Tickers	Precios
0	ALUA	19.15
1	BBAR	73.70
2	BMA	144.40
3	BYMA	234.00

Asimismo, le podemos pasar un índice personalizado (una lista de elementos que tenga la misma cantidad de elementos que "data")

```
data = {
'Tickers':['ALUA', 'BBAR', 'BMA', 'BYMA'],
'Precios':[19.15,73.7 , 14 4.4, 234]
}
tabla = pd.DataFrame(data, index=["activo_1","activo_2","activo_3","activo_4"])
tabla
```

	Tickers	Precios
activo_1	ALUA	19.15
activo_2	BBAR	73.70
activo_3	BMA	144.40
activo_4	BYMA	234.00

Creación a Partir de una lista de diccionarios

Esta estructura de entrada es más rara, pero también es posible recibir datos de esta forma algún día y por suerte pandas se encarga de transformar todo esto en un data frame de la misma forma sencilla

```
data = [
    {"ticker": "ALUA", "Precio": 19.15, "Tipo": "Accion"},
    {"ticker": "BBAR", "Precio": 73.7, "Tipo": "Accion"},
    {"ticker": "BMA", "Precio": 144.4},
    {"ticker": "BYMA", "Precio": 234, "Tipo": "Accion"},
]

tabla = pd.DataFrame(data)
tabla
```

	ticker	Precio	Tipo
0	ALUA	19.15	Accion
1	BBAR	73.70	Accion
2	BMA	144.40	NaN
3	BYMA	234.00	Accion

También puedo al crearla definir qué columnas quiero

```
data = [
    {"ticker": "ALUA", "Precio": 19.15, "Tipo": "Accion"},
    {"ticker": "BBAR", "Precio": 73.7, "Tipo": "Accion"},
    {"ticker": "BMA", "Precio": 144.4},
    {"ticker": "BYMA", "Precio": 234, "Tipo": "Accion"},
]
tabla = pd.DataFrame(data, columns=["ticker", "Precio"])
tabla
```

	ticker	Precio
0	ALUA	19.15
1	BBAR	73.70
2	BMA	144.40
3	BYMA	234.00

Creación a partir de otro DataFrame

Partiendo de un DataFrame se puede armar otro con las columnas específicas que le pasemos e incluso en la misma línea podemos definir cuál va a ser el índice del DataFrame nuevo

```
data = pd.read_excel('AAPL.xlsx', sheet_name='Hoja1')
copia = data[['timestamp', 'open', 'close']].copy().set_index("timestamp")
copia.head(4)
```

	open	close
timestamp		
2020-03-06	282.00	289.03
2020-03-05	295.52	292.92
2020-03-04	296.44	302.74
2020-03-03	303.67	289.32

Algo idéntico produce el método filter(), al que le pasamos como argumento la lista de columnas

```
data = pd.read_excel('AAPL.xlsx', sheet_name='Hoja1')
copia = data.filter(['timestamp', 'open', 'close']).set_index("timestamp")
copia.head(4)
```

	open	close
timestamp		
2020-03-06	282.00	289.03
2020-03-05	295.52	292.92
2020-03-04	296.44	302.74
2020-03-03	303.67	289.32

Algo parecido, pero "al revés" sería crear la copia a partir del original MENOS las columnas que no queremos

```
data = pd.read_excel('AAPL.xlsx', sheet_name='Hoja1')
copia = data.drop(['high', 'low', 'adjusted_close'], axis=1)
copia.head()
```

	timestamp	open	close	volume
0	2020-03-06	282.00	289.03	56544246
1	2020-03-05	295.52	292.92	46893219
2	2020-03-04	296.44	302.74	54794568
3	2020-03-03	303.67	289.32	79868852
4	2020-03-02	282.28	298.81	85349339

Ejercicios

- 1- Abrir el archivo AAPL.xlsx generar un DataFrame igual agregando una columna que contenga la variación porcentual intradiaria (entre precio de apertura y cierre sobre el precio de apertura)
- 2- Mostrar la tabla en pantalla, de manera que solo muestre 2 decimales de la columna calculada, y que solo muestre las dos primeras y las dos últimas filas
- 3- Guardar la tabla en un archivo llamado AAPL_ej.xlsx en la hoja llamada Ej3, y guardar solo las columnas de la fecha y la variación intradiaria que acabamos de calcular
Ayuda: Para indicarle al método to_excel que columnas quiero guardar puedo usar el atributo "columns" (si, así de intuitivo es todo lo que tiene pandas, enjoy it)

```

#-----#
# Rta Ejercicio 1 #
# -----#

import pandas as pd
data = pd.read_excel('AAPL.xlsx')
data['variacion_intra'] = 100 * (data.close - data.open) / data.open
data.drop(["high","low"], axis=1).head()

```

	timestamp	open	close	adjusted_close	volume	variacion_intra
0	2020-03-06	282.00	289.03	289.03	56544246	2.492908
1	2020-03-05	295.52	292.92	292.92	46893219	-0.879805
2	2020-03-04	296.44	302.74	302.74	54794568	2.125219
3	2020-03-03	303.67	289.32	289.32	79868852	-4.725524
4	2020-03-02	282.28	298.81	298.81	85349339	5.855888

```

#-----#
# Rta Ejercicio 2 #
# -----#

pd.options.display.max_rows = 4
pd.options.display.precision = 2

data.drop(["high","low"], axis=1)

```

	timestamp	open	close	adjusted_close	volume	variacion_intra
0	2020-03-06	282.00	289.03	289.03	56544246	2.49
1	2020-03-05	295.52	292.92	292.92	46893219	-0.88
...
5031	2000-03-07	126.44	122.87	122.87	3.81	2437600
5032	2000-03-06	126.00	125.69	125.69	3.90	1880000

5033 rows × 6 columns

```

#-----#
# Rta Ejercicio 3 #
# -----#

columnas = ["timestamp","variacion_intra"]
data.to_excel('AAPL_Ej.xlsx', sheet_name='Ej3', columns = columnas)

```

Lectura de DataFrame y sus atributos

Acceso a datos de la tabla

Head(n) / Tail(n)

Estos métodos nos permiten ver los primeros/últimos "n" valores de una tabla respectivamente

```
import pandas as pd
pd.options.display.max_rows = 8
data = pd.read_excel('AAPL.xlsx', sheet_name='Hoja1')
data.head(3)
```

	timestamp	open	high	low	close	adjusted_close	volume
0	2020-03-06	282.00	290.82	281.23	289.03	289.03	56544246
1	2020-03-05	295.52	299.55	291.41	292.92	292.92	46893219
2	2020-03-04	296.44	303.40	293.13	302.74	302.74	54794568

Si en lugar de head usamos tail, obviamente vamos a acceder a la cola de los datos

```
data.tail(4)
```

	timestamp	open	high	low	close	adjusted_close	volume
5029	2000-03-09	120.87	125.00	118.25	122.25	3.7894	2470700
5030	2000-03-08	122.87	123.94	118.56	122.00	3.7816	2421700
5031	2000-03-07	126.44	127.44	121.12	122.87	3.8086	2437600
5032	2000-03-06	126.00	129.13	125.00	125.69	3.8960	1880000

Métodos loc e iloc

Con estos métodos accederemos a cuallquier elemento o rango de elementos de un DataFrame
La estructura es la siguiente: `data.iloc[filas, columnas]`

La diferencia entre "loc" e "iloc" es que "iloc" llama a los indices y "loc" llama a los nombres de los elementos

Por lo general a las filas se accede con índices numéricos o en su defecto de fechas
mientras que las columnas pueden llamarse de ambas formas

Empecemos por las filas

Seleccionamos un solo elemento

Para seleccionar un solo elemento usamos la locacion concreta a la que queremos referirnos,

La locación `loc` es el valor del índice

La locacion `iloc` es el número de elemento (fila) del datafram, empezando a contar desde el 0

```
data.loc[2]
```

```
timestamp      2020-03-04 00:00:00
open            296.44
high            303.4
low             293.13
close           302.74
adjusted_close  302.74
volume          54794568
Name: 2, dtype: object
```

```
data.iloc[2]
```

```
timestamp      2020-03-04 00:00:00
open            296.44
high            303.4
low             293.13
close           302.74
adjusted_close  302.74
volume          54794568
Name: 2, dtype: object
```

Como podemos ver la locación `loc` y la locación del valor índice `iloc` en este caso coincide, esto es así porque nuestro índice es un índice numérico

Para entender mejor la diferencia entre loc e iloc vamos a crear una tabla o DataFrame igual a la anterior, pero le vamos a poner como índice la fecha en lugar de el índice numérico con que viene la tabla original

```
import pandas as pd
pd.options.display.max_rows = 8
data = pd.read_excel('AAPL.xlsx',
sheet_name='Hoja1') dataIndiceFecha =
data.set_index("timestamp") dataIndiceFecha
```

	open	high	low	close	adjusted_close	volume
timestamp						
2020-03-06	282.00	290.82	281.23	289.03	289.0300	56544246
2020-03-05	295.52	299.55	291.41	292.92	292.9200	46893219
2020-03-04	296.44	303.40	293.13	302.74	302.7400	54794568
2020-03-03	303.67	304.00	285.80	289.32	289.3200	79868852
...
2000-03-09	120.87	125.00	118.25	122.25	3.7894	2470700
2000-03-08	122.87	123.94	118.56	122.00	3.7816	2421700
2000-03-07	126.44	127.44	121.12	122.87	3.8086	2437600
2000-03-06	126.00	129.13	125.00	125.69	3.8960	1880000

5033 rows × 6 columns

Ahora vamos a acceder al tercer valor de la serie, el del 5 de marzo, con los métodos loc e iloc respectivamente

En el caso del iloc sigue siendo igual que antes, ya que este método llama al "n" elemento de la tabla, y no depende del índice

Ahora al referirnos a la locación del índice, como cambiamos el índice ya no puedo llamar al elemento "2" porque ese elemento ya no existe, porque va a ir a buscar ese elemento al índice y se va a encontrar con solo valores de fecha, entonces aca le tengo que poner la fecha exacta que quiero que me devulva

```
dataIndiceFecha.loc["2020-03-04"]
```

	open	high	low	close	adjusted_close	volume
timestamp						
2020-03-04	296.44	303.4	293.13	302.74	302.74	54794568

```
dataIndiceFecha.iloc[2]
```

open	296.44
high	303.40
low	293.13
close	302.74
adjusted_close	302.74
volume	54794568.00
Name:	2020-03-04 00:00:00, dtype: float64

Seleccionamos algunos elementos cualquiera de la tabla

Para ello le pedimos un listado de elementos separados con coma, como todo listado lo ponemos entre []

```
data.iloc[[2,4]]
```

	timestamp	open	high	low	close	adjusted_close	volume
2	2020-03-04	296.44	303.40	293.13	302.74	302.74	54794568
4	2020-03-02	282.28	301.44	277.72	298.81	298.81	85349339

Seleccionamos un rango de filas dentro de una tabla

Para ello, indicamos el rango como desde:hasta

```
data.loc[2:4]
```

	timestamp	open	high	low	close	adjusted_close	volume
2	2020-03-04	296.44	303.40	293.13	302.74	302.74	54794568
3	2020-03-03	303.67	304.00	285.80	289.32	289.32	79868852
4	2020-03-02	282.28	301.44	277.72	298.81	298.81	85349339

Y esto? que onda?

```
data.iloc[2:4]
```

	timestamp	open	high	low	close	adjusted_close	volume
2	2020-03-04	296.44	303.4	293.13	302.74	302.74	54794568
3	2020-03-03	303.67	304.0	285.80	289.32	289.32	79868852

Como que nos voló un elemento ¿qué significa esto?

Si lo pensamos un poco el método loc llama a los nombres, es decir llama a todos los elementos que se llaman 2,3 o 4, pero el método iloc llama a los índices, y ahí aplica un criterio numérico que es de **desde inclusive a hasta no inclusive**

Y como sería algo así con iloc si tenemos como índice una fecha?

```
dataIndiceFecha.loc["2020-03-04":"2020-03-02"]
```

	open	high	low	close	adjusted_close	volume
timestamp						
2020-03-04	296.44	303.40	293.13	302.74	302.74	54794568
2020-03-03	303.67	304.00	285.80	289.32	289.32	79868852
2020-03-02	282.28	301.44	277.72	298.81	298.81	85349339

Es muy interesante que no importa si encuentra exactamente los valores de los extremos en un índice de fecha

```
dataIndiceFecha.loc["2020-03-04":"2020-03-01"]
```

	open	high	low	close	adjusted_close	volume
timestamp						
2020-03-04	296.44	303.40	293.13	302.74	302.74	54794568
2020-03-03	303.67	304.00	285.80	289.32	289.32	79868852
2020-03-02	282.28	301.44	277.72	298.81	298.81	85349339

Como pueden ver esto se empieza a poner interesante porque me permite hacer un slice por rangos de fechas de forma muy cómoda

```
dataIndiceFecha.loc["2020-03-01":"2020-02-20"]
```

	open	high	low	close	adjusted_close	volume
timestamp						
2020-02-28	257.26	278.41	256.37	273.36	273.36	106721230
2020-02-27	281.10	286.00	272.96	273.52	273.52	80151381
2020-02-26	286.53	297.88	286.50	292.65	292.65	49678431
2020-02-25	300.95	302.53	286.13	288.08	288.08	57668364
2020-02-24	297.26	304.18	289.23	298.18	298.18	55548828
2020-02-21	318.62	320.45	310.50	313.05	313.05	32426415
2020-02-20	322.63	324.65	318.21	320.30	320.30	25141489

Ahora vamos a seleccionar todo

```
data.iloc[:]
```

	timestamp	open	high	low	close	adjusted_close	volume
0	2020-03-06	282.00	290.82	281.23	289.03	289.0300	56544246
1	2020-03-05	295.52	299.55	291.41	292.92	292.9200	46893219
2	2020-03-04	296.44	303.40	293.13	302.74	302.7400	54794568
3	2020-03-03	303.67	304.00	285.80	289.32	289.3200	79868852
...
5029	2000-03-09	120.87	125.00	118.25	122.25	3.7894	2470700
5030	2000-03-08	122.87	123.94	118.56	122.00	3.7816	2421700
5031	2000-03-07	126.44	127.44	121.12	122.87	3.8086	2437600
5032	2000-03-06	126.00	129.13	125.00	125.69	3.8960	1880000

5033 rows × 7 columns

Selección de columnas

Para seleccionar columnas como dijimos, luego de la selección de filas debemos poner una coma e indicar las columnas deseadas

Veamos un ejemplo en el que seleccionaremos para todos los elementos, solo las columnas, timestamp y close

```
data.loc[:,['timestamp','close']]
```

	timestamp	close
0	2020-03-06	289.03
1	2020-03-05	292.92
2	2020-03-04	302.74
3	2020-03-03	289.32
...
5029	2000-03-09	122.25
5030	2000-03-08	122.00
5031	2000-03-07	122.87
5032	2000-03-06	125.69

5033 rows × 2 columns

Accedemos a las primeras 3 filas de determinadas columnas

```
data.loc[0:2,['timestamp','close','volume']]
```

	timestamp	close	volume
0	2020-03-06	289.03	56544246
1	2020-03-05	292.92	46893219
2	2020-03-04	302.74	54794568

Vemos el mismo ejemplo, pero llamando a las columnas por su número de columna

```
data.iloc[0:2,[0,4,6]]
```

	timestamp	close	volume
0	2020-03-06	289.03	56544246
1	2020-03-05	292.92	46893219

Accedemos a las ultimas 3 filas de determinadas columnas

```
data.tail(3).loc[:,['timestamp','close','volume']]
```

	timestamp	close	volume
5030	2000-03-08	122.00	2421700
5031	2000-03-07	122.87	2437600
5032	2000-03-06	125.69	1880000

Accedemos a las ultimas 3 filas de determinadas columnas ordenado al revés

```
data.tail(3).loc[::-1,['timestamp','close','volume']]
```

	timestamp	close	volume
5032	2000-03-06	125.69	1880000
5031	2000-03-07	122.87	2437600
5030	2000-03-08	122.00	2421700

Accedemos a los primeros 20 valores pero saltando de a 3

```
data[0:20:3]
```

	timestamp	open	high	low	close	adjusted_close	volume
0	2020-03-06	282.00	290.82	281.23	289.03	289.03	56544246
3	2020-03-03	303.67	304.00	285.80	289.32	289.32	79868852
6	2020-02-27	281.10	286.00	272.96	273.52	273.52	80151381
9	2020-02-24	297.26	304.18	289.23	298.18	298.18	55548828
12	2020-02-19	320.00	324.57	320.00	323.62	323.62	23495991
15	2020-02-13	324.19	326.22	323.35	324.87	324.87	23686892
18	2020-02-10	314.18	321.55	313.85	321.55	321.55	27337215

Damos vuelta la tabla

```
data = pd.read_excel('AAPL.xlsx', sheet_name='Hoja1')
data = data.loc[::-1]
data
```

	timestamp	open	high	low	close	adjusted_close	volume
5032	2000-03-06	126.00	129.13	125.00	125.69	3.8960	1880000
5031	2000-03-07	126.44	127.44	121.12	122.87	3.8086	2437600
5030	2000-03-08	122.87	123.94	118.56	122.00	3.7816	2421700
5029	2000-03-09	120.87	125.00	118.25	122.25	3.7894	2470700
...
3	2020-03-03	303.67	304.00	285.80	289.32	289.3200	79868852
2	2020-03-04	296.44	303.40	293.13	302.74	302.7400	54794568
1	2020-03-05	295.52	299.55	291.41	292.92	292.9200	46893219
0	2020-03-06	282.00	290.82	281.23	289.03	289.0300	56544246

5033 rows x 7 columns

Como vemos nos queda el indice al revés porque dio vuelta la tabla
 Para resetear el indice usamos el metodo: `reset_index(drop=True)`

Si no usamos la opción drop=False nos deja el índice viejo como una columna llamada index

```
data = pd.read_excel('AAPL.xlsx', sheet_name='Hoja1')
data = data.loc[::-1]
data = data.reset_index(drop=True)
data.head()
```

	timestamp	open	high	low	close	adjusted_close	volume
0	2000-03-06	126.00	129.13	125.00	125.69	3.8960	1880000
1	2000-03-07	126.44	127.44	121.12	122.87	3.8086	2437600
2	2000-03-08	122.87	123.94	118.56	122.00	3.7816	2421700
3	2000-03-09	120.87	125.00	118.25	122.25	3.7894	2470700
4	2000-03-10	121.69	127.94	121.00	125.75	3.8979	2219700

```
data = pd.read_excel('AAPL.xlsx', sheet_name='Hoja1')
data = data.loc[::-1]
data = data.reset_index(drop=False)
data.head()
```

	index	timestamp	open	high	low	close	adjusted_close	volume
0	5032	2000-03-06	126.00	129.13	125.00	125.69	3.8960	1880000
1	5031	2000-03-07	126.44	127.44	121.12	122.87	3.8086	2437600
2	5030	2000-03-08	122.87	123.94	118.56	122.00	3.7816	2421700
3	5029	2000-03-09	120.87	125.00	118.25	122.25	3.7894	2470700
4	5028	2000-03-10	121.69	127.94	121.00	125.75	3.8979	2219700

Filtrado

Selección de un rango de fechas

```
data = pd.read_excel('AAPL.xlsx')

#para que el indice de la tabla sea la fecha
data.set_index('timestamp', inplace=True)

#para que me deje la tabla ordenada por fecha ascendente
data = data[::-1]

data.loc["2000-03-26":"2000-04-02"]
```

	open	high	low	close	adjusted_close	volume
timestamp						
2000-03-27	137.63	144.75	136.88	139.56	4.3259	2492700
2000-03-28	137.25	142.00	137.13	139.13	4.3126	1812200
2000-03-29	139.38	139.44	133.83	135.94	4.2137	2141400
2000-03-30	133.56	137.69	125.44	125.75	3.8979	3700000
2000-03-31	127.44	137.25	126.00	135.81	4.2097	3612800

Selección por filtros condicionales

Si en lugar de buscar una posición concreta (que la busca en el índice, queremos buscar una coincidencia pero en otra columna, no tenemos más que solicitar la condición buscada, tan sencillo como eso)

```
# el .loc no es necesario Lo pongo aca para mostrarlo
filtro = data.loc[data['volume']>50000000]
filtro
```

	open	high	low	close	adjusted_close	volume
timestamp						
2000-09-29	28.19	29.00	25.37	25.75	1.5963	132529300
2005-01-13	73.65	74.42	69.73	69.80	4.3272	56512800
2005-03-03	44.34	44.41	41.22	41.79	5.1815	50416200
2005-04-14	38.81	39.98	36.84	37.26	4.6198	98328300
...
2020-03-02	282.28	301.44	277.72	298.81	298.8100	85349339
2020-03-03	303.67	304.00	285.80	289.32	289.3200	79868852
2020-03-04	296.44	303.40	293.13	302.74	302.7400	54794568
2020-03-06	282.00	290.82	281.23	289.03	289.0300	56544246

401 rows × 6 columns

Como pueden ver me devuelve la tabla (cortada para optimizar el proceso de output) y me devuelve la cantidad de filas y columnas de mi filtro

Filtros Avanzados

Supongamos que queremos filtrar unos datos combinando filtros de diferentes columnas o diferentes condiciones sobre valores de la misma columna (rangos por ejemplo), bueno, para eso lo primero que podríamos intentar sería ir concatenando condiciones una tras otra. Veamos un ejemplo, aquí listaremos las filas con volumen mayor a 6 millones y que además hayan tenido un precio de cierre menor a usd 100

```

filtro = data[data['volume']>50000000]
filtro = filtro[filtro['close']<100]
filtro

```

	open	high	low	close	adjusted_close	volume
timestamp						
2000-09-29	28.19	29.000	25.37	25.75	1.5963	132529300
2005-01-13	73.65	74.420	69.73	69.80	4.3272	56512800
2005-03-03	44.34	44.410	41.22	41.79	5.1815	50416200
2005-04-14	38.81	39.980	36.84	37.26	4.6198	98328300
...
2016-05-16	92.39	94.390	91.65	93.88	88.5167	61259756
2016-06-17	96.62	96.650	95.30	95.33	89.8839	61008219
2016-06-24	92.91	94.655	92.65	93.40	88.0642	75311356
2016-07-26	96.82	97.970	96.42	96.67	91.1473	56239822

106 rows × 6 columns

Si bien es válida como solución es un poco desprolija ya que tuvimos que filtrar dos veces, la primera vez filtramos "data" y pusimos el resultado en la variable "filtro" y la segunda vez filtramos la variable "filtro"

También se pudo haber escrito el mismo código de la siguiente forma:

```

filtro =
data[data['volume']>50000000].loc[data['close']<100] filtro

```

	open	high	low	close	adjusted_close	volume
timestamp						
2000-09-29	28.19	29.000	25.37	25.75	1.5963	132529300
2005-01-13	73.65	74.420	69.73	69.80	4.3272	56512800
2005-03-03	44.34	44.410	41.22	41.79	5.1815	50416200
2005-04-14	38.81	39.980	36.84	37.26	4.6198	98328300
...
2016-05-16	92.39	94.390	91.65	93.88	88.5167	61259756
2016-06-17	96.62	96.650	95.30	95.33	89.8839	61008219
2016-06-24	92.91	94.655	92.65	93.40	88.0642	75311356
2016-07-26	96.82	97.970	96.42	96.67	91.1473	56239822

106 rows × 6 columns

¿Otra forma más prolífica de hacerlo?

Claro, siempre hay muchas formas diferentes de hacer lo mismo en programación, insisto, nunca piensen en términos de cuál es mejor o peor, no existe eso, van a haber ocasiones en las que les sirva más una que otra, o directamente ocasiones en donde por alguna razón una forma no se puede usar y hay que buscar otra, en fin, acá parece todo demasiado obvio porque son siempre ejemplos muy sencillos que uso, pero en la práctica siempre aparecen dificultades, siempre es mejor saber varias formas de resolver el mismo problema.

Otra forma más prolífica de hacer lo mismo sería definir los filtros por separado en un paso y en un segundo paso "combinarlos" vemos:

```
#estos son los valores que cumplen la primera condición
filtro1 = data[data['volume'] > 50000000]

#estos son los valores que cumplen la segunda condición
filtro2 = data[data['close'] < 100]

#merge es un método que combina ambos frames, Los "mergea" :)
#how="inner" es la intersección de ambos
#how="outer" es la unión de ambos

filtro = filtro1.merge(filtro2, how="inner")
filtro
```

	open	high	low	close	adjusted_close	volume
0	28.19	29.000	25.37	25.75	1.5963	132529300
1	73.65	74.420	69.73	69.80	4.3272	56512800
2	44.34	44.410	41.22	41.79	5.1815	50416200
3	38.81	39.980	36.84	37.26	4.6198	98328300
...
102	92.39	94.390	91.65	93.88	88.5167	61259756
103	96.62	96.650	95.30	95.33	89.8839	61008219
104	92.91	94.655	92.65	93.40	88.0642	75311356
105	96.82	97.970	96.42	96.67	91.1473	56239822

106 rows × 6 columns

Pero supongamos que quisiera que en lugar de una conjunción de ambas condiciones sea ^o una o la otra o ambas

```
#estos son los valores que cumplen la primera condición
filtro1 = data[data['volume']>50000000]

#estos son los valores que cumplen la segunda condición
filtro2 = data[data['close']<100]

filtro = filtro1.merge(filtro2, how="outer")
filtro
```

	open	high	low	close	adjusted_close	volume
0	28.19	29.00	25.370	25.75	1.5963	132529300
1	73.65	74.42	69.730	69.80	4.3272	56512800
2	44.34	44.41	41.220	41.79	5.1815	50416200
3	38.81	39.98	36.840	37.26	4.6198	98328300
...
2301	100.00	100.46	99.735	99.96	94.2494	26275968
2302	99.83	101.00	99.130	99.43	93.7497	32702028
2303	99.26	99.30	98.310	98.66	93.0237	28313669
2304	98.25	98.84	96.920	97.34	91.7791	40382921

2305 rows × 6 columns

A mi mucho esto no me gusta, me parece demasiado vueltoso definir los filtros por separado, pero hay ocasiones en donde esos filtros se van a ir dando dinámicamente en una rutina, y si además de eso tenemos 5 o 10 condiciones, la verdad que ahí empieza a cobrar sentido definir los filtros por separado y luego manejar la combinación de estos

Pero bueno, eso es parte del criterio de cada uno a la hora de programar. Siempre es recomendable que el código sea entendible por un tercero, a modo de chiste en el rubro decimos que hay que codear pensando que el que va a venir atrás nuestro, a seguir nuestro trabajo es un asesino serial, así que no habría que hacerlo renegar mucho para que entienda el código

Filtros Avanzados con OPERADORES lógicos

Ahora veremos la forma de combinar condiciones usando operadores dentro de la condición

Los dos operadores lógicos básicos son obviamente AND y OR:

- Condición AND = &
- Condición OR = |

```
#Método definiendo filtros por separado y "mergeando"
filtro1 = data[data['volume']>50000000]
filtro2 = data[data['close']<100]
filtro = filtro1.merge(filtro2, how="outer")

# Usando operadores Lógicos
filtro = data[(data['volume']>50000000) | (data['close']<100)]

filtro
```

	open	high	low	close	adjusted_close	volume
timestamp						
2000-05-10	104.06	105.00	98.75	99.31	3.0783	4777600
2000-05-19	99.25	99.25	93.37	94.00	2.9137	6613100
2000-05-22	93.75	93.75	86.00	89.94	2.7879	6745600
2000-05-23	90.50	93.37	85.62	85.81	2.6599	4621300
...
2020-03-02	282.28	301.44	277.72	298.81	298.8100	85349339
2020-03-03	303.67	304.00	285.80	289.32	289.3200	79868852
2020-03-04	296.44	303.40	293.13	302.74	302.7400	54794568
2020-03-06	282.00	290.82	281.23	289.03	289.0300	56544246

2305 rows × 6 columns

```
#Método definiendo filtros por separado y "mergeando"
filtro1 = data[data['volume']>50000000]
filtro2 = data[data['close']<100]
filtro = filtro1.merge(filtro2, how="inner")

# Usando operadores Lógicos
filtro = data[(data['volume']>50000000) & (data['close']<100)]

filtro
```

	open	high	low	close	adjusted_close	volume
timestamp						
2000-09-29	28.19	29.000	25.37	25.75	1.5963	132529300
2005-01-13	73.65	74.420	69.73	69.80	4.3272	56512800
2005-03-03	44.34	44.410	41.22	41.79	5.1815	50416200
2005-04-14	38.81	39.980	36.84	37.26	4.6198	98328300
...
2016-05-16	92.39	94.390	91.65	93.88	88.5167	61259756
2016-06-17	96.62	96.650	95.30	95.33	89.8839	61008219
2016-06-24	92.91	94.655	92.65	93.40	88.0642	75311356
2016-07-26	96.82	97.970	96.42	96.67	91.1473	56239822

106 rows × 6 columns

Ordenamiento de Tabla

Empecemos ordenando por una sola columna, por ejemplo por fecha ascendente

```
data = pd.read_excel('AAPL.xlsx')
ordenado = data.sort_values(by='adjusted_close', ascending=True)
ordenado.head(8)
```

	timestamp	open	high	low	close	adjusted_close	volume
4250	2003-04-17	13.20	13.25	12.72	13.12	0.8134	11004600
4249	2003-04-21	13.13	13.19	12.98	13.14	0.8146	2720000
4254	2003-04-11	14.05	14.44	12.93	13.20	0.8183	24869800
4251	2003-04-16	12.99	13.67	12.92	13.24	0.8208	18146000
4245	2003-04-25	13.46	13.58	13.23	13.35	0.8276	3666400
4252	2003-04-15	13.59	13.60	13.30	13.39	0.8301	5428000
4246	2003-04-24	13.52	13.61	13.00	13.44	0.8332	5805500
4248	2003-04-22	13.18	13.62	13.09	13.51	0.8375	5367300

Ahora vamos a usar doble criterio de ordenamiento, es decir, ordenamos por precio ajustado, y si tenemos dos valores con el mismo precio ajustado ordenamos por volumen ascendente

```
# redondeamos en solo 2 decimales para poder apreciar el resultado
data.adjusted_close = round(data.adjusted_close,2)

ordenado = data.sort_values(by=['adjusted_close',
'volume'],ascending=[True, True])
ordenado.head(8)
```

	timestamp	open	high	low	close	adjusted_close	volume
4249	2003-04-21	13.13	13.19	12.98	13.14	0.81	2720000
4250	2003-04-17	13.20	13.25	12.72	13.12	0.81	11004600
4251	2003-04-16	12.99	13.67	12.92	13.24	0.82	18146000
4254	2003-04-11	14.05	14.44	12.93	13.20	0.82	24869800
4245	2003-04-25	13.46	13.58	13.23	13.35	0.83	3666400
4252	2003-04-15	13.59	13.60	13.30	13.39	0.83	5428000
4246	2003-04-24	13.52	13.61	13.00	13.44	0.83	5805500
4247	2003-04-23	13.53	13.63	13.37	13.58	0.84	3744300

Tratamiento de los elementos del DataFrame

Básicamente un data frame es, digamos, una tabla bidimensional, por lo tanto puedo desagregarlo en "filas" o "columnas" unidimensionales y a cada elemento de estos convertirlo en una lista, tupla o diccionario, veamos ejemplos

Extraigamos la columna del volumen:

```
data = pd.read_excel('AAPL.xlsx', sheet_name='Hoja1')

#También es válido: vol = data['volume']
vol = data.volume
vol
```

```
0      56544246
1      46893219
2      54794568
3      79868852
485349339
...
50282219700
50292470700
50302421700
50312437600
50321880000
Name: volume, Length: 5033, dtype: int64
```

Ahora bien, ¿qué tipo de dato es lo que obtengo?
Veámoslo:

```
type(vol)  
pandas.core.series.Series
```

Es un objeto, una serie de un DataFrame de pandas, pero supongamos que quiero una lista nomás

Como vimos en la unidad de listas, para convertir algo en una lista tenemos la función `list()`

Entonces hagamos directamente así:

```
data = pd.read_excel('AAPL.xlsx', sheet_name='Hoja1')  
vol = list(data.volume)  
type(vol)  
  
list
```

Y ustedes dirán ¿para qué quiero tener la columna de volúmenes suelta en una lista? jaja, bueno, por ahora son solo "herramientas sueltas" que andan por ahí y les recomiendo fuertemente que les presten atención porque les aseguro que las van a necesitar.

No se preocupen por recordar exactamente como se usan todas estas herramientas, nadie lo recuerda, pero es importante que sepan que existen las herramientas y que por algún rincón perdido de su memoria lo dejen asentado.

El día que tengan que resolver un problema que necesite de este tipo de herramientas lo importante es que solo recuerden eso, como para tener en cuenta que cuentan con la herramienta, después es re común que no sepan como era que se usaba exactamente, pero como les decía lo importante es lo otro

Bueno, ahora que tengo una columna guardada como una lista, claramente podemos usar todas las herramientas que ya aprendimos sobre listas, por ejemplo, si quiera saber el volumen promedio, solo tengo que usar las funciones `sum()` y `len()`

Veamos el ejemplo

```
data = pd.read_excel('AAPL.xlsx', sheet_name='Hoja1')  
vol = list(data.volume)  
volMedio = sum(vol)/len(vol)  
volMedio
```

23694203.887343533

También puedo armar una lista con cualquier fila, incluso con la fila de las columnas del dataframe

```
data = pd.read_excel('AAPL.xlsx', sheet_name='Hoja1')
columnas = list(data.columns)
columnas
['timestamp', 'open', 'high', 'low', 'close', 'adjusted_close', 'volume']

data = pd.read_excel('AAPL.xlsx', sheet_name='Hoja1')
primeraFila = list(data.loc[0])
primeraFila
[Timestamp('2020-03-06 00:00:00'), 282.0, 290.82, 281.23, 289.03, 289.03, 56544246]
```

Armado de diccionario partiendo de una fila de un DataFrame

```
ordenadoPorVolumen = data.sort_values(by='volume', ascending=False)
mayorVolumen = ordenadoPorVolumen.head(3)
mayorVolumen
```

	timestamp	open	high	low	close	adjusted_close	volume
1382	2014-09-09	99.080	103.08	96.14	97.99	89.4259	189560600
1141	2015-08-24	94.870	108.80	92.00	103.12	95.7091	162206292
1285	2015-01-28	117.625	118.12	115.31	115.31	105.6872	146477063

```
volDict = mayorVolumen.to_dict(orient="list")
volDict
```

```
{'timestamp': [Timestamp('2014-09-09 00:00:00'),
Timestamp('2015-08-24 00:00:00'),
Timestamp('2015-01-28 00:00:00')],
'open': [99.08, 94.87, 117.625],
'high': [103.08, 108.8, 118.12],
'low': [96.14, 92.0, 115.31],
'close': [97.99, 103.12, 115.31],
'adjusted_close': [89.4259, 95.7091, 105.6872],
'volume': [189560600, 162206292, 146477063]}
```

También puedo armar una lista con cualquier fila, incluso con la fila de las columnas del dataframe

```
data = pd.read_excel('AAPL.xlsx', sheet_name='Hoja1')
columnas = list(data.columns)
columnas

['timestamp', 'open', 'high', 'low', 'close', 'adjusted_close', 'volume']
```

```
data = pd.read_excel('AAPL.xlsx', sheet_name='Hoja1')
primeraFila = list(data.loc[0])
primeraFila
```

```
[Timestamp('2020-03-06 00:00:00'), 282.0, 290.82, 281.23, 289.03, 289.03, 56544246]
```

Armado de diccionario partiendo de una fila de un DataFrame

```
ordenadoPorVolumen = data.sort_values(by='volume', ascending=False)
mayorVolumen = ordenadoPorVolumen.head(3)
mayorVolumen
```

	timestamp	open	high	low	close	adjusted_close	volume
1382	2014-09-09	99.080	103.08	96.14	97.99	89.4259	189560600
1141	2015-08-24	94.870	108.80	92.00	103.12	95.7091	162206292
1285	2015-01-28	117.625	118.12	115.31	115.31	105.6872	146477063

```
volDict = mayorVolumen.to_dict(orient="list")
volDict
```

```
{'timestamp': [Timestamp('2014-09-09 00:00:00'),
   Timestamp('2015-08-24 00:00:00'),
   Timestamp('2015-01-28 00:00:00')],
 'open': [99.08, 94.87, 117.625],
 'high': [103.08, 108.8, 118.12],
 'low': [96.14, 92.0, 115.31],
 'close': [97.99, 103.12, 115.31],
 'adjusted_close': [89.4259, 95.7091, 105.6872],
 'volume': [189560600, 162206292, 146477063]}
```

```
# Otras orientaciones posibles son: series, split, records, index
# Esto puede ser util cuando veamos JSON, o trabajemos con esos objetos
```

```
volDict = mayorVolumen.to_dict(orient="dict")
volDict
```

```
{'timestamp': {1382: Timestamp('2014-09-09 00:00:00'),
 1141: Timestamp('2015-08-24 00:00:00'),
 1285: Timestamp('2015-01-28 00:00:00')},
'open': {1382: 99.08, 1141: 94.87, 1285: 117.625},
'high': {1382: 103.08, 1141: 108.8, 1285: 118.12},
'low': {1382: 96.14, 1141: 92.0, 1285: 115.31},
'close': {1382: 97.99, 1141: 103.12, 1285: 115.31},
'adjusted_close': {1382: 89.4259, 1141: 95.7091, 1285: 105.6872},
'velume': {1382: 189560600, 1141: 162206292, 1285: 146477063}}
```

Como dijimos, por ahora no vamos a utilizar las estructuras de los diccionarios, esto nos va a resultar muy útil sobre todo para hacer screeners, imaginén que obtenemos todas las cotizaciones del mercado de un país vía WEB y queremos guardar los tickers y sus precios actuales, la mejor manera de tenerlo disponible en memoria es con un diccionario clave->valor, en donde la clave sea el ticker y el valor su precio actual, si los datos los tenemos en un DataFrame que tiene muchas columnas, seguramente necesite utilizar un método como este

Una vez que tengo el diccionario, en nuestro caso "volDict" puedo acceder a cualquiera de sus claves y ver que contienen, en este caso como el diccionario viene de una tabla de 3 elementos, cada clave contendrá en el valor una lista con esos tres elementos Veámoslo más sencillo con el ejemplo

```
volDict['timestamp']
```

```
[Timestamp('2014-09-09 00:00:00'),
Timestamp('2015-08-24 00:00:00'),
Timestamp('2015-01-28 00:00:00')]
```

```
volDict['volume']
```

```
[189560600, 162206292, 146477063]
```

Ejercicios

Tomando los datos de las cotizaciones del archivo AAPL.xlsx:

- 1- Mostrar los datos correspondientes al 15 de mayo de 2008
- 2- Mostrar los 3 datos posteriores al 15 de mayo de 2008
- 3- Contar la cantidad de ruedas entre el 15 de mayo de 2008 y el 20 de agosto de 2011
- 4- Contar la cantidad de ruedas entre el 15 de mayo de 2008 y el 20 de agosto de 2011, siempre que el precio de cierre haya sido mayor al precio de apertura
- 5- Mostrar las ruedas entre el 15 de mayo de 2008 y el 20 de agosto de 2011, en las que haya subido en algún momento del día más del 7% del precio de apertura. Eliminar las columnas de volumen y cierre ajustado
- 6- Mostrar las 5 ruedas de mayor upside intradiario respecto del precio de apertura de 2018
- 7- Mostrar las 5 ruedas de mayor downside intradiario respecto del precio de apertura, para ruedas del año 2019
- 8- Mostrar las 10 ruedas de mayor upside intradiario en todo el historial y guardar en un Excel solo las columnas de la fecha y el upside intradiario % de esas 10 ruedas
- 9- Mostrar el listado de ruedas en los que el volumen en nominales haya sido más de 10 veces superior al volumen medio histórico (solo usando las herramientas vistas hasta acá) Y mostrar el movimiento % intradiario (cierre respecto a apertura)
- 10- Buscar ruedas de más de 150 millones o menos de 1 millón de nominales, mostrar la fecha y el volumen

Respuestas

```
#-----#
# Rta Ejercicio 1 #
#-----#
import pandas as pd
data = pd.read_excel('AAPL.xlsx', sheet_name='Hoja1')
dataIndiceFecha = data.set_index("timestamp")
dataIndiceFecha.loc["2008-05-15"]
```

	open	high	low	close	adjusted_close	volume
timestamp						
2008-05-15	186.78	189.9	184.2	189.73	23.5242	31186000

```
#-----#
# Rta Ejercicio 2 #
#-----#
# Alternativa N°1
data = pd.read_excel('AAPL.xlsx', sheet_name='Hoja1')
dataIndiceFecha = data.set_index("timestamp")
posteriores = dataIndiceFecha.loc["2020-03-06":"2008-05-15"]
posteriores.tail().iloc[1:4]
```

	open	high	low	close	adjusted_close	volume
timestamp						
2008-05-20	181.83	186.16	180.12	185.9000	23.0494	34637500
2008-05-19	187.90	188.69	181.30	183.6000	22.7642	33779300
2008-05-16	190.30	190.30	187.00	187.6201	23.2626	27348900

O bien la otra forma de hacerlo sería:

```
#-----#
# Rta Ejercicio 2 #
#-----#
# Alternativa N°2

data = pd.read_excel('AAPL.xlsx', sheet_name='Hoja1')
data = data.sort_values(by='timestamp', ascending=True)
dataIndiceFecha = data.set_index("timestamp")
posteriores = dataIndiceFecha.loc["2008-05-15":"2020-03-06"]
posteriores.iloc[1:4]
```

	open	high	low	close	adjusted_close	volume
timestamp						
2008-05-16	190.30	190.30	187.00	187.6201	23.2626	27348900
2008-05-19	187.90	188.69	181.30	183.6000	22.7642	33779300
2008-05-20	181.83	186.16	180.12	185.9000	23.0494	34637500

O bien una tercera alternativa completamente válida también:

```
#-----#
# Rta Ejercicio 2 #
#-----#
# Alternativa N°3

data = pd.read_excel('AAPL.xlsx', sheet_name='Hoja1')
posteriores = data.loc[data['timestamp'] > "2008-05-15"]
posteriores.tail(3)
```

	timestamp	open	high	low	close	adjusted_close	volume
2969	2008-05-20	181.83	186.16	180.12	185.9000	23.0494	34637500
2970	2008-05-19	187.90	188.69	181.30	183.6000	22.7642	33779300
2971	2008-05-16	190.30	190.30	187.00	187.6201	23.2626	27348900

```
#-----#
# Rta Ejercicio 3 #
#-----#

import pandas as pd
data = pd.read_excel('AAPL.xlsx', sheet_name='Hoja1')
dataIndiceFecha = data.set_index("timestamp")
rango = dataIndiceFecha.loc["2011-08-20":"2008-05-15"]
len(rango)
```

```
#-----#
# Rta Ejercicio 4 #
#-----#
data = pd.read_excel('AAPL.xlsx', sheet_name='Hoja1')
dataIndiceFecha = data.set_index("timestamp")
rango = dataIndiceFecha.loc["2011-08-20":"2008-05-15"]
rangoFiltro = rango.loc[rango['close']>rango['open']]
len(rangoFiltro)
```

421

```
#-----#
# Rta Ejercicio 5 #
#-----#
data = pd.read_excel('AAPL.xlsx', sheet_name='Hoja1')
data['max_upside'] = round((data['high']/data['open']-1)*100,2)
dataIndiceFecha = data.set_index("timestamp")
rango = dataIndiceFecha.loc["2011-08-20":"2008-05-15"]
rangoFiltro = rango.loc[rango['high']/rango['open']-1 > 0.07] rangoFiltro.drop(["volume","adjusted_close"],axis=1)
```

	open	high	low	close	max_upside
timestamp					
2008-12-03	89.4000	96.23	88.80	95.90	7.64
2008-11-24	85.2100	94.79	84.84	92.95	11.24
2008-11-13	89.8700	96.44	86.02	96.44	7.31
2008-10-29	100.8600	109.54	99.94	104.55	8.61
2008-10-24	90.3300	97.90	90.11	96.38	8.38
2008-10-10	85.7000	100.00	85.00	96.80	16.69
2008-10-08	85.9125	96.33	85.68	89.79	12.13
2008-10-06	91.9600	98.78	87.54	98.14	7.42
2008-09-18	130.5700	140.00	120.68	134.09	7.22
2008-07-22	148.9000	162.76	146.53	162.02	9.31

```
#-----#
# Rta Ejercicio 6 #
#-----#

data = pd.read_excel('AAPL.xlsx', sheet_name='Hoja1')
data['max_upside'] = round((data['high']/data['open']-1)*100,2)
dataIndiceFecha = data.set_index("timestamp")
rango = dataIndiceFecha.loc["2018-12-31":"2018-01-01"] rango
= rango.sort_values(by='max_upside',ascending=False)
rango.drop(["volume","adjusted_close"],axis=1,inplace=True)
rango.head(5)
```

	open	high	low	close	max_upside
timestamp					
2018-12-26	148.30	157.23	146.72	157.17	6.02
2018-02-06	154.83	163.72	154.00	163.03	5.74
2018-04-04	164.88	172.01	164.77	171.61	4.32
2018-08-02	200.58	208.38	200.35	207.39	3.89
2018-10-23	215.83	223.25	214.70	222.73	3.44

```
#-----#
# Rta Ejercicio 7 #
#-----#

data = pd.read_excel('AAPL.xlsx', sheet_name='Hoja1')
data['max_downside'] = round((data['low']/data['open']-1)*100,2)
dataIndiceFecha = data.set_index("timestamp")
rango = dataIndiceFecha.loc["2018-12-31":"2018-01-01"] rango
= rango.sort_values(by='max_downside',ascending=True)
rango.drop(["volume","adjusted_close"],axis=1,inplace=True)
rango.head(5)
```

	open	high	low	close	max_downside
timestamp					
2018-10-29	219.19	219.69	206.09	212.24	-5.98
2018-12-21	156.86	158.16	149.63	150.73	-4.61
2018-02-09	157.07	157.89	150.24	156.41	-4.35
2018-10-10	225.46	226.35	216.05	216.36	-4.17
2018-12-19	166.00	167.45	159.09	160.89	-4.16

```
#-----#
# Rta Ejercicio 8 #
#-----#
data = pd.read_excel('AAPL.xlsx', sheet_name='Hoja1')
data['max_upside'] = round((data['high']/data['open']-1)*100,2)
dataIndiceFecha = data.set_index("timestamp")
data = data.sort_values(by='max_upside', ascending=False)
top10 = data.head(10)
top10.set_index("timestamp", inplace=True)
top10.to_excel("AAPL_ejercicio8.xlsx" , columns=["max_upside"],
index=False) top10[["max_upside"]]
```

	max_upside
timestamp	
2008-10-10	16.69
2001-01-03	15.10
2015-08-24	14.68
2003-05-05	14.29
2001-01-11	13.85
2000-04-17	13.19
2000-06-21	12.75
2008-10-08	12.13
2001-04-18	11.64
2001-03-02	11.63

```
#-----#
# Rta Ejercicio 9 #
#-----#
data = pd.read_excel('AAPL.xlsx', sheet_name='Hoja1')
data['intradiario'] = round((data.close / data.open -1)*100 ,2)
volMedio = sum(list(data.volume))/len(list(data.volume))
altoVolumen = data[data.volume > volMedio*5]
altoVolumen.set_index("timestamp",inplace=True)
altoVolumen[["volume","intradiario"]]
```

	volume	intradiario
timestamp		
2016-01-27	133369674	-2.73
2015-08-24	162206292	8.70
2015-08-21	128275471	-4.23
2015-08-04	124138623	-2.37
2015-04-28	118923970	-2.90
2015-01-28	146477063	-1.97
2014-09-09	189560600	-1.10
2014-09-03	125233100	-4.03
2008-01-23	120463200	2.12
2007-01-09	119617800	7.08
2000-09-29	132529300	-8.66

```
#-----#
# Rta Ejercicio 10 #
#-----#
filtro = data[(data['volume']>150000000) |  

 (data['volume']<1000000)] filtro[["timestamp","volume"]]
```

	timestamp	volume
1141	2015-08-24	162206292
1382	2014-09-09	189560600
4328	2002-12-24	702500
4580	2001-12-24	904100

Funciones básicas de Pandas

pct_change()

Esta función nos permite calcular fácilmente el porcentaje de variación entre filas, debe ser una de las funciones más usadas en python/pandas

Vamos a seguir trabajando con los datos históricos de AAPL del archivo AAPL.xlsx que pueden descargar en <http://bit.ly/39Am64T> (<http://bit.ly/39Am64T>)

En este ejemplo agregamos la variación % de cada rueda respecto al día anterior.

Como los datos en el Excel están desde los más recientes a los más antiguos, vamos a dar vuelta la serie con [::-1] como vimos en la unidad anterior

.. y ¿por qué necesitamos dar vuelta la serie?, bueno en realidad como veremos no es tan necesario, pero en principio la función pct_change lo que hace es calcular la variación porcentual con respecto al valor de fila anterior, por lo tanto, si no damos vuelta primero la serie, lo que va a hacer es calcular la variación respecto al día de mañana de cada rueda

```
import pandas as pd
data = pd.read_excel('AAPL.xlsx')

data = data[::-1]
data['variacion'] = round(data['adjusted_close'].pct_change() * 100,2)
data = data.drop(["high","low","volume"], axis=1).set_index("timestamp")
data.head()
```

	open	close	adjusted_close	variacion
timestamp				
2000-03-06	126.00	125.69	3.8960	NaN
2000-03-07	126.44	122.87	3.8086	-2.24
2000-03-08	122.87	122.00	3.7816	-0.71
2000-03-09	120.87	122.25	3.7894	0.21
2000-03-10	121.69	125.75	3.8979	2.86

Como argumento le podemos pasar el intervalo es decir cada cuantas filas queremos que nos calcule el porcentaje

Por default el intervalo=1

En el siguiente ejemplo calculamos la variación porcentual cada 3 ruedas, ya que es lo que le pasamos en el argumento de la función

Como se darán cuenta, los primeros 3 valores figuran como "NaN" y esto es porque hasta que no llega a la rueda número 4 de la serie no tiene una "3ra rueda anterior" para comparar, más adelante, veremos como tratar esos "valores vacíos o no válidos"

```
data = pd.read_excel('AAPL.xlsx')
data = data[::-1]

data['variacion'] = round(data['adjusted_close'].pct_change(3) * 100,2)
data = data.drop(["high","low","volume"], axis=1).set_index("timestamp")
data.head()
```

	open	close	adjusted_close	variacion
timestamp				
2000-03-06	126.00	125.69	3.8960	NaN
2000-03-07	126.44	122.87	3.8086	NaN
2000-03-08	122.87	122.00	3.7816	NaN
2000-03-09	120.87	122.25	3.7894	-2.74
2000-03-10	121.69	125.75	3.8979	2.34

Una utilidad muy piola que le podemos dar al argumento de cada cuantas filas aplicamos la función, es utilizar el -1 que lo que hace obviamente es en lugar de hacerlo con respecto a la fila anterior lo hace respecto a la fila siguiente, claramente, con esta función podemos evitarnos tener que dar vuelta la tabla para poder usarla y que me calcule lo que efectivamente quería

```
data = pd.read_excel('AAPL.xlsx')
data['variacion'] = round(data['adjusted_close'].pct_change(-1) * 100,2)
data = data.drop(["high","low","volume"], axis=1).set_index("timestamp")
data
```

	open	close	adjusted_close	variacion
timestamp				
2020-03-06	282.00	289.03	289.0300	-1.33
2020-03-05	295.52	292.92	292.9200	-3.24
2020-03-04	296.44	302.74	302.7400	4.64
2020-03-03	303.67	289.32	289.3200	-3.18
2020-03-02	282.28	298.81	298.8100	9.31
...
2000-03-10	121.69	125.75	3.8979	2.86
2000-03-09	120.87	122.25	3.7894	0.21
2000-03-08	122.87	122.00	3.7816	-0.71
2000-03-07	126.44	122.87	3.8086	-2.24
2000-03-06	126.00	125.69	3.8960	NaN

5033 rows × 4 columns

diff()

La función `diff()` es muy parecida a `pct_change()` solo que en lugar de calcular una variación porcentual lo que calcula es una diferencia absoluta, es decir simplemente una resta entre los valores consecutivos de las filas

En el siguiente ejemplo calculamos la variación en dólares del precio de AAPL entre el cierre de dos ruedas, para esto no podemos usar los cierres ajustados como lo hacíamos con las variaciones porcentuales ya que de hacer eso no obtendríamos la variación en dólares sino en dólares ajustados por splits y dividendos, y claramente no relevaría para nada las variaciones absolutas reales

```
data = pd.read_excel('AAPL.xlsx')
data = data[::-1]
data['variacion_precio'] = data['close'].diff()
data = data.drop(['high','low','volume'], axis=1).set_index("timestamp")
```

	open	close	adjusted_close	variacion_precio
timestamp				
2000-03-06	126.00	125.69	3.8960	NaN
2000-03-07	126.44	122.87	3.8086	-2.82
2000-03-08	122.87	122.00	3.7816	-0.87
2000-03-09	120.87	122.25	3.7894	0.25
2000-03-10	121.69	125.75	3.8979	3.50

Acepta el mismo argumento que la función `pct_change`, es decir que le puedo pasar cada cuantas filas quiero saber la diferencia, y claramente puedo usar el mismo truco de poner valores negativos, que en lugar de "x" celdas antes va a tomar como referencia "x" celdas después de la fila dada

```
data = pd.read_excel('AAPL.xlsx')
data['variacion_precio'] = data['close'].diff(-1)
data = data.drop(['high','low','volume'], axis=1).set_index("timestamp")
data.tail()
```

	open	close	adjusted_close	variacion_precio
timestamp				
2000-03-10	121.69	125.75	3.8979	3.50
2000-03-09	120.87	122.25	3.7894	0.25
2000-03-08	122.87	122.00	3.7816	-0.87
2000-03-07	126.44	122.87	3.8086	-2.82
2000-03-06	126.00	125.69	3.8960	NaN

eval()

La función eval() evalúa una condición dada como string y devuelve True o False en función del resultado de la evaluación. Podemos asignar este output a una nueva columna de una tabla

Esta función va a ser útil para sistemas de trading, ya que un sistema de trading completo va a requerir muchas condiciones, y la manera más prolífica de armar la tabla de decisión que gatille cada acción va a ser teniendo guardado en un frame una columna para cada condición en donde el valor de la columna sea True o False según el método

Veamos un ejemplo para tener en una columna la condición para cada rueda de si el precio de cierre es mayor al de apertura, lo que me generaría una vela verde en un chart de velas

```
data = pd.read_excel('AAPL.xlsx')
data['vela_verde'] = data.eval(' close >= open ')
data = data.drop(["high","low","volume"], axis=1).set_index("timestamp")
data.head()
```

	open	close	adjusted_close	vela_verde
timestamp				
2020-03-06	282.00	289.03	289.03	True
2020-03-05	295.52	292.92	292.92	False
2020-03-04	296.44	302.74	302.74	True
2020-03-03	303.67	289.32	289.32	False
2020-03-02	282.28	298.81	298.81	True

Cabe destacar que la función eval en realidad puede remplazarse simplemente por la evaluación que retorna un True O False

En el siguiente ejemplo vemos como sería el mismo ejemplo anterior
Pero la ventaja de usar la función EVAL es que podemos pasarle en un string la condición y ese string puede más adelante venir de otro lado, ya lo veremos

```
data = pd.read_excel('AAPL.xlsx')
data['vela_verde'] = data['close'] >= data['open']
data = data.drop(["high","low","volume"], axis=1).set_index("timestamp")
data.head(4)
```

	open	close	adjusted_close	vela_verde
timestamp				
2020-03-06	282.00	289.03	289.03	True
2020-03-05	295.52	292.92	292.92	False
2020-03-04	296.44	302.74	302.74	True
2020-03-03	303.67	289.32	289.32	False

También podemos realizar alguna cuenta, por ejemplo, supongamos que quisiera tener una columna que indique TRUE solo las ruedas cuyo cierre fue más de un 3% mayor a la apertura, en ese caso deberíamos preguntar: `close >= open * 1.03`

```
data = pd.read_excel('AAPL.xlsx')
data['condicion'] = data.eval(' close >= open * 1.03')
data = data.drop(['high','low','volume'], axis=1).set_index("timestamp")
```

timestamp	open	close	adjusted_close	condicion
2020-03-06	282.00	289.03	289.03	False
2020-03-05	295.52	292.92	292.92	False
2020-03-04	296.44	302.74	302.74	False
2020-03-03	303.67	289.32	289.32	False
2020-03-02	282.28	298.81	298.81	True

Vamos a calcular en una columna aparte el movimiento intradiario para verificar que es correcto lo que hicimos

```
data = pd.read_excel('AAPL.xlsx')
data['mov_intra'] = round((data.close / data.open - 1)*100, 2)
data['condicion'] = data.eval(' close >= open * 1.03')
data = data.drop(['high','low','volume'], axis=1).set_index("timestamp")
```

timestamp	open	close	adjusted_close	mov_intra	condicion
2020-03-06	282.00	289.03	289.03	2.49	False
2020-03-05	295.52	292.92	292.92	-0.88	False
2020-03-04	296.44	302.74	302.74	2.13	False
2020-03-03	303.67	289.32	289.32	-4.73	False
2020-03-02	282.28	298.81	298.81	5.86	True

También podemos agregar al string de la comparación un operador lógico: y sus respectivos caracteres abreviados: AND (&) , OR (|)

Pongamos entonces la columna con la condición cuando el movimiento intradiario es mayor al 3% o menos al -3%

```
data = pd.read_excel('AAPL.xlsx')
data['mov_intra'] = round((data.close / data.open - 1)*100, 2)
data['condicion'] = data.eval(' close >= open * 1.03 or close <= open * 0.97')
data = data.drop(["high","low","volume"], axis=1).set_index("timestamp")
data.head()
```

	open	close	adjusted_close	mov_intra	condicion
timestamp					
2020-03-06	282.00	289.03	289.03	2.49	False
2020-03-05	295.52	292.92	292.92	-0.88	False
2020-03-04	296.44	302.74	302.74	2.13	False
2020-03-03	303.67	289.32	289.32	-4.73	True
2020-03-02	282.28	298.81	298.81	5.86	True

O bien también podríamos haber solucionado el mismo problema usando la función **ABS**

```
data = pd.read_excel('AAPL.xlsx')
data['mov_intra'] = round((data.close / data.open - 1)*100, 2)
data['condicion'] = data.eval(' abs (close/open - 1) > 0.03')
data = data.drop(["high","low","volume"], axis=1).set_index("timestamp")
data.head()
```

	open	close	adjusted_close	mov_intra	condicion
timestamp					
2020-03-06	282.00	289.03	289.03	2.49	False
2020-03-05	295.52	292.92	292.92	-0.88	False
2020-03-04	296.44	302.74	302.74	2.13	False
2020-03-03	303.67	289.32	289.32	-4.73	True
2020-03-02	282.28	298.81	298.81	5.86	True

Shift()

La función shift() nos permite acceder al valor de la fila anterior para una columna dada, si usamos shift(-1) accedemos al valor siguiente, de modo general si ponemos shift(n) accedemos a la fila "n" enésima anterior

Esta función nos va a permitir básicamente movernos verticalmente en la tabla

Esto es muy útil para series de datos de precios, por ejemplo para funciones de gap donde debemos comparar la apertura actual contra el cierre de la rueda anterior, es decir debemos hacer un cálculo con valores de filas diferentes

La función nos permitirá calcular cualquier variación o fórmula respecto a diferentes tiempos de la serie ya que utilizando shift(n) nos referiremos a la enésima fila anterior

También es muy útil para calcular valores forward de una serie dada, ya que al poner: shift(-n) estamos refiriéndonos a la enésima fila posterior

Veamos el caso típico de usar esta función para calcular el gap de una rueda (recordemos que el gap es el salto entre el precio de apertura en una rueda dada, respecto al precio de cierre de la rueda anterior o del valor anterior de la serie)

Como la serie la tenemos desde reciente a antiguo, la tenemos que poner al revés para que el anterior sea realmente el anterior y no el día posterior

```
data = pd.read_excel('AAPL.xlsx')
data = data[::-1]
data['cierre_previo']=data['close'].shift()
data['gap_positivo']=data.eval('open>cierre_previo')
data = data.drop(['high','low','volume'], axis=1).set_index("timestamp")
data.head()
```

	open	close	adjusted_close	cierre_previo	gap_positivo
timestamp					
2000-03-06	126.00	125.69	3.8960	NaN	False
2000-03-07	126.44	122.87	3.8086	125.69	True
2000-03-08	122.87	122.00	3.7816	122.87	False
2000-03-09	120.87	122.25	3.7894	122.00	False
2000-03-10	121.69	125.75	3.8979	122.25	False

O bien usar `shift(-1)` que en lugar de buscar hacia atrás buscará hacia adelante, es más prolíjo el código así pero claramente es más confuso de pensarlo

```
data = pd.read_excel('AAPL.xlsx')
data['cierre_previo'] = data['close'].shift(-1)
data['gap_positivo'] = data.eval('open > cierre_previo')
data = data.drop(['high', 'low', 'volume'], axis=1).set_index("timestamp")
data.tail()
```

	open	close	adjusted_close	cierre_previo	gap_positivo
timestamp					
2000-03-10	121.69	125.75	3.8979	122.25	False
2000-03-09	120.87	122.25	3.7894	122.00	False
2000-03-08	122.87	122.00	3.7816	122.87	False
2000-03-07	126.44	122.87	3.8086	125.69	True
2000-03-06	126.00	125.69	3.8960	NaN	False

Otra forma de resolver lo mismo pudo haber sido algo así:

```
data = pd.read_excel('AAPL.xlsx')
data['gap_positivo'] = data.open > data.close.shift(-1)
data = data.drop(['high', 'low', 'volume'], axis=1).set_index("timestamp")
data.tail()
```

	open	close	adjusted_close	gap_positivo
timestamp				
2000-03-10	121.69	125.75	3.8979	False
2000-03-09	120.87	122.25	3.7894	False
2000-03-08	122.87	122.00	3.7816	False
2000-03-07	126.44	122.87	3.8086	True
2000-03-06	126.00	125.69	3.8960	False

Como se darán cuenta siempre hay infinidad de maneras de resolver el mismo problema, yo lo que recomiendo al principio es que utilicen la forma que mejor comprendan al leerla si la escribiera otro, esto en la etapa de aprendizaje es lo mas importante, lo principal es que entiendan lo que están haciendo y no que prueban por probar 20 cosas hasta que alguna funcione y después no sepan por que les funcionó o no, no se rían es mucho más común de lo que piensan esto

Conteo de condiciones por columna

Les muestro simplemente un ejemplo de un conteo por columna de una determinada condición como introducción a un tema que veremos en la próxima unidad que es el agrupamiento de datos, simplemente a modo introductorio veamos distintas formas de contabilizar una determinada condición en una serie de datos (que bien podría ser una columna de un dataframe)

Volvamos al ejemplo anterior en donde armábamos una columna que sea True cuando el Gap de apertura es positivo y False cuando es negativo

La primera y mas intuitiva forma de contabilizar los casos True y False es iterando, es decir haciendo un ciclo FOR que recorra todos los valores, y le vaya sumando un caso a un contador de positivos y a uno de negativos según el caso (En realidad en nuestro ejemplo es negativos o neutros, pero bueno, es un detalle, lo importante es entender las herramientas por ahora)

```
data = pd.read_excel('AAPL.xlsx')
data['gap_positivo'] = data.open > data.close.shift(-1)

positivos = 0
negativos = 0

for dato in data['gap_positivo']:
    if dato == True:
        positivos += 1
    else:
        negativos +=1

print(F"Positivos: {positivos}, Negativos: {negativos}")
```

Positivos: 2816, Negativos: 2217

Bien, problema resuelto

Pero vamos un poco más allá, y preguntémonos ¿de qué otra forma podría haber resuelto esto sin necesidad de iterar (es decir recorrer uno por uno) todos los valores de la columna?

Una primera idea que les dejo es usar la función count() y simplemente una selección por condición

Es decir primero vamos a seleccionar de la columna "gap_positivo" los que son True, y le aplicamos el método count() para contabilizarlos

```
import pandas as pd
data = pd.read_excel('AAPL.xlsx')
data['gap_positivo'] = data.open > data.close.shift(-1)

positivos = data.gap_positivo[data.gap_positivo == True].count()
negativos = data.gap_positivo[data.gap_positivo == False].count()
print(F"Positivos: {positivos}, Negativos: {negativos}")
```

Positivos: 2816, Negativos: 2217

Obviamente me da lo mismo que antes, es la idea.

Bueno, veamos para ir adelantándonos al próximo tema como se podría haber hecho lo mismo de una forma mas "prolija"

Los conteos de determinadas condiciones de una serie SIEMPRE se pueden resolver por agrupamiento

Pandas tiene una función específica para resolver este tipo de problemas que es `groupby()` veamos un mínimo adelanto

```
data = pd.read_excel('AAPL.xlsx')
data['gap_positivo'] = data.open > data.close.shift(-1)
data.groupby('gap_positivo').size()
```

```
gap_positivo
False    2217
True     2816
dtype: int64
```

Fíjense que sencilla manera de resolver el mismo problema, del mismo modo, reseteando el índice vemos la salida como un dataframe mas prolífico, pero es solo un detalle

```
data = pd.read_excel('AAPL.xlsx')
data['gap_positivo'] = data.open > data.close.shift(-1)
data.groupby('gap_positivo').size().reset_index(name='total')
```

gap_positivo	total
0	False 2217
1	True 2816

Datos Perdidos

Saber si tenemos datos NaN, o cuantos tenemos

Una de las cosas más cotidianas a las que nos tenemos que enfrentar cuando tratamos con datos es a los datos perdidos, o datos no numéricos dentro de la serie, esto se puede dar por diversos motivos y estos también dependerán de la fuente de datos, pero por ejemplo si barremos cotizaciones de activos en diferentes mercados, van a haber fechas en las que el mismo activo tendrá un valor en un mercado pero no en el otro ya que en alguno será feriado local.

Otra razón puede no ser exógena a nosotros y puede que la generemos nosotros mismos, es muy

común en el cálculo de ratios que el denominador sea cero y por ende el ratio no se puede calcular generando "un hueco" o datos no válidos, en fin, hay infinidad de factores, lo importante es primero saber reconocer estos datos y luego saber tratarlos, veámoslo

Volvamos al ejemplo que calculamos la variación respecto a la rueda anterior, al empezar la primera rueda de la serie no tiene rueda anterior, es decir que esa primera fila no va a tener un dato válido en la columna variación, y lo llenará con un NaN (Not a number)

```
data = pd.read_csv('AAPL.csv')
data = data[::-1]
data['variacion'] = round(data['adjusted_close'].pct_change() * 100,2)
data = data.drop(["high","low","volume"], axis=1).set_index("timestamp")
data.head()
```

	open	close	adjusted_close	variacion
timestamp				
6/3/2000	126.00	125.69	3.8960	NaN
7/3/2000	126.44	122.87	3.8086	-2.24
8/3/2000	122.87	122.00	3.7816	-0.71
9/3/2000	120.87	122.25	3.7894	0.21
10/3/2000	121.69	125.75	3.8979	2.86

Una forma rápida de saber la cantidad de datos no válidos por columna es usar isnull() y luego una suma ya que le pondrá un 1 a los que cumplen esa condición de ser nulos (no válidos)

```
data.isnull().sum()
```

```
open          0
close         0
adjusted_close 0
variacion     1
dtype: int64
```

Como vemos en la columna variación nos indica que tenemos 1 valor nulo, el resto de las columnas está ok

Como consejo general SIEMPRE verifiquen la cantidad de datos nulos de sus dataframes, es una simple línea de código que no les cuesta nada escribirla y saber esta info les puede ahorrar muchos dolores de cabeza futuros

Tratamiento de datos NaN

Existen varios métodos para tratar los datos no válidos:

- dropna() Los elimina
- fillna(x) Los cambia por x
- replace(x,y) Reemplaza los x por y
- interpolate() Interpola por valor de filas adyacentes

Partimos de estos datos

```
data = pd.read_excel('AAPL.xlsx')
data = data[::-1]
data['variacion'] = round(data['adjusted_close'].pct_change() * 100,2)
data = data.drop(["high","low","volume"], axis=1).set_index("timestamp")
data.head()
```

	open	close	adjusted_close	variacion
timestamp				
2000-03-06	126.00	125.69	3.8960	NaN
2000-03-07	126.44	122.87	3.8086	-2.24
2000-03-08	122.87	122.00	3.7816	-0.71
2000-03-09	120.87	122.25	3.7894	0.21
2000-03-10	121.69	125.75	3.8979	2.86

Eliminamos los datos no válidos con dropna()

```
data = pd.read_excel('AAPL.xlsx')
data = data[::-1]
data['variacion'] = round(data['adjusted_close'].pct_change() * 100,2)
data = data.drop(["high","low","volume"], axis=1).set_index("timestamp")

data = data.dropna()
data.head()
```

	open	close	adjusted_close	variacion
timestamp				
2000-03-07	126.44	122.87	3.8086	-2.24
2000-03-08	122.87	122.00	3.7816	-0.71
2000-03-09	120.87	122.25	3.7894	0.21
2000-03-10	121.69	125.75	3.8979	2.86
2000-03-13	122.12	121.31	3.7602	-3.53

Remplazamos los datos no válidos con fillna()

```
data = pd.read_excel('AAPL.xlsx')
data = data[::-1]
data['variacion'] = round(data['adjusted_close'].pct_change() * 100,2)
data = data.drop(["high","low","volume"], axis=1).set_index("timestamp")

data = data.fillna("Aca estaba el NaN")
data.head()
```

	open	close	adjusted_close	variacion
timestamp				
2000-03-06	126.00	125.69	3.8960	Aca estaba el NaN
2000-03-07	126.44	122.87	3.8086	-2.24
2000-03-08	122.87	122.00	3.7816	-0.71
2000-03-09	120.87	122.25	3.7894	0.21
2000-03-10	121.69	125.75	3.8979	2.86

Remplazamos un valor por otro con replace()

```
data = pd.read_excel('AAPL.xlsx')
data = data[::-1]
data['variacion'] = round(data['adjusted_close'].pct_change() * 100,2)
data = data.drop(["high","low","volume"], axis=1).set_index("timestamp")

data = data.replace(-2.24,"Aca estaba el -2.24")
data.head()
```

	open	close	adjusted_close	variacion
timestamp				
2000-03-06	126.00	125.69	3.8960	NaN
2000-03-07	126.44	122.87	3.8086	Aca estaba el -2.24
2000-03-08	122.87	122.00	3.7816	-0.71
2000-03-09	120.87	122.25	3.7894	0.21
2000-03-10	121.69	125.75	3.8979	2.86

Interpolado con el método interpolate()

En este caso no tiene sentido, solo lo pongo para mostrar la herramienta, pero supongamos que necesite calcular algún indicador de ratios que dependa de un valor vacío en alguna columna, en ese caso para "darle continuidad" a la serie podemos interpolar el dato vacío por los datos anterior y posterior

En el ejemplo usamos el parámetro `limit_direction` porque al tener vacío el primer valor de la serie y no tener ningún valor anterior si no le indicamos nada no lo completaría, los valores posibles son "both", "backward" y "forward", en este caso con "backward" también funciona porque el que le falta es el anterior para interpolar. Cabe aclarar que por default hace una interpolación lineal pero en la documentación oficial de pandas pueden encontrar todos los parámetros disponibles y los diferentes métodos de interpolación que se pueden aplicar

```
data = pd.read_excel('AAPL.xlsx')
data = data[::-1]
data['variacion'] = round(data['adjusted_close'].pct_change() * 100,2)
data = data.drop(['high','low','volume'], axis=1).set_index("timestamp")

data = data.interpolate(limit_direction='both')
data.head()
```

	open	close	adjusted_close	variacion
timestamp				
2000-03-06	126.00	125.69	3.8960	-2.24
2000-03-07	126.44	122.87	3.8086	-2.24
2000-03-08	122.87	122.00	3.7816	-0.71
2000-03-09	120.87	122.25	3.7894	0.21
2000-03-10	121.69	125.75	3.8979	2.86

Promedios Móviles

Método Rolling

El método rolling nos permite realizar medias móviles con mucha facilidad pasándole a la función como argumento la cantidad de ruedas hacia atrás que necesitamos tomar en cuenta, y por último el tipo de operación que queremos, si buscamos una media móvil esa operación será mean(), si buscamos una volatilidad será std()

ATENCION: Es importante que si vamos a utilizar el método rolling tengamos los datos ordenados cronológicamente de antiguos a recientes ya que de lo contrario estaríamos calculando un forward-rolling (podría ser útil para buscar correlaciones o para backtestings pero ya veremos más adelante eso)

```
data = pd.read_excel('AAPL.xlsx')
data.sort_values('timestamp', inplace=True)
data['SMA 20'] = data.adjusted_close.rolling(20).mean()
data = data.drop(['high', 'low', 'volume'], axis=1).set_index("timestamp")

data = data.dropna()
data
```

	open	close	adjusted_close	SMA 20
timestamp				
2000-03-31	127.44	135.81	4.2097	3.991245
2000-04-03	135.50	133.31	4.1322	4.003055
2000-04-04	132.63	127.31	3.9462	4.009935
2000-04-05	126.47	130.38	4.0414	4.022925
2000-04-06	130.63	125.19	3.8805	4.027480
...
2020-03-02	282.28	298.81	298.8100	310.494580
2020-03-03	303.67	289.32	289.3200	309.564625
2020-03-04	296.44	302.74	302.7400	308.797390
2020-03-05	295.52	292.92	292.9200	307.409470
2020-03-06	282.00	289.03	289.0300	305.639500

5014 rows × 4 columns

Ojo con el formato en que vienen ordenadas las fechas, esto a veces nos podemos confiar demasiado y resulta que vienen en un string que termina reconociendo como string y no como fecha, ergo cuando ordenemos por fecha vamos a tener dolores de cabeza importantes

```
data = pd.read_csv('AAPL.csv')
data['timestamp'].astype

<bound method NDFrame.astype of
0                               6/3/2020
5032                            6/3/2000
Name: timestamp, Length: 5033, dtype: object>
```

De hecho fijense que pasa si ordenamos por fecha eso así como viene:

Como verán tomó la fecha como un string y ordenó las fechas alfabéticamente (que no es lo mismo que ordenar por fecha)

```
data = pd.read_csv('AAPL.csv')
data = data.sort_values(by='timestamp', ascending=True)
data = data.drop(['high', 'low', 'volume'], axis=1).set_index("timestamp")
```

```
data
```

	open	close	adjusted_close
timestamp			
1/10/2001	15.49	15.540	0.9634
1/10/2002	14.59	14.510	0.8995
1/10/2003	20.75	20.790	1.2889
1/10/2004	39.15	38.670	2.3973
1/10/2007	154.63	156.339	19.3842
...
9/9/2013	505.00	506.170	64.5341
9/9/2014	99.08	97.990	89.4259
9/9/2015	113.76	110.150	102.2339
9/9/2016	104.64	103.130	97.7618
9/9/2019	214.84	214.170	213.0237

5033 rows × 3 columns

Veamos los tipos:

```
data = pd.read_excel('AAPL.xlsx')
data['timestamp'].astype
```

```
<bound method NDFrame.astype of 0      2020-03-06
5032    2000-03-06
Name: timestamp, Length: 5033, dtype: datetime64[ns]>
```

En cambio, si tomamos el Excel en lugar del CSV, fijense la diferencia:

```
data = pd.read_excel('AAPL.xlsx')
data = data.sort_values(by='timestamp', ascending=True)
data = data.drop(['high', 'low', 'volume'], axis=1).set_index("timestamp")
data
```

	open	close	adjusted_close
timestamp			
2000-03-06	126.00	125.69	3.8960
2000-03-07	126.44	122.87	3.8086
2000-03-08	122.87	122.00	3.7816
2000-03-09	120.87	122.25	3.7894
2000-03-10	121.69	125.75	3.8979
...
2020-03-02	282.28	298.81	298.8100
2020-03-03	303.67	289.32	289.3200
2020-03-04	296.44	302.74	302.7400
2020-03-05	295.52	292.92	292.9200
2020-03-06	282.00	289.03	289.0300

5033 rows × 3 columns

En este caso si ordenó correctamente las fechas cuando toma los datos de un libro de Excel versus cuando los toma de un CSV, a pesar que son los mismos datos, no son el mismo formato en que están almacenados, el CSV almacena todo como texto plano mientras que el Excel tiene un montón de tipos de formatos prefijados, muchas veces nos salvan las papas, como en este caso que nos facilitó enormemente el tema fechas, pero otras veces puede ser un dolor de cabeza como por ejemplo cuando hay enteros muy grandes y quedan configurados como notación científica.

Lo importante es siempre estar atentos cuando traemos datos de cualquier fuente y lo primero que debemos acostumbrarnos a hacer es verificar que los datos que trae, los trae como los tiene que traer, para eso nada mejor que someterlos a pruebas de ordenamiento, filtro etc..

Usamos rolling para calcular un Rolling sigma (desvió estándar)

Primero vamos a calcular para cada fecha de la tabla el desvió estándar o volatilidad de los rendimientos diarios, y lo vamos a anualizar multiplicando por la raíz de la cantidad de ruedas que hay en un año

Para ello apelamos al método `rolling(n).std()` ya que rolling nos proporcionará la ventana de "n" elementos a los que les aplicamos el desvió estándar

```
import math
data = pd.read_excel('AAPL.xlsx')
data = data.sort_values(by='timestamp', ascending=True)

res = pd.DataFrame()
res['fecha'] = data['timestamp']
res['variacion'] = round(data['adjusted_close'].pct_change() * 100, 2)
res['sigma_40'] = round(res.variacion.rolling(40).std() * math.sqrt(250), 2)
res = res.dropna()
res.head()
```

	fecha	variacion	sigma_40
4992	2000-05-02	-5.18	73.87
4991	2000-05-03	-2.38	73.90
4990	2000-05-04	-3.80	74.47
4989	2000-05-05	2.19	74.70
4988	2000-05-08	-2.65	74.57

Ahora que ya tenemos el sigma puntual de cada fecha para sus últimas 40 ruedas vamos a calcular la media móvil mensual de esa volatilidad (media móvil mensual nos referimos a la volatilidad de 40 ruedas promedio de las últimas 20)

```
data = pd.read_excel('AAPL.xlsx')
data = data.sort_values(by='timestamp', ascending=True)

res = pd.DataFrame()
res['fecha'] = data['timestamp']
res['variacion'] = round(data['adjusted_close'].pct_change() * 100, 2)
res['sigma_40'] = round(res.variacion.rolling(40).std() * math.sqrt(250), 2)
res['sigma_40_Rolling'] = round(res.sigma_40.rolling(20).mean(), 2)

res = res.dropna()
res.head()
```

	fecha	variacion	sigma_40	sigma_40_Rolling
4973	2000-05-30	1.38	68.37	73.78
4972	2000-05-31	-4.07	68.77	73.53
4971	2000-06-01	6.10	70.46	73.35
4970	2000-06-02	3.86	70.97	73.18
4969	2000-06-05	-1.35	70.51	72.97

Si observamos bien el resultado vemos que esta tabla arranca de mas tarde, casi un mes más tarde que la anterior, y es así porque el método rolling del sigma_40 necesita una ventana de 20 datos que fue lo que le pedimos, por lo tanto, hasta no haber pasado los primeros 20 datos de la serie anterior, no tenemos datos completos y como usamos luego la función dropna() es ahí donde eliminamos esos datos

Atenti! Usando round() a todo el frame

Otra manera de redondear toda la tabla es utilizar el método round(n) de pandas, donde "n" es la cantidad de decimales. Cuando apliquemos este método a toda la tabla los números enteros o los textos no los va a afectar la función, solo afecta a los números flotantes

Y otra manera de calcular la raíz cuadrada para no importar la librería math es usando la potencia ** y elevarlo a la 1/2 o 0.5 que es lo mismo que aplicar raíz cuadrada

```
data = pd.read_excel('AAPL.xlsx')
data = data.sort_values(by='timestamp', ascending=True)

res = pd.DataFrame()
res['fecha'] = data['timestamp']
res['variacion'] = data['adjusted_close'].pct_change() * 100
res['sigma_40'] = res.variacion.rolling(40).std() * math.sqrt(250)
res['sigma_40_Rolling'] = res.sigma_40.rolling(20).mean()

res = res.dropna().round(2)
res.head()
```

	fecha	variacion	sigma_40	sigma_40_Rolling
4973	2000-05-30	1.38	68.37	73.78
4972	2000-05-31	-4.07	68.77	73.53
4971	2000-06-01	6.10	70.45	73.35
4970	2000-06-02	3.86	70.97	73.18
4969	2000-06-05	-1.35	70.50	72.97

Ahora observen con atención este cuadro y el anterior y veamos que cosa rara hay!!
Pero como ¿no son iguales? ¿Hay algún error del programa? jaja, me río porque me parafraseo a

mi mismo cuando vi esto por primera vez.. no, jamás piensen que es un error de python, por lo general en el 100% de los casos es un error nuestro!! Pero ¿dónde está el error? por que da diferente? cual es el correcto?

Lo cierto es que no hay errores, ambos son diferentes y correctos ya que en el primer caso redondeo las variaciones, y con esas variaciones redondeadas luego calculo los sigmas, y con esos sigmas redondeados, luego calculo las medias móviles de esos sigmas.. En el otro caso calculo todo y al final redondeo todo junto

Es más razonable la segunda forma de resolver esto, porque en la primera estamos arrastrando errores pequeños pero que si los arrastramos mucho los amplificamos, no es parte del contenido de este curso así que no nos vamos a extender en este tema más, solo quería mostrar el ejemplo porque es un vicio muy común ir redondeando a mitad de camino resultados parciales que luego son inputs de otros resultados a redondear

Así que quedense con eso, conviene calcular todo lo que haya que calcular, y al final de todo, cuando ya no queden cuentas que hacer, ahí redondeamos los outputs o los resultados finales

Método ewm

Medias móviles exponenciales con método ewm()

Para las medias móviles exponenciales podemos usar el método `ewm`, para los interesados en aprender mas de este método pueden ir a la documentación, pero les adelanto que si en lugar de usar el atributo `span=n`, siendo n la cantidad de ruedas hacia atrás con un $\alpha = 2(n+1)$, pueden directamente setear el α con el atributo "alpha"

Veamos el ejemplito para calcular una media móvil exponencial de 20 ruedas

```
data = pd.read_excel('AAPL.xlsx')
data = data.sort_values(by='timestamp', ascending=True)

data['SMA 20'] = data.adjusted_close.rolling(20).mean()
data['EMA 20'] = data.adjusted_close.ewm(span=20, adjust=False).mean()
data = data.drop(['high', 'low', 'volume'], axis=1).set_index("timestamp")

data = data.dropna().round(2)
data.head()
```

	open	close	adjusted_close	SMA 20	EMA 20
timestamp					
2000-03-31	127.44	135.81		4.21	3.99
2000-04-03	135.50	133.31		4.13	4.00
2000-04-04	132.63	127.31		3.95	4.01
2000-04-05	126.47	130.38		4.04	4.02
2000-04-06	130.63	125.19		3.88	4.03
					4.04

Funciones Acumulativas

Las funciones acumulativas nos permiten trabajar en la fila "n" de una tabla con todos los datos de las filas "0 a n"

Dentro de este tipo de funciones encontramos:

- cummax() El máximo acumulado (ideal para máximo histórico por fecha)
- cummin() El mínimo acumulado (ideal para mínimo histórico por fecha)
- cumsum() La suma acumulada (ideal para armado de subtotales por fecha)
- cumprod() El producto acumulado (ideal para rendimiento compuesto)

Cummax()

El uso típico que le vamos a dar a esta función es para saber el máximo histórico de una serie en cada punto de la misma

```
import pandas as pd
data = pd.read_excel('AAPL.xlsx')
data = data.sort_values(by='timestamp', ascending=True)
data = data.drop(['high', 'low', 'volume'], axis=1).set_index("timestamp")

data['maxHist'] = data.adjusted_close.cummax()
data.head(6)
```

	open	close	adjusted_close	maxHist
timestamp				
2000-03-06	126.00	125.69	3.8960	3.8960
2000-03-07	126.44	122.87	3.8086	3.8960
2000-03-08	122.87	122.00	3.7816	3.8960
2000-03-09	120.87	122.25	3.7894	3.8960
2000-03-10	121.69	125.75	3.8979	3.8979
2000-03-13	122.12	121.31	3.7602	3.8979

Cummin()

Obviamente es lo mismo que el cummax pero para mínimos, la combinación de cummax() y cummin() va a ser muy útil para backtestings de drawdowns y recuperaciones posteriores

```
data = pd.read_excel('AAPL.xlsx')
data = data.sort_values(by='timestamp', ascending=True)
data = data.drop(['high', 'low', 'volume'], axis=1).set_index("timestamp")

data['minHist'] = data.adjusted_close.cummin()
data.head(6)
```

	open	close	adjusted_close	minHist
timestamp				
2000-03-06	126.00	125.69	3.8960	3.8960
2000-03-07	126.44	122.87	3.8086	3.8086
2000-03-08	122.87	122.00	3.7816	3.7816
2000-03-09	120.87	122.25	3.7894	3.7816
2000-03-10	121.69	125.75	3.8979	3.7816
2000-03-13	122.12	121.31	3.7602	3.7602

Cumsum()

Cumsum() es obviamente una función de sumas acumuladas, que en la posición de la fila "n" nos devuelve la suma de "0 a n" (inclusive)

$$\text{cumsum } (X_n) = \sum_{i=0}^n x_i$$

En el ejemplo aprovechamos para borrar las columnas de OHLC par limpiar un poco la salida con la función drop()

```
data = pd.read_excel('AAPL.xlsx')
data = data.sort_values(by='timestamp', ascending=True)

data['volumenAcum'] = data.volume.cumsum()/1000000
data = data.drop(['open','high','low','close'],1).dropna().round(2)
data.head(6)
```

	timestamp	adjusted_close	volume	volumenAcum
5032	2000-03-06	3.90	1880000	1.88
5031	2000-03-07	3.81	2437600	4.32
5030	2000-03-08	3.78	2421700	6.74
5029	2000-03-09	3.79	2470700	9.21
5028	2000-03-10	3.90	2219700	11.43
5027	2000-03-13	3.76	2713900	14.14

Cumprod()

`Cumprod()` es una función de productorio, es decir el producto acumulado de 0 a n, para la fila n

$$\text{cumprod } (X_n) = \prod_{i=0}^n x_i$$

Vamos a usar esta fórmula para calcular rendimiento compuesto, es muy sencillo:

- 1- Creamos una columna "change_pct" con el valor "r", rendimiento porcentual diario.
- 2- Creamos una columna "dailyFactor" con el valor $(1+r)$
- 3- Aplicamos el productorio a cada fila de la columna "dailyFator" y restamos 1 al resultado.

```
data = pd.read_excel('AAPL.xlsx')
data = data.sort_values(by='timestamp', ascending=True)

data['change_pct'] = data['adjusted_close'].pct_change()
data['dailyFactor'] = 1 + data['change_pct']
data['rendimientoAcum'] = (data.dailyFactor.cumprod()-1)*100

data = data.drop(['open','high','low','close','volume'],1).dropna().round(4).
set_index("timestamp")
data.head(6)
```

	adjusted_close	change_pct	dailyFactor	rendimientoAcum
timestamp				
2000-03-07	3.8086	-0.0224	0.9776	-2.2433
2000-03-08	3.7816	-0.0071	0.9929	-2.9363
2000-03-09	3.7894	0.0021	1.0021	-2.7361
2000-03-10	3.8979	0.0286	1.0286	0.0488
2000-03-13	3.7602	-0.0353	0.9647	-3.4856
2000-03-14	3.5414	-0.0582	0.9418	-9.1016

Por supuesto que lo podríamos haber hecho en una sola línea, pero la verdad es medio ilegible el código, no se entiende bien que hace toda esa línea y seguramente el dia que querramos modificar algo y repasemos el código no vamos a entender lo que escribimos ahí y eso nos puede hacer dudar de si estaba bien o no, etc etc..

Lo que quiero decir es que a veces por meter en una sola linea forzadamente muchas instrucciones terminamos haciendo un código compacto pero complejo de leer a simple vista y la idea de "mantenibilidad" del código es buscar que sea fácilmente legible y entendible que hace cada línea

Les dejo acá como quedaría remplazando esos 3 pasos en uno solo, saquen sus propias conclusiones

```
data = pd.read_excel('AAPL.xlsx')
data = data.sort_values(by='timestamp', ascending=True)

data['rendimientoAcum'] = ((1 +
data['adjusted_close'].pct_change()).cumprod ()-1)*100
data = data.drop(['open','high','low','close'],1).dropna().round(4)
data.head(6)
```

	timestamp	adjusted_close	volume	rendimientoAcum
5031	2000-03-07	3.8086	2437600	-2.2433
5030	2000-03-08	3.7816	2421700	-2.9363
5029	2000-03-09	3.7894	2470700	-2.7361
5028	2000-03-10	3.8979	2219700	0.0488
5027	2000-03-13	3.7602	2713900	-3.4856
5026	2000-03-14	3.5414	3826600	-9.1016

Funciones de Conteo

Las funciones de conteo nos sirven para saber donde estamos parados cuando recibimos un dataset, una vez recibido se supone que a partir de allí tenemos el control, "se supone" digo porque a veces por más que partamos de una serie de 1000 datos si hacemos operaciones con métodos de rolling (pasados) o forward usando shift(-n) vamos a ver que terminamos con menos datos de los que empezamos en esas columnas y no siempre es algo que se tiene en cuenta y puede ocasionar serios problemas

Bueno, vamos que métodos de conteo tenemos disponibles:

count(), any() y all()

- count() devuelve para cada columna la cantidad de valores validos
- any() devuelve True a las columnas que al menos tienen un valor verdadero
- all() devuelve True a las columnas a las que TODOS sus valores son verdaderos

Recuerden que cualquier número distinto de 0 se evalúa como True

Para entender cómo funcionan veamos un ejemplo en el que sabemos que vamos a generar "huecos" de datos.

Vamos a calcular por ejemplo:

- 1 - Variación diaria, ya sabemos que perdemos el primer dato de la serie porque no tenemos el Precio de cierre del día anterior al inicial
- 2 - Media móvil de 200 ruedas (Obviamente acá vamos a perder los primeros 199 datos)
- 3 - Variación Forward Semanal, acá vamos a perder los últimos 5 datos ya que 6 ruedas antes de llegar al final de la serie no tendremos su cierre t+5

Veamos que me devuelve el método count()

```
data = pd.read_excel('AAPL.xlsx')
data = data.sort_values(by='timestamp', ascending=True)

data['vraicionDiaria'] = data.adjusted_close.pct_change()
data['SMA 200'] = data.adjusted_close.rolling(200).mean()
data['Variacion Forward Semanal'] = data.adjusted_close.shift(-5)/data.adjusted_close - 1
data = data.drop(['open','high','low','close'],1)

data.count()

timestamp                5033
adjusted_close             5033
volume                     5033
vraicionDiaria            5032
SMA 200                    4834
Variacion Forward Semanal  5028
dtype: int64
```

Como verán de los 5033 datos, la variación diaria tiene 1 menos, la media móvil de 200 tiene 199 datos menos, y la variación forward semanal 5 datos menos

Ahora pregunto ¿que pasará con la cantidad de filas si aplica a la tabla final el método dropna()?

No prueben el código, deténganse a pensarlo, es bueno ejercitarse el pensamiento lógico antes de probar por probar

Si están suponiendo que el número final es el de menor cantidad de filas es un error.
Piensen por qué les digo esto

Bueno, si no quieren pensar les pego acá el código pero no les voy a explicar por qué da menos que la columna con menor cantidad, dedúzcanlo solos

```
data = pd.read_excel('AAPL.xlsx')
data = data.sort_values(by='timestamp', ascending=True)

data['vraiacionDiaria'] = data.adjusted_close.pct_change()
data['SMA 200'] = data.adjusted_close.rolling(200).mean()
data['Variacion Forward Semanal'] = data.adjusted_close.shift(-5)/data.adjusted_close -1
data = data.drop(['open','high','low','close'],1).dropna()

data.count()
```

timestamp	4829
adjusted_close	4829
volume	4829
vraiacionDiaria	4829
SMA 200	4829
Variacion Forward Semanal	4829
dtype: int64	

Veamos conteos de True o False.. Para ello vamos a crear las siguientes columnas:

- 1- IntraBajo: True si el movimiento intradiario, su valor absoluto, es menor o igual a 2% (alcista o bajista), False si pasa lo contrario
- 2- IntraMedio: True si el movimiento intradiario es menor o igual a 5% y mayor a 2%
- 3- IntraAlto: True si el movimiento intradiario es menor o igual al 15% y mayor a 5%
- 4- IntraInmenso: True si el movimiento intradiario es mayor al 15%

```

data = pd.read_excel('AAPL.xlsx')
data = data.sort_values(by='timestamp', ascending=True)

data['Intra %'] = (data.close / data.open - 1)*100
data['IntraBajo'] = abs(data['Intra %']) <= 2
data['IntraMedio'] = (abs(data['Intra %']) > 2) & (abs(data['Intra %']) <= 5)
data['IntraAlto'] = (abs(data['Intra %']) > 5) & (abs(data['Intra %']) <= 15)
data['IntraEnorme'] = abs(data['Intra %']) > 15

data = data.drop(['open','high','low','close','volume','adjusted_close'],1)
data.set_index("timestamp", inplace=True)

# La siguiente opción la ponemos para mostrar el formato con:
# Separador de miles: coma
# Separador decimal: Punto
# Cantidad de decimales: 2
pd.options.display.float_format = '{:,.2f}'.format

data.head()

```

	Intra %	IntraBajo	IntraMedio	IntraAlto	IntraEnorme
timestamp					
2000-03-06	-0.25	True	False	False	False
2000-03-07	-2.82	False	True	False	False
2000-03-08	-0.71	True	False	False	False
2000-03-09	1.14	True	False	False	False
2000-03-10	3.34	False	True	False	False

```
data.sum()
```

```

adjusted_close    293,807.51
Intra %          125.41
IntraBajo         3,827.00
IntraMedio        1,043.00
IntraAlto         163.00
IntraEnorme      0.00
dtype: float64

```

Como vemos del único intervalo que no hay ningún True es IntraEnorme

Entonces veamos lo que nos devuelve la función all()

Esta función es True cuando TODOS los elementos son True o evalúan como True

```
data = pd.read_excel('AAPL.xlsx')
data = data.sort_values(by='timestamp', ascending=True)

data['Intra %'] = (data.close / data.open - 1) * 100
data['IntraBajo'] = abs(data['Intra %']) <= 2
data['IntraMedio'] = (abs(data['Intra %']) > 2) & (abs(data['Intra %']) <= 5)
data['IntraAlto'] = (abs(data['Intra %']) > 5) & (abs(data['Intra %']) <= 15)
data['IntraEnorme'] = abs(data['Intra %']) > 15
data = data.drop(['open', 'high', 'low', 'close'], 1)

data.all()
```

timestamp	True
adjusted_close	True
volume	True
Intra %	False
IntraBajo	False
IntraMedio	False
IntraAlto	False
IntraEnorme	False
dtype: bool	

Claramente la función all() nos devuelve false a todos los intervalos ya que en ninguno de ellos son todas las filas True

Sin embargo la función any() que devuelve True cuando ALGUNA de las filas es True, nos va a dar True en todos los intervalos excepto en IntraEnorme

```
data = pd.read_excel('AAPL.xlsx')
data = data.sort_values(by='timestamp', ascending=True)

data['Intra %'] = (data.close / data.open - 1) * 100
data['IntraBajo'] = abs(data['Intra %']) <= 2
data['IntraMedio'] = (abs(data['Intra %']) > 2) & (abs(data['Intra %']) <= 5)
data['IntraAlto'] = (abs(data['Intra %']) > 5) & (abs(data['Intra %']) <= 15)
data['IntraEnorme'] = abs(data['Intra %']) > 15
data = data.drop(['open', 'high', 'low', 'close'], 1)

data.any()
```

timestamp	True
adjusted_close	True
volume	True
Intra %	True
IntraBajo	True
IntraMedio	True
IntraAlto	True
IntraEnorme	False
dtype: bool	

Links de interés

- Pandas Doc
<https://pandas.pydata.org/pandas-docs/stable/reference/frame.html>
[\(https://pandas.pydata.org/pandas-docs/stable/reference/frame.html\)](https://pandas.pydata.org/pandas-docs/stable/reference/frame.html)
<http://bit.ly/2TP848V> (<http://bit.ly/2TP848V>)

Material y Archivos Usados

- En esta unidad Usamos: AAPL.csv , AAPL.xlsx , SPY.xlsx y QQQ.xlsx
<http://bit.ly/39Am64T> (<http://bit.ly/39Am64T>)

Ejercicios Propuestos

Tomar la serie de precios históricos de AAPL del archivo AAPL.xlsx y en función de ello:

- 1- Armar una tabla que tenga como índice la fecha, y como columnas el volumen y otra columna llamada "Dif Volumen" con los valores absolutos en dólares de variación de volumen respecto a la rueda anterior
- 2- Armar una tabla desde cero que tenga solo las fechas como índice, y como columnas los valores de cierre sin ajustar y los valores absolutos en dólares de variación de los precios de cierre sin ajustar respecto a 5 ruedas forward
- 3- Armar una Tabla que tenga la fecha como índice y una columna para el "open", otra "close", otra con el movimiento intradiario (Close-Open) y otra con el movimiento intradiario de la rueda anterior, plantear dos alternativas diferentes para el cálculo
- 4- Armar con la tabla de los movimientos intradiarios del día y de la rueda anterior (en % sobre sus respectivos precios de apertura), que tenga solo la fecha de índice, esas dos columnas (movimiento intradiario y de la rueda anterior) y una tercera columna que tenga True solo cuando el valor absoluto del movimiento intradiario sea mayor al de la rueda anterior
- 5- Contabilizar los casos en que se cumple la condición que el valor absoluto del movimiento intradiario es mayor al 3%
- 6- Armar una tabla con el valor del Gap de apertura y con el valor de la variación porcentual del

cierre ajustado 5 ruedas hacia adelante. La tabla no debe contener valores inválidos en ninguna columna

7- Dada la tabla del ejercicio anterior, con el valor del Gap de apertura y con el valor de la variación porcentual del cierre ajustado 5 ruedas hacia adelante:

a - Generar a partir de ella una nueva tabla solo con las filas cuyos gaps sean mayores a +1%

b - Contabilizar la cantidad de datos nulos de la tabla del item "a" c - Calcular el % de casos de esta última tabla en los cuales la variación forward es positiva

8- Calcular la media móvil de 20 ruedas del volumen medio en millones de dólares.

Para estimar el volumen en dólares usar el volumen en nominales multiplicado por el precio promedio (OHLC). Devolver una tabla solo con el precio medio, el volumen en millones y la media de 20 ruedas del volumen en millones sin datos nulos

9- Armar una tabla solo con los siguientes datos:

- a- Fecha
- b- Precio de cierre ajustado
- c- Media móvil simple de 50 ruedas
- d- Media móvil simple de 200 ruedas
- e- Variación forward a 20 ruedas

10- dada la tabla del ejercicio anterior con el cruce de medias simples de 50 y 200, calcular que porcentaje de las ruedas en los que el cruce es positivo (SMA 50 > SMA 200) la variación en las siguientes 20 ruedas es positiva

11 - Leer el archivo de precios históricos y armar una tabla con una media móvil exponencial rápida y una media móvil exponencial lenta (que previamente le pasamos como variables: fasto y slow) y un valor de cruce que sea el % que la media móvil rápida está por encima de la media móvil lenta

La tabla resultante tiene que tener solo el cierre ajustado, las medias móviles y el cruce. Las columnas deben tomar automáticamente el nombre "SMA X" con x representando los valores de las variables fast y slow

12 - Leer el archivo de precios históricos y mostrar la fecha, el cierre ajustado, y el cierre máximo histórico hasta cada fecha

- a- Hacerlo sin usar la función cummax
- b- Hacerlo usando cummax y comparar los resultados con el punto "a"

13- Armar una tabla con los precios de AAPL con la fecha, el precio de cierre ajustado y el rendimiento acumulado (tasa compuesta diaria) en la columna Yield partiendo desde el 1ro de enero de 2008 y hasta el 31 de diciembre de 2008

13b- Devolver solo el ultimo valor de la columna Yield (el rendimiento acumulado de 2008) en la variable "rta"

13c- Poner al inicio del código una variable llamada "año" que tenga el año al que quiera saber su rendimiento acumulado y me devuelva dicho rendimiento anual

14- Usando el código del ejercicio "13c", generar una lista con los años desde 2005 a 2015 y devolver un DataFrame con los rendimientos acumulados (tasa compuesta diaria) año por año

```

#-----#
# Rta Ejercicio 1 #
#-----#

import pandas as pd
data = pd.read_excel('AAPL.xlsx')
data = data[::-1]
data.set_index('timestamp', inplace=True)
data = data.drop(['open', 'high', 'low', 'close', 'adjusted_close'], 1)
data['Dif Volumen'] = data['volume'].diff()
data.head()

```

	volume	Dif Volumen
timestamp		
2000-03-06	1880000	NaN
2000-03-07	2437600	557600.0
2000-03-08	2421700	-15900.0
2000-03-09	2470700	49000.0
2000-03-10	2219700	-251000.0

```

#-----#
# Rta Ejercicio 2 #
#-----#

data = pd.read_excel('AAPL.xlsx')
data = data[::-1]

res = data.filter(['timestamp', 'close'], axis=1)
res.set_index('timestamp', inplace=True)
res['Delta Precio 5r Forward'] = res['close'].diff(-5)
res.head()

```

	close	Delta Precio 5r Forward
timestamp		
2000-03-06	125.69	4.38
2000-03-07	122.87	8.62
2000-03-08	122.00	5.75
2000-03-09	122.25	0.69
2000-03-10	125.75	0.75

```

#-----#
# Rta Ejercicio 3  #
#-----#
data = pd.read_excel('AAPL.xlsx')
data = data[::-1]
res = data.filter(['timestamp','close','open'],axis=1)
res.set_index('timestamp' , inplace=True)
res['Intra'] = res['close'] - res['open']
res['IntraAnterior'] = res['close'].shift(1) - res['open'].shift(1)
res.head()

```

	close	open	Intra	IntraAnterior
timestamp				
2000-03-06	125.69	126.00	-0.31	NaN
2000-03-07	122.87	126.44	-3.57	-0.31
2000-03-08	122.00	122.87	-0.87	-3.57
2000-03-09	122.25	120.87	1.38	-0.87
2000-03-10	125.75	121.69	4.06	1.38

```

#-----#
# Rta Ejercicio 3  #
#-----#
# 2da Alternativa

data = pd.read_excel('AAPL.xlsx')
data = data[::-1]

res = data.filter(['timestamp','close','open'],axis=1)
res.set_index('timestamp' , inplace=True)
res['Intra'] = res['close'] - res['open']
res['IntraAnterior'] = res['Intra'].shift(1) res.head()

```

	close	open	Intra	IntraAnterior
timestamp				
2000-03-06	125.69	126.00	-0.31	NaN
2000-03-07	122.87	126.44	-3.57	-0.31
2000-03-08	122.00	122.87	-0.87	-3.57
2000-03-09	122.25	120.87	1.38	-0.87
2000-03-10	125.75	121.69	4.06	1.38

```
#-----#
# Rta Ejercicio 4 #
#-----#
data = pd.read_excel('AAPL.xlsx')
data = data[::-1]
res = data.filter(['timestamp','close','open'],axis=1)
res.set_index('timestamp' , inplace=True)
res['Intra'] = (res['close'] / res['open'] -1)*100
res['IntraAnterior'] = res['Intra'].shift(1)
res['Condicion'] = abs(res['Intra']) > abs(res['IntraAnterior'])
res.head()
```

	close	open	Intra	IntraAnterior	Condicion
timestamp					
2000-03-06	125.69	126.00	-0.246032	NaN	False
2000-03-07	122.87	126.44	-2.823474	-0.246032	True
2000-03-08	122.00	122.87	-0.708065	-2.823474	False
2000-03-09	122.25	120.87	1.141723	-0.708065	True
2000-03-10	125.75	121.69	3.336346	1.141723	True

```
#-----#
# Rta Ejercicio 5 #
#-----#
data = pd.read_excel('AAPL.xlsx')
data = data[::-1]
res = data.filter(['timestamp','close','open'],axis=1)
res.set_index('timestamp' , inplace=True)
res['Intra'] = (res['close'] / res['open'] -1)*100
res['Condicion'] = abs(res['Intra']) > 3
res.Condicion.sum()
```

611

```
#-----#
# Rta Ejercicio 6 #
#-----#
data = pd.read_excel('AAPL.xlsx')
data = data[::-1]

res = data.filter(['timestamp','close','open','adjusted_close'],axis=1)
res.set_index('timestamp' , inplace=True)
res['Gap'] = (res['open'] / res['close'].shift(1) -1)*100
res['VarForward'] = (res['adjusted_close'].shift(-5) /
res['adjusted_close'] -1)*100
res.dropna().head()
```

	close	open	adjusted_close	Gap	VarForward
timestamp					
2000-03-07	122.87	126.44	3.8086	0.596706	-7.015701
2000-03-08	122.00	122.87	3.7816	0.000000	-4.712291
2000-03-09	122.25	120.87	3.7894	-0.926230	-0.564733
2000-03-10	125.75	121.69	3.8979	-0.458078	-0.597758
2000-03-13	121.31	122.12	3.7602	-2.886680	1.393543

```
#-----#
# Rta Ejercicio 7a #
#-----#
data = pd.read_excel('AAPL.xlsx')
data = data[::-1]

res = data.filter(['timestamp','close','open','adjusted_close'],axis=1)
res.set_index('timestamp' , inplace=True)
res['Gap'] = (res['open'] / res['close'].shift(1) -1)*100
res['VarForward'] = (res['adjusted_close'].shift(-5) /
res['adjusted_close'] -1)*100

filtro.head()
```

	close	open	adjusted_close	Gap	VarForward
timestamp					
2000-03-15	116.25	115.62	3.6034	1.199125	24.035633
2000-03-31	135.81	127.44	4.2097	1.343936	-2.988336
2000-04-07	131.75	127.25	4.0839	1.645499	-15.090967
2000-04-13	113.81	111.50	3.5278	2.059497	4.444696
2000-04-20	118.87	123.69	3.6846	2.121863	4.366824

```
#-----#
# Rta Ejercicio 7b #
#-----#
data = pd.read_excel('AAPL.xlsx')
data = data[::-1]
res = data.filter(['timestamp','close','open','adjusted_close'],axis=1)
res.set_index('timestamp' , inplace=True)
res['Gap'] = (res['open'] / res['close'].shift(1) -1)*100
res['VarForward'] = (res['adjusted_close'].shift(-5) /
res['adjusted_close'] -1)*100
res.isnull().sum()
```

close	0
open	0
adjusted_close	0
Gap	1
VarForward	5
dtype: int64	

```
#-----#
# Rta Ejercicio 7c #
#-----#
data = pd.read_excel('AAPL.xlsx')
data = data[::-1]
res = data.filter(['timestamp','close','open','adjusted_close'],axis=1)
res.set_index('timestamp' , inplace=True)
res['Gap'] = (res['open'] / res['close'].shift(1) -1)*100
res['VarForward'] = (res['adjusted_close'].shift(-5)/res['adjusted_close']-1)*100
filtro = res.loc[res.Gap>1]
filtro.VarForward.loc[filtro.VarForward > 0].count() / filtro.VarForward.count()
0.5509325681492109
```

```

# Rta Ejercicio 8      #
#                      #
#-----#
data = pd.read_excel('AAPL.xlsx')
data = data[::-1]
data['precioMedio'] = (data.open + data.close + data.high + data.low)/4
data['volumenMln'] = round(data.volume * data.precioMedio / 1000000)
res = data.filter(['timestamp','precioMedio','volumenMln'],axis=1)
res.set_index('timestamp' , inplace=True)
res['SMA20 Vol'] = res.volumenMln.rolling(20).mean()
res.dropna()

```

	precioMedio	volumenMln	SMA20 Vol
timestamp			
2000-03-31	131.6250	476.0000	402.7500
2000-04-03	134.4375	394.0000	410.5500
2000-04-04	127.4225	751.0000	432.9500
.....			
2020-03-04	298.9275	16,380.0000	13,432.3500
2020-03-05	294.8500	13,826.0000	13,645.1000
2020-03-06	285.7700	16,159.0000	14,027.0000

```

#-----#
# Rta Ejercicio 9      #
#-----#
data = pd.read_excel('AAPL.xlsx')
data = data[::-1]
data['SMA50'] = data.adjusted_close.rolling(50).mean()
data['SMA200'] = data.adjusted_close.rolling(200).mean()
data['VarFw'] = (data.adjusted_close.shift(-2)/ data.adjusted_close -1)*100

res = data.filter(['timestamp','adjusted_close','SMA50','SMA200','VarFw'],axis=1)
res.set_index('timestamp' , inplace=True)
res.dropna().head()

```

	adjusted_close	SMA 50	SMA 200	Var Forward 20r
timestamp				
2000-12-15	0.8716	1.1646	2.8047	-0.4245
2000-12-18	0.8834	1.1548	2.7897	0.9169
2000-12-19	0.8679	1.1452	2.7750	0.4263
2000-12-20	0.8915	1.1371	2.7605	4.3073
2000-12-21	0.8716	1.1302	2.7459	4.4860

```

#-----#
# Rta Ejercicio 10 #
#-----#

data = pd.read_excel('AAPL.xlsx')
data = data[::-1]
data['SMA50'] = data.adjusted_close.rolling(50).mean()
data['SMA200'] = data.adjusted_close.rolling(200).mean()
data['VarFw']=(data.adjusted_close.shift(-50)/ data.adjusted_close-1)*100
res = data.filter(['timestamp', 'adjusted_close', 'SMA50', 'SMA200', 'VarFw'], axis=1)
res = res.set_index('timestamp').dropna()
resFiltro = res.loc[ res['SMA50'] > res['SMA200'] ].copy()
resFiltro['Condicion'] = resFiltro['VarFw'] > 0
positivos = resFiltro['Condicion'].sum()
totales = resFiltro['Condicion'].count()

positivos / totales * 100

```

71.24753590537877

```

#-----#
# Rta Ejercicio 11 #
#-----#

data = pd.read_excel('AAPL.xlsx')
data = data[::-1]
fast = 10
slow = 20
fastKey = 'SMA '+str(fast)
slowKey = 'SMA '+str(slow)
data[fastKey] = data.adjusted_close.ewm(span=fast, adjust=False).mean()
data[slowKey] = data.adjusted_close.ewm(span=slow, adjust=False).mean()
data['cruce'] = (data[fastKey] / data[slowKey] - 1) *100
res = data.filter(['timestamp', 'adjusted_close', slowKey, fastKey, 'cruce'], axis=1)
res.set_index('timestamp').dropna()

```

	adjusted_close	SMA 20	SMA 10	cruce
timestamp				
2000-03-06	3.8960	3.8960	3.8960	0.0000
2000-03-07	3.8086	3.8877	3.8801	-0.1946
2000-03-08	3.7816	3.8776	3.8622	-0.3965
2000-03-09	3.7894	3.8692	3.8490	-0.5224
...
2020-03-03	289.3200	302.5751	295.3992	-2.3716
2020-03-04	302.7400	302.5908	296.7339	-1.9356
2020-03-05	292.9200	301.6698	296.0404	-1.8661
2020-03-06	289.0300	300.4660	294.7658	-1.8971

```
#-----#
# Rta Ejercicio 12a #
#-----#
#Volvemos a poner precision de 4 flotantes porque había quedado en 2 de antes
pd.options.display.float_format = '{:.4f}'.format

data = pd.read_excel('AAPL.xlsx')
data = data[::-1]
data = data.reset_index(drop=True)

for i in range(len(data)):
    data.loc[i,'maximo_hist']=data.loc[0:i,'adjusted_close'].max()

data = data.drop(["open","high","low","close","volume"],axis=1)
data.set_index("timestamp", inplace=True)
data.head(8)
```

timestamp	adjusted_close	maximo_hist
2000-03-06	3.8960	3.8960
2000-03-07	3.8086	3.8960
2000-03-08	3.7816	3.8960
2000-03-09	3.7894	3.8960
2000-03-10	3.8979	3.8979
2000-03-13	3.7602	3.8979
2000-03-14	3.5414	3.8979
2000-03-15	3.6034	3.8979

```
#-----#
# Rta Ejercicio 12b #
#-----#
data = pd.read_excel('AAPL.xlsx')
data = data[::-1]
data['maximo_h']=data.adjusted_close.cummax()
data = data.drop(["open","high","low","close","volume"],axis=1)
data.set_index("timestamp", inplace=True)
data.head(8)
```

	adjusted_close	maximo_h
timestamp		
2000-03-06	3.8960	3.8960
2000-03-07	3.8086	3.8960
2000-03-08	3.7816	3.8960
2000-03-09	3.7894	3.8960
2000-03-10	3.8979	3.8979
2000-03-13	3.7602	3.8979
2000-03-14	3.5414	3.8979
2000-03-15	3.6034	3.8979

```
#-----#
# Rta Ejercicio 13-a #
#-----#
data = pd.read_excel('AAPL.xlsx')
data = data[::-1]

res = data.filter(['timestamp','adjusted_close'],axis=1)
res = res.set_index('timestamp').dropna()
resFiltro = res.loc[(res.index>='2008-01-01')&(res.index<='2008-12-31')].copy()
resFiltro['factor'] = resFiltro.adjusted_close /
resFiltro.adjusted_close.shift(1)
resFiltro['Yield'] = (resFiltro['factor'].cumprod() -1)*100
resFiltro.tail().round(2)
```

	adjusted_close	factor	Yield
timestamp			
2008-12-24	10.54	0.98	-56.35
2008-12-26	10.64	1.01	-55.96
2008-12-29	10.74	1.01	-55.55
2008-12-30	10.70	1.00	-55.71
2008-12-31	10.58	0.99	-56.19

```
#-----#
# Rta Ejercicio 13-b #
#-----#

data = pd.read_excel('AAPL.xlsx')
data = data[::-1]

res = data.filter(['timestamp','adjusted_close'],axis=1)
res = res.set_index('timestamp').dropna()
resFiltro = res.loc[ (res.index >= '2008-01-01') &
                    (res.index <= '2008-12-31' ) ].copy()

resFiltro['factor'] = resFiltro.adjusted_close/resFiltro.adjusted_close.shift(1)
resFiltro['Yield'] = (resFiltro['factor'].cumprod() -1)*100

float(resFiltro[-1:]['Yield'])
```

-56.194686602256816

```
#-----#
# Rta Ejercicio 13-c #
#-----#

data = pd.read_excel('AAPL.xlsx')
data = data[::-1]

año = 2008
desde = pd.Timestamp(año, 1, 1)
hasta = pd.Timestamp(año, 12, 31)

res = data.filter(['timestamp','adjusted_close'],axis=1)
res = res.set_index('timestamp').dropna()
resFiltro = res.loc[ (res.index >= desde) & (res.index <= hasta) ].copy()
resFiltro['factor'] = resFiltro.adjusted_close/resFiltro.adjusted_close.shift(1)
resFiltro['Yield'] = (resFiltro['factor'].cumprod() -1)*100

float(resFiltro[-1:]['Yield'])
```

-56.194686602256816

```

#-----#
# Rta Ejercicio 14 #
#-----#

import datetime, calendar
data = pd.read_excel('AAPL.xlsx')
data = data[::-1]

años = []
for i in range(2005,2016):
    años.append(i)

rendimientos = pd.DataFrame(index=años)
for año in años:
    desde = pd.Timestamp(año-1, 12, 31)
    hasta = pd.Timestamp(año+1, 1, 1)

    if desde.weekday()==6:
        desde = pd.Timestamp(año-1, 12, 29)
    elif desde.weekday()==5:
        desde = pd.Timestamp(año-1, 12, 30)

    res = data.filter(['timestamp','adjusted_close'],axis=1)
    res = res.set_index('timestamp').dropna()
    res = res.loc[ (res.index >= desde) & (res.index <= hasta) ].copy()
    resFiltro = res['adjusted_close']/resFiltro['adjusted_close'].shift(1)
    resFiltro['factor'] = resFiltro['adjusted_close'].cumprod() -1)*100
    resFiltro['Yield'] = (resFiltro['factor'].cumprod() -1)*100
    rend = float(resFiltro[-1]['Yield'])
    rendimientos.loc[año,"Yield"] = round(rend,2)

rendimientos

```

	Yield
2005	123.26
2006	18.01
2007	133.48
2008	-56.91
2009	146.90
2010	53.07
2011	25.56
2012	32.59
2013	8.06
2014	40.62
2015	-3.02

Claramente esta no es la forma más práctica de obtener los rendimientos anualizados, pero viene bien el ejemplo para luego, hacerlo de forma mucho más práctica y sencilla en la próxima unidad donde veamos resampleo y agrupamiento de datos por fechas y otros agrupamientos

Agrupamiento y Resampleo

Agrupamiento - GroupBy

Empecemos con un agrupando por timestamps es decir por criterios de fechas

Para ello como la serie que vamos a utilizar que es la de AAPL.xlsx que venimos utilizando en capítulos previos, viene ordenada al revés lo primero que haremos es ordenarla de antiguos a recientes

También vamos a importar la librería datetime que la hemos visto en unidades anteriores para trabajar con objetos de fechas

```
import pandas as pd
import datetime as dt

data = pd.read_excel('AAPL.xlsx')
data = data.sort_values('timestamp', ascending=True).set_index("timestamp")
data.head()
```

	open	high	low	close	adjusted_close	volume
timestamp						
2000-03-06	126.00	129.13	125.00	125.69	3.8960	1880000
2000-03-07	126.44	127.44	121.12	122.87	3.8086	2437600
2000-03-08	122.87	123.94	118.56	122.00	3.7816	2421700
2000-03-09	120.87	125.00	118.25	122.25	3.7894	2470700
2000-03-10	121.69	127.94	121.00	125.75	3.8979	2219700

Bien, solo era para reapasararlo, ahora ni bien tenemos la serie de datos ordenada ascendente por fecha, lo que vamos a hacer es agruparla por año

Si solo la agrupo, lo que me va a devolver pandas es un objeto agrupado, no datos, por lo que si quiero algún dato útil tengo que pasarle alguna función a ese objeto de datos agrupados, vamos a arrancar usando la función count () que obviamente lo que hace es contar los objetos

Veamos el ejemplo agrupando entonces por año y simplemente contando la cantidad de datos de cada año

```
data = pd.read_excel('AAPL.xlsx')
data = data.sort_values('timestamp', ascending=True)
agrupados = data.adjusted_close.groupby(data.timestamp.dt.year).count().to_frame()
agrupados.index.name = "Año"
agrupados.columns = ['Cantidad de Ruedas']
agrupados.head()
```

Cantidad de Ruedas	
Año	
2000	209
2001	248
2002	252
2003	252
2004	252

La función `to_frame()` utilizada es para que la respuesta sea un objeto DataFrame estándar (esto me permite por ejemplo cambiarle los nombres al índice y columnas fácilmente como vimos en el ejemplo), ya que si no la uso, me va a devolver la tabla de datos pero no como un DataFrame

Son detalles pero que en algún momento si quieren reutilizar esta tabla para alguna función de pandas que hayamos visto, si no lo tienen como objeto tipo DataFrame les va a tirar un error, así que ya que estamos les dejo la función porque va a ser útil más adelante

Así como usamos la función `count()` podemos usar otro tipo de funciones, por ejemplo:

- `first()`
- `last()`
- `min()`
- `max()`

Vamos a ver a continuación un ejemplo práctico de cómo aplicar estas agrupaciones todas juntas en un mismo DataFrame

```

data = pd.read_excel('AAPL.xlsx')
data = data.sort_values('timestamp', ascending=True)

agrupados = pd.DataFrame()

agrupados['first'] = data.adjusted_close.groupby(data.timestamp.dt.year).first()
agrupados['last'] = data.adjusted_close.groupby(data.timestamp.dt.year).last()
agrupados['max'] = data.adjusted_close.groupby(data.timestamp.dt.year).max()
agrupados['min'] = data.adjusted_close.groupby(data.timestamp.dt.year).min()

agrupados.round(2).head(8)

```

	first	last	max	min
timestamp				
2000	3.90	0.92	4.47	0.87
2001	0.92	1.36	1.65	0.92
2002	1.44	0.89	1.62	0.84
2003	0.92	1.32	1.54	0.81
2004	1.32	3.99	4.24	1.32
2005	3.92	8.91	9.30	3.92
2006	9.27	10.52	11.38	6.28
2007	10.39	24.56	24.78	10.32

También podemos aplicar sobre los datos agrupados otro tipo de funciones más matemáticas o estadísticas como por ejemplo:

- count()
- sum()
- prod()
- std()
- median()
- skew()
- quantile()

No se preocupen por si no entienden que significa alguna de ellas, las veremos en detalle en la siguiente unidad

Entonces agrupemos por año calculando estos parámetros para cada año

```
data = pd.read_excel('AAPL.xlsx')
data = data.sort_values('timestamp', ascending=True)

agrupados = pd.DataFrame()

agrupados['count'] = data.adjusted_close.groupby(data.timestamp.dt.year).count()
agrupados['sum'] = data.adjusted_close.groupby(data.timestamp.dt.year).sum()
agrupados['mean'] = data.adjusted_close.groupby(data.timestamp.dt.year).mean()
agrupados['std'] = data.adjusted_close.groupby(data.timestamp.dt.year).std()
agrupados['median'] = data.adjusted_close.groupby(data.timestamp.dt.year).median()
agrupados['skew'] = data.adjusted_close.groupby(data.timestamp.dt.year).skew()
agrupados['q_01pct']=data.adjusted_close.groupby(data.timestamp.dt.year).quantile(0.01)

agrupados.round(2).head(10)
```

	count	sum	mean	std	median	skew	q_01pct
timestamp							
2000	209	569.06	2.72	1.11	3.12	-0.54	0.87
2001	248	310.86	1.25	0.15	1.24	0.20	0.94
2002	252	299.01	1.19	0.27	1.07	0.19	0.86
2003	252	289.69	1.15	0.21	1.20	-0.16	0.82
2004	252	555.02	2.20	0.82	1.91	1.21	1.37
2005	252	1458.39	5.79	1.42	5.32	1.10	4.00
2006	251	2203.73	8.78	1.18	8.58	0.34	6.53
2007	251	3992.03	15.90	4.37	15.36	0.44	10.41
2008	253	4453.73	17.60	4.18	18.40	-0.28	10.56
2009	252	4587.25	18.20	4.94	17.65	-0.04	10.29

Cálculos antes de agrupar

Obviamente como se habrán dado cuenta, la potencia de pandas empieza a tomar color cuando combinamos con el agrupamiento todas las funciones que vimos en la unidad anterior, funciones de rolling, cumulative, y demás joyitas

Empecemos con algo bien sencillito, calculemos ahora el rendimiento medio diario y agrupemos esto, cada año

```
data = pd.read_excel('AAPL.xlsx')
data = data.sort_values(by='timestamp', ascending=True)
data['pct_change'] = data.adjusted_close.pct_change()
agrupados = data['pct_change'].groupby(data['timestamp'].dt.year).mean()*100
agrupados.index.name = "Año"
agrupados.head()
```

Año	Rendimiento
2000	-0.497082
2001	0.232789
2002	-0.121224
2003	0.185294
2004	0.470234

Name: pct_change, dtype: float64

Fíjense que en el caso anterior no lo convertimos en un DataFrame y me lo muestra como una serie en lugar de una tabla

Calculemos ahora el rendimiento compuesto de cada año

Para ello podríamos intentar sumar los rendimientos diarios de cada año en lugar de contar, hay una función que es sum() que los sumaría y listo, pero no estaríamos calculando un rendimiento compuesto, sino la suma de los rendimientos porcentuales que no es lo mismo.. ¿Entonces?

Hay otra función que es prod() que calcula el producto de todos los valores de una serie (los del mismo año en nuestro caso), pero para que ello sea un rendimiento compuesto hay que sumarle 1 a cada rendimiento diario, y al final de todo restarle 1 y multiplicarlo por 100 (no nos vamos a detener en este cálculo, es un cálculo sencillo de matemática financiera básica)

```
import datetime as dt
data = pd.read_excel('AAPL.xlsx')
data = data.sort_values(by='timestamp', ascending=True)
data['change_pct'] = data['adjusted_close'].pct_change()+1
agrupados = data.change_pct.groupby(data['timestamp'].dt.year).prod().to_frame()
agrupados.index.name = "Año"
agrupados.columns = ['Rendimiento']
agrupados['Rendimiento'] = (agrupados['Rendimiento'] - 1)*100
agrupados.head()
```

Rendimiento**Año**

2000	-76.321869
2001	47.176152
2002	-34.565810
2003	49.122017
2004	201.358696

¿Y si quisieramos calcular el rendimiento de cada mes?

¿Solo hay que remplazar dt.year por dt.month?

```
data = pd.read_excel('AAPL.xlsx')
data = data.sort_values(by='timestamp', ascending=True)
data['change_pct'] = data['adjusted_close'].pct_change() + 1

agrupados = data.change_pct.groupby(data['timestamp'].dt.month).prod().to_frame()
agrupados.columns = ['Rendimiento']
agrupados['Rendimiento'] = (agrupados['Rendimiento'] - 1) * 100
agrupados.head(12)
```

Rendimiento**timestamp**

1	48.177514
2	31.275957
3	206.983707
4	68.965965
5	18.061155
6	-4.737532
7	111.091820
8	146.512050
9	-55.851952
10	191.323687
11	34.496522
12	-27.371822

Obviamente lo que calculamos ahí es el rendimiento compuesto de cada mes en forma independiente del año en el que el mes se encuentra, no tiene mucho sentido práctico verdad? bueno quizá nos sirva como idea de estacionalidad pero vayamos a calcular lo que en realidad queremos que es una serie con los rendimientos de cada mes

Para ello en lugar de decirle que agrupe por una sola variable (el mes) le vamos a decir que agrupe por una lista de variables

Como nos vamos a referir a una lista, la vamos a poner entre [], y ahí le decimos que el primer criterio sea el año y el segundo el mes

```
data = pd.read_excel('AAPL.xlsx')
data = data.sort_values(by='timestamp', ascending=True)
data['change_pct'] = data.adjusted_close.pct_change() + 1

agrupados = data.change_pct.groupby([
    data['timestamp'].dt.year,
    data['timestamp'].dt.month
]).prod().to_frame()

agrupados.columns = ['Rendimiento']
agrupados['Rendimiento'] = (agrupados['Rendimiento'] - 1) * 100
agrupados.tail(8)
```

Rendimiento		
	timestamp	timestamp
2019	8	-1.643035
	9	7.296168
	10	11.068442
	11	7.751734
	12	9.878417
	2020	5.400976
2020	1	-11.467252
	3	5.732368

¿Y si queremos ver cual es el mejor dia de la semana?

```
data = pd.read_excel('AAPL.xlsx')
data = data.sort_values(by='timestamp', ascending=True)
data['change_pct'] = data['adjusted_close'].pct_change() * 100

agrupados = data.change_pct.groupby(data['timestamp'].dt.dayofweek).mean().to_frame()
agrupados.columns = ['Rendimiento']
agrupados.index.name = "Dia de Semana"

dias = ["Lunes", "Martes", "Miercoles", "Jueves", "Viernes"]
for i in range(len(agrupados)):
    agrupados.loc[i, "Dia Nombre"] = dias[i]
agrupados
```

Rendimiento Dia Nombre

Dia de Semana

	Rendimiento	Dia Nombre
0	0.316007	Lunes
1	0.092570	Martes
2	0.158719	Miercoles
3	0.171399	Jueves
4	-0.132945	Viernes

O Si recuerdan la librería Calendar, podríamos usar:

```
import locale, calendar
locale.setlocale(locale.LC_TIME, "esp")

data = pd.read_excel('AAPL.xlsx')
data = data.sort_values(by='timestamp', ascending=True)
data['change_pct'] = data['adjusted_close'].pct_change()*100

agrupados = data.change_pct.groupby(data['timestamp'].dt.dayofweek).mean().to_frame()
agrupados.columns = ['Rendimiento']
agrupados.index.name="Dia de Semana"

for i in range(len(agrupados)):
    agrupados.loc[i,"Dia Nombre"] = calendar.day_name[i]
agrupados
```

Rendimiento Dia Nombre

Dia de Semana

	Rendimiento	Dia Nombre
0	0.316007	lunes
1	0.092570	martes
2	0.158719	miércoles
3	0.171399	jueves
4	-0.132945	viernes

Bueno, como se imaginarán las posibilidades que se abren son infinitas, podemos usar todas las funciones de pandas, tanto las de rolling, las acumulativas, las estadísticas y combinarlas con agrupamientos y filtros, con lo cual en 5 líneas de código nos permite prácticamente sacar cualquier tipo de reporte del comportamiento estadístico de una serie

Recordemos el siguiente ejercicio de la unidad anterior: 14- Usando el código del ejercicio "13c", generar una lista con los años desde 2005 a 2015 y devolver un DataFrame con los rendimientos acumulados (tasa compuesta diaria) año por año

Su resolución sin usar agrupamientos era algo como lo siguiente:

```
import datetime, calendar

data = pd.read_excel('AAPL.xlsx')
data = data[::-1]

años = []

for i in range(2005,2016):
    años.append(i)

rendimientos = pd.DataFrame(index=años)

for año in años:
    desde = pd.Timestamp(año-1, 12, 31)
    hasta = pd.Timestamp(año+1, 1, 1)

    if desde.weekday()==6:
        desde = pd.Timestamp(año-1, 12, 29)
    elif desde.weekday()==5:
        desde = pd.Timestamp(año-1, 12, 30)

    res = data.filter(['timestamp','adjusted_close'],axis=1)
    res = res.set_index('timestamp').dropna()
    resFiltro = res.loc[ (res.index >= desde) & (res.index <= hasta) ].copy()
    resFiltro['factor'] = resFiltro.adjusted_close/resFiltro.adjusted_close.shift(1)
    resFiltro['Yield'] = (resFiltro['factor'].cumprod() -1)*100
    rend = float(resFiltro[-1:]['Yield'])
    rendimientos.loc[año,"Yield"] = round(rend,2)

rendimientos
```

	Yield
2005	123.26
2006	18.01
2007	133.48
2008	-56.91
2009	146.90
2010	53.07
2011	25.56
2012	32.59
2013	8.06
2014	40.62
2015	-3.02

Ahora veamos cómo se simplifica esto pudiendo usar agrupamientos:

```
data = pd.read_excel('AAPL.xlsx')
data = data.sort_values('timestamp', ascending=True)
data['factor']=data.adjusted_close.pct_change()+1

yields = pd.DataFrame()
yields['Yield'] = (data.factor.groupby(data.timestamp.dt.year).prod() -1)*100
yields.round(2).loc[2005:2015]
```

Yield	
timestamp	
2005	123.26
2006	18.01
2007	133.48
2008	-56.91
2009	146.90
2010	53.07
2011	25.66
2012	32.59
2013	8.06
2014	40.62
2015	-3.02

¿y.. qué me contás? ¿vale la pena o no vale la pena ir aprendiendo herramientas nuevas?

Agrupamiento y agrupamiento combinado para conteo de un booleano

Vamos a ver el tema del agrupamiento y conteo con alguna variable Booleana ya que no vamos a contar una variable continua porque no tiene sentido

Para ello vamos a crear una tabla con las columnas "gap_positivo" y "vela_verde" que sean True o False según el gap sea positivo en el primer caso y cuando la vela sea verde en el segundo

```
data = pd.read_excel('AAPL.xlsx')
data = data.sort_values('timestamp', ascending=True)

tabla = pd.DataFrame()
data['cierre_previo']=data['close'].shift()
data['gap_positivo']=data.eval('open>cierre_previo')
data['vela_verde']=data.eval("close>open")
data.drop(['high','low','adjusted_close','volume'],axis=1,inplace=True)
data
```

	timestamp	open	close	cierra_previo	gap_positivo	vela_verde
5032	2000-03-06	126.00	125.69	NaN	False	False
5031	2000-03-07	126.44	122.87	125.69	True	False
5030	2000-03-08	122.87	122.00	122.87	False	False
5029	2000-03-09	120.87	122.25	122.00	False	True
5028	2000-03-10	121.69	125.75	122.25	False	True
...
4	2020-03-02	282.28	298.81	273.36	True	True
3	2020-03-03	303.67	289.32	298.81	True	False
2	2020-03-04	296.44	302.74	289.32	True	True
1	2020-03-05	295.52	292.92	302.74	False	False
0	2020-03-06	282.00	289.03	292.92	False	True

5033 rows × 6 columns

Ahora que ya tenemos esta tabla, vamos a contabilizar los datos de estas dos columnas. Vemos primero el caso de contabilizar una columna por separado:

```
data.groupby('gap_positivo').size().reset_index(name='total')
```

	gap_positivo	total
0	False	2217
1	True	2816

Bien, ahora, si en el argumento de `groupby()` le pasamos una lista de columnas, en lugar de una sola columna, tenemos:

```
data.groupby(['gap_positivo', 'vela_verde']).size().reset_index(name='total')
```

	gap_positivo	vela_verde	total
0	False	False	1063
1	False	True	1154
2	True	False	1419
3	True	True	1397

Otros tipos de agrupamiento por timestamps

Otros tipos de agrupamiento pueden ser por

- week (nro de semana del año, 1 a 53)
- dayofweek (0 lunes, 1 martes... 6 domingo)
- day (número de día del mes)
- month (número de mes del año)

En el siguiente ejemplo obtenemos una lista con los rendimientos medios según semana del año, este tipo de listas nos puede servir para ver si el activo tiene estacionalidad comparando correlaciones de esta serie con un índice de referencia

```
data = pd.read_excel('AAPL.xlsx')
data = data.sort_values(by='timestamp', ascending=True)
data['change_pct'] = data['adjusted_close'].pct_change() * 100

agrupados=data.change_pct.groupby(data['timestamp'].dt.week).mean().to_frame().round(2)
agrupados.columns = ['Rendimiento']
list(agrupados.Rendimiento)

[0.41, 0.31, -0.29, -0.2, 0.32, 0.22, 0.43, -0.07, 0.1, 0.09, 0.39, 0.43, 0.11, 0.37,
-0.46, 0.37, 0.3, 0.21, 0.15, -0.13, 0.12, 0.11, -0.02, -0.27, 0.28, 0.01, 0.42,
0.23, 0.21, -0.05, 0.17, -0.13, 0.4, 0.33, 0.46, -0.18, 0.29, -0.01, -0.6, -0.33,
0.68, -0.02, 0.46, 0.56, -0.31, -0.09, 0.18, 0.61, -0.18, -0.28, -0.13, 0.29, -0.08]
```

Agrupamiento con filtros

Una de las cosas más interesantes luego de los filtros, las funciones rolling y acumulativas y los agrupamientos, es aplicar filtros posteriores a los agrupamientos, la potencia que brinda toda esta combinación es enorme.

Veamos ejemplos, empecemos por aplicar un filtro y luego agrupamiento.

Vamos a calcular la cantidad de veces por año que AAPL tuvo subas de más del 10% en una sola rueda

```
data = pd.read_excel('AAPL.xlsx')
data = data.sort_values(by='timestamp', ascending=True)
data['change_pct'] = data['adjusted_close'].pct_change()*100

data = data.loc[data['change_pct']>10] #Aca le agregamos el filtro

agrupados= data.change_pct.groupby(data['timestamp'].dt.year).count().to_frame()
agrupados.columns = ['Subas +10%']
agrupados
```

Subas +10%

timestamp	
2000	2
2001	5
2003	1
2004	3
2006	1
2007	1
2008	2

Ahora hagamos al revés, hagamos un agrupamiento y luego un filtro, veamos por ejemplo las semanas históricas en que el rendimiento fue mayor al 13%

```
data = pd.read_excel('AAPL.xlsx')
data = data.sort_values('timestamp', ascending=True)
data['factor']=data.adjusted_close.pct_change()+1

yields = pd.DataFrame()
yields['Yield'] = (data.factor.groupby(
                    [data.timestamp.dt.year, data.timestamp.dt.week]
                    ).prod() -1)*100
filtro = yields.loc[yields.Yield >13]
filtro
```

Yield		
timestamp	timestamp	
2000	25	13.369419
	34	13.620673
	44	19.885277
2001	3	13.437177
	12	17.232591
2003	19	26.646573
2004	42	16.485649
	48	17.001930
2006	29	19.834461
2011	41	14.115566
2018	18	13.251593

O bien otra herramienta también muy potente es combinar un agrupamiento con un posterior ordenamiento

Este es similar al ejemplo anterior, pero acá en lugar de buscar semanas que hayan rendido mas del "x%" buscamos el TOP10

```
data = pd.read_excel('AAPL.xlsx')
data = data.sort_values('timestamp', ascending=True)
data['factor']=data.adjusted_close.pct_change()+1

yields = pd.DataFrame()
yields['Yield'] = (data.factor.groupby([data.timestamp.dt.year,
data.timestamp.dt.week]).prod() -1)*100
ordenado = yields.sort_values("Yield", ascending=False)
ordenado.head(10)
```

Yield		
timestamp	timestamp	
2003	19	26.646573
2000	44	19.885277
2006	29	19.834461
2001	12	17.232591
2004	48	17.001930
	42	16.485649
2011	41	14.115566
2000	34	13.620673
2001	3	13.437177
2000	25	13.369419

Resampleo

Mediante el método resample podemos reagrupar rápidamente en función de diferentes timeframes una serie dada

Es importante aclarar que el índice de la tabla debe ser el timestamp

Las Frecuencias posibles son

- B = business day frequency
- D = calendar day frequency
- W = weekly frequency
- M = month end frequency
- BM = business month end frequency
- MS = month start frequency
- BMS = business month start frequency
- Q = quarter end frequency
- BQ = business quarter endfrequency
- QS = quarter start frequency
- BQS = business quarter start frequency
- A = year end frequency
- BA = business year end frequency
- AS = year start frequency
- BAS = business year start frequency
- BH = business hour frequency
- H = hourly frequency
- T = minutely frequency
- S = secondly frequency
- L = milliseconds

Cierres al último día del mes de cada año

```
data = pd.read_excel('AAPL.xlsx')
data = data.sort_values('timestamp', ascending=True)

#esta instrucción es para que el índice sea el
#timestamp data.set_index('timestamp', inplace=True)

data = data['adjusted_close'].resample('M').last().to_frame()
data.head()

adjusted_close
```

timestamp	adjusted_close
2000-03-31	4.2097
2000-04-30	3.8455
2000-05-31	2.6037
2000-06-30	3.2472
2000-07-31	3.1499

Cierres al último día HABIL del mes de cada año

vemos por ejemplo que en Abril de 2000, en lugar de tomar el día 30 toma el día 28 que fue viernes efectivamente, de todos modos el valor de la columna sigue siendo el mismo que es el del ultimo día del mes de abril de 2000, pero cambia el valor del índice resampleado

```
data = pd.read_excel('AAPL.xlsx')
data.set_index('timestamp', inplace=True)
data = data.sort_values('timestamp', ascending=True)

#Si usamos BM usa el ultimo dia habil del mes en el timestamp
data = data['adjusted_close'].resample('BM').last().to_frame()

data.head()
```

	adjusted_close
timestamp	
2000-03-31	4.2097
2000-04-28	3.8455
2000-05-31	2.6037
2000-06-30	3.2472
2000-07-31	3.1499

Cierres al primer día del mes de cada año

```
data = pd.read_excel('AAPL.xlsx')
data = data.sort_values('timestamp', ascending=True)

data.set_index('timestamp', inplace=True)
data = data['adjusted_close'].resample('MS').first().to_frame()
data.head()
```

	adjusted_close
timestamp	
2000-03-01	3.8960
2000-04-01	4.1322
2000-05-01	3.8532
2000-06-01	2.7625
2000-07-01	3.3049

Cierres cada bimestre

```
data = pd.read_excel('AAPL.xlsx')
data = data.sort_values('timestamp', ascending=True)

data.set_index('timestamp', inplace=True)
data = data['adjusted_close'].resample('2M').last().to_frame()
data
```

adjusted_close

timestamp

2000-03-31	4.2097
2000-05-31	2.6037
2000-07-31	3.1499
2000-09-30	1.5963
2000-11-30	1.0229
...	...
2019-07-31	211.0910
2019-09-30	222.7712
2019-11-30	266.6085
2020-01-31	308.7671
2020-03-31	289.0300

121 rows × 1 columns

Cierres cada lustro

```
data = pd.read_excel('AAPL.xlsx')
data = data.sort_values('timestamp', ascending=True)

data.set_index('timestamp', inplace=True)
data = data['adjusted_close'].resample('5A').last().to_frame()
data
```

adjusted_close

timestamp

2000-12-31	0.9225
2005-12-31	8.9135
2010-12-31	39.9936
2015-12-31	98.1154
2020-12-31	289.0300

Trabajando con intervalos de minutos

Para trabajar con intervalos menores a los que venimos trabajando vamos a usar el archivo AAPL_INTRAL.xlsx de la carpeta con la que venimos trabajando hasta ahora que pueden descargar en <http://bit.ly/39Am64T> (<http://bit.ly/39Am64T>)

Veamos primero que me levanta del archivo:

```
data = pd.read_excel('AAPL_INTRAL.xlsx')
data = data.sort_values('datetime', ascending=True)

data.set_index('datetime', inplace=True)
data.head(6)
```

	open	high	low	close	adjusted_close	volume
datetime						
2020-02-14 14:30:00	324.74	325.20	324.56	325.20	325.20	711919
2020-02-14 14:32:00	325.23	325.84	325.17	325.84	325.84	201604
2020-02-14 14:34:00	325.89	325.89	325.69	325.69	325.69	189701
2020-02-14 14:36:00	325.71	325.71	325.02	325.14	325.14	165750
2020-02-14 14:38:00	325.08	325.37	325.08	325.37	325.37	199297
2020-02-14 14:40:00	325.35	325.37	325.10	325.10	325.10	122597

Como vemos partimos de una serie cada 2 minutos, vamos a resamplearla cada 10 minutos:

```
data = pd.read_excel('AAPL_INTRAL.xlsx')
data = data.sort_values('datetime', ascending=True)

data.set_index('datetime', inplace=True)
data = data['adjusted_close'].resample('10T').first().to_frame()
data.head()
```

	adjusted_close
datetime	
2020-02-14 14:30:00	325.20
2020-02-14 14:40:00	325.10
2020-02-14 14:50:00	324.22
2020-02-14 15:00:00	324.20
2020-02-14 15:10:00	323.28

Acá la resampleamos cada 1 día

Ojo al poner first() tomamos el primer valor de día, obviamente si ponemos last() tomariamos el ultimo

Del mismo modo si ponemos mean() tomaríamos el valor medio

```
data = pd.read_excel('AAPL_INTRA.xlsx')
data = data.sort_values('datetime', ascending=True)

data.set_index('datetime', inplace=True)
data = data['adjusted_close'].resample('1D').first().to_frame()
data.columns = ["Apertura"]
data.head()
```

Apertura

	datetime
2020-02-14	325.20
2020-02-15	NaN
2020-02-16	NaN
2020-02-17	NaN
2020-02-18	315.06

Como ven aparecen días sin datos, claramente son feriados y/o sábados y domingos, para sacar esos datos, como ya vimos podemos usar la función dropna()

```
data = pd.read_excel('AAPL_INTRA.xlsx')
data = data.sort_values('datetime', ascending=True)

data.set_index('datetime', inplace=True)
data = data['adjusted_close'].resample('1D').last().to_frame().dropna()
data.columns = ["Cierre"]
data.head()
```

Cierre

	datetime
2020-02-14	324.95
2020-02-18	319.01
2020-02-19	323.62
2020-02-20	320.31
2020-02-21	313.03

Armando una tabla de varias columnas con un resampleo

Obvio que se puede armar una tabla de varias columnas "resampleadas", de más está decir que deben resamplearse con la misma frecuencia de tiempo

Veamos un ejemplo armando una tabla con Precio medio inicial y final por día partiendo de la tabla de precios cada 2 minutos

```
data = pd.read_excel('AAPL_INTRA.xlsx')
data = data.sort_values('datetime', ascending=True)
data.set_index('datetime', inplace=True)

tabla = data['adjusted_close'].resample('D').mean().to_frame().dropna()
tabla['first'] =
data['adjusted_close'].resample('D').first().to_frame().dropna()
tabla['last'] =
data['adjusted_close'].resample('D').last().to_frame().dropna()
tabla.columns = ["Precio Medio", "Precio Apertura", "Precio Cierre"]

tabla.round(2).head()
```

	Precio Medio	Precio Apertura	Precio Cierre
datetime			
2020-02-14	324.54	325.20	324.95
2020-02-18	317.64	315.06	319.01
2020-02-19	323.67	320.48	323.62
2020-02-20	321.01	324.02	320.31
2020-02-21	314.75	320.07	313.03

Material y Archivos Usados

- En esta unidad Usamos: AAPL.csv , AAPL.xlsx , AAPL_INTRA.xlsx
<http://bit.ly/39Am64T> (<http://bit.ly/39Am64T>)

Ejercicios

- 1 - Mostrar los años que AAPL tuvo rendimiento negativo y sus rendimientos en una tabla
- 2 - Armar una tabla con los meses y años en que el rendimiento del mes haya superado el 20%
- 3 - Armar un top 10 de los meses de mayor rendimiento usando group by sacando una tabla con el mes y año y rendimiento
- 4 - Dado el archivo intradiario de precios de AAPL calcular el precio medio resampleando por día
- 5 - Dado el archivo intradiario de precios de AAPL calcular el volumen total resampleando por semana
- 6 - Dado el archivo intradiario de precios de AAPL calcular el precio medio y el volumen total en nominales, resampleando por día
Una vez calculado, calcular en la misma tabla el volumen negociado a precio medio (en millones de USD)
- 7- Realizar el mismo cálculo del monto negociado por día pero esta vez calcular el monto negociado a cada 2 minutos y luego resamplear sumando los montos negociados
Comparar los resultados con los del ejercicio anterior ¿da lo mismo? ¿por qué? ¿cuál es el correcto?
- 8 - Dado el archivo intradiario de precios de AAPL calcular:
 - a - el precio medio, mínimo y máximo de cada día
 - b - luego calcular la oportunidad de trading intradiario como el rango porcentual de movimiento (max-min) sobre el medio
 - c - Contar la cantidad de días, de los 60 de la muestra, que ese rango fue mayor al 5%
 - d - Mostrar los 5 días de mayor rango porcentual de movimiento

Respuestas

```
#-----#
# Rta Ejercicio 1 #
#-----#
data = pd.read_excel('AAPL.xlsx')
data = data.sort_values(by='timestamp', ascending=True)
data['change_pct'] = data['adjusted_close'].pct_change() + 1

agrupados = ((data.change_pct.groupby(data['timestamp'].dt.year).prod() - 1) * 100).to_frame().round(2)
agrupados.columns = ['Rendimiento']
agrupados.index.name = "Año"
agrupados = agrupados.loc[agrupados.Rendimiento < 0] agrupados
```

	Rendimiento
	Año
2000	-76.32
2002	-34.57
2008	-56.91
2015	-3.02
2018	-5.39
2020	-1.34

```
#-----#
# Rta Ejercicio 2 #
#-----#
data = pd.read_excel('AAPL.xlsx')
data = data.sort_values(by='timestamp', ascending=True)
data['change_pct'] = data.adjusted_close.pct_change() + 1

agrupados = data.change_pct.groupby(
    [data['timestamp'].dt.year, data['timestamp'].dt.month]).prod().to_frame()
agrupados.columns = ['Rendimiento']
agrupados['Rendimiento'] = (agrupados['Rendimiento'] - 1) * 100
agrupados = agrupados.loc[agrupados['Rendimiento'] > 20]
agrupados
```

		Rendimiento
imestamp	timestamp	
2000	6	24.714829
2001	1	45.289973
	3	20.929821
	11	21.302590
2003	5	26.225045
2004	10	35.224576
	11	27.957519
2007	5	21.433651
	10	23.770259
2008	4	21.219291
2018	8	20.043477

```
#-----#
# Rta Ejercicio 3 #
#-----#
data = pd.read_excel('AAPL.xlsx')
data = data.sort_values('timestamp', ascending=True)
data['factor']=data.adjusted_close.pct_change()+1

yields = pd.DataFrame()
yields['Rendimiento'] = (data.factor.groupby([data.timestamp.dt.year, data.timestamp.dt.month]).prod() -1)*100
yields.index.names = ["Año","mes"]
ordenado = yields.sort_values("Rendimiento", ascending=False)
ordenado.head(10)
```

Rendimiento		
Año	mes	
2001	1	45.289973
2004	10	35.224576
	11	27.957519
2003	5	26.225045
2000	6	24.714829
2007	10	23.770259
	5	21.433651
2001	11	21.302590
2008	4	21.219291
2001	3	20.929821

```

#-----#
# Rta Ejercicio 4 #
#-----#
data = pd.read_excel('AAPL_INTRA.xlsx')
data = data.sort_values('datetime', ascending=True)
data.set_index('datetime', inplace=True)
data = data['adjusted_close'].resample('1D').mean().to_frame().dropna()
data.columns = ["Precio Medio"]
data.head()

```

Precio Medio

datetime	Precio Medio
2020-02-14	324.542256
2020-02-18	317.638718
2020-02-19	323.670872
2020-02-20	321.013487
2020-02-21	314.748359

```

#-----#
# Rta Ejercicio 5 #
#-----#
data = pd.read_excel('AAPL_INTRA.xlsx')
data = data.sort_values('datetime', ascending=True)
data.set_index('datetime', inplace=True)

data = data['volume'].resample('1W').sum().to_frame().dropna()
data = data.loc[data.volume>0]
data.columns = ["Vol Total"]

data.head()

```

Vol Total

datetime	Vol Total
2020-02-16	17553874
2020-02-23	107058254
2020-03-01	319054025
2020-03-08	300176373
2020-03-15	376865438

```

#-----#
# Rta Ejercicio 6 #
#-----#

data = pd.read_excel('AAPL_INTRA.xlsx')
data = data.sort_values('datetime', ascending=True)
data.set_index('datetime', inplace=True)

tabla = data['adjusted_close'].resample('1D').mean().to_frame().dropna()
tabla['Vol'] = data['volume'].resample('1D').sum().to_frame().dropna()

tabla.columns = ["Precio", "Vol"]
tabla['MontoNegociado'] = tabla.Precio * tabla.Vol /1000000
tabla.head().round(2)

```

	Precio	Vol	MontoNegociado
datetime			
2020-02-14	324.54	17553874	5696.97
2020-02-18	317.64	35075626	11141.38
2020-02-19	323.67	20197381	6537.30
2020-02-20	321.01	22793333	7316.97
2020-02-21	314.75	28991914	9125.16

```

#-----#
# Rta Ejercicio 7 #
#-----#

data = pd.read_excel('AAPL_INTRA.xlsx')
data = data.sort_values('datetime', ascending=True)
data.set_index('datetime', inplace=True)
data['monto'] = data.adjusted_close * data.volume / 1000000

tabla = data['monto'].resample('1D').sum().to_frame().dropna()
tabla = tabla.loc[tabla.monto > 0]
tabla.head().round(2)

```

Esa es la forma mas correcta de calcular el monto negociado por dia,
porque de la otra forma no estaba ponderando los precios
de cada vela de 2 minutos al volumen de la misma

	monto
datetime	
2020-02-14	5698.01
2020-02-18	11126.93
2020-02-19	6526.64
2020-02-20	7322.32
2020-02-21	9125.26

```

#-----#
# Rta Ejercicio 8a-b #
#-----#

data = pd.read_excel('AAPL_INTRA.xlsx')
data = data.sort_values('datetime', ascending=True)
data.set_index('datetime', inplace=True)

tabla = data['adjusted_close'].resample('1D').mean().to_frame().dropna()
tabla['min'] =
data['adjusted_close'].resample('1D').min().to_frame().dropna()
tabla['max'] =
data['adjusted_close'].resample('1D').max().to_frame().dropna()
tabla.columns = ["Medio", "Min", "Max"]

tabla['rangoP'] = (tabla.Max - tabla.Min)/tabla.Medio *
100
tabla.head().round(2)

```

	Medio	Min	Max	rangoP
datetime				
2020-02-14	324.54	322.99	325.84	0.88
2020-02-18	317.64	315.06	319.63	1.44
2020-02-19	323.67	320.48	324.54	1.25
2020-02-20	321.01	318.73	324.38	1.76
2020-02-21	314.75	310.84	320.07	2.93

```

#-----#
# Rta Ejercicio 8c #
#-----#

data = pd.read_excel('AAPL_INTRA.xlsx')
data = data.sort_values('datetime', ascending=True)
data.set_index('datetime', inplace=True)

tabla = data['adjusted_close'].resample('1D').mean().to_frame().dropna()
tabla['min'] =
data['adjusted_close'].resample('1D').min().to_frame().dropna()
tabla['max'] =
data['adjusted_close'].resample('1D').max().to_frame().dropna()
tabla.columns = ["Medio", "Min", "Max"]

tabla['rangoP'] = (tabla.Max - tabla.Min)/tabla.Medio * 100

tabla.loc[tabla.rangoP>5].rangoP.count()

```

```

#-----#
# Rta Ejercicio 8d #
#-----#

data = pd.read_excel('AAPL_INTRA.xlsx')
data = data.sort_values('datetime', ascending=True)
data.set_index('datetime', inplace=True)

tabla = data['adjusted_close'].resample('1D').mean().to_frame().dropna()
tabla['min'] =
data['adjusted_close'].resample('1D').min().to_frame().dropna()
tabla['max'] =
data['adjusted_close'].resample('1D').max().to_frame().dropna()
tabla.columns = ["Medio", "Min", "Max"]

tabla['rangoP'] = (tabla.Max - tabla.Min)/tabla.Medio * 100

top5 = tabla.sort_values("rangoP", ascending=False).head()
top5

```

	Medio	Min	Max	rangoP
datetime				
2020-03-20	246.418718	228.02	251.77	9.638067
2020-03-13	260.156073	253.10	278.06	9.594241
2020-03-12	255.731755	248.00	269.90	8.563661
2020-02-28	267.740667	257.69	278.03	7.596903
2020-03-16	252.888519	240.00	258.49	7.311522

Funciones Estadísticas de Pandas

Estadísticas básicas de la Tabla - Describe()

Esto nos devuelve la media, el desvió estándar, el valor mínimo y máximo, y los valores de los 4 cuartiles de cada columna de la tabla, es realmente muy útil para tener un vistazo rápido de los datos

```
import pandas as pd
pd.options.display.precision = 3
data = pd.read_excel('AAPL.xlsx')
data.describe()
```

	open	high	low	close	adjusted_close	volume
count	5033.000	5033.000	5033.000	5033.000	5033.000	5.033e+03
mean	174.087	175.947	172.059	174.050	58.376	2.369e+07
std	161.042	162.345	159.498	160.947	67.394	1.813e+07
min	12.990	13.190	12.720	13.120	0.813	7.025e+05
25%	55.480	56.940	54.530	55.650	4.585	1.009e+07
50%	122.600	124.750	120.810	122.420	27.556	2.035e+07
75%	213.130	214.500	210.320	212.460	94.127	3.204e+07
max	702.410	705.070	699.570	702.100	327.200	1.896e+08

También puedo localizar un valor cualquiera de toda la tabla describe()

```
estadisticas = data.describe().round()
estadisticas.loc['mean']['volume']
```

23694204.0

Valor Absoluto de una columna

```
data = pd.read_excel('AAPL.xlsx')
data = data.sort_values(by='timestamp', ascending=True)
data['change_pct'] = data['adjusted_close'].pct_change()*100
data['change_pct_abs'] = data.change_pct.abs()
data.drop(['open', 'high', 'low', 'close'], axis=1, inplace=True)
data.head()
```

	timestamp	adjusted_close	volume	change_pct	change_pct_abs
5032	2000-03-06	3.896	1880000	NaN	NaN
5031	2000-03-07	3.809	2437600	-2.243	2.243
5030	2000-03-08	3.782	2421700	-0.709	0.709
5029	2000-03-09	3.789	2470700	0.206	0.206
5028	2000-03-10	3.898	2219700	2.863	2.863

Concatenación

Antes de empezar a ver cuestiones más avanzadas de estadística vamos a ver primero como concatenamos diferentes tablas entre sí

Vamos a suponer que tenemos un data feed (en nuestro caso por ahora vamos a partir de archivos de Excel como fuente de datos hasta que veamos la unidad de APIs) de AAPL otro de SPY y otro de QQQ y queremos volcar en una sola tabla los rendimientos diarios de cada uno

Lo primero que vamos a hacer es con cada data feed armar una tabla con las columnas que necesitemos ya precalculadas, en nuestro caso solo las variaciones porcentuales diarias

```
aapl = pd.read_excel('AAPL.xlsx')
aapl = aapl.sort_values('timestamp', ascending=True)
aapl['change_pct'] = aapl['adjusted_close'].pct_change()*100
aapl.set_index('timestamp', inplace=True)

spy = pd.read_excel('SPY.xlsx')
spy = spy.sort_values('timestamp', ascending=True)
spy['change_pct'] = spy['adjusted_close'].pct_change()*100
spy.set_index('timestamp', inplace=True)

qqq = pd.read_excel('QQQ.xlsx')
qqq = qqq.sort_values('timestamp', ascending=True)
qqq['change_pct'] = qqq['adjusted_close'].pct_change()*100
qqq.set_index('timestamp', inplace=True)

#veamos una cualquiera:
qqq.head()
```

	open	high	low	close	adjusted_close	volume	change_pct
timestamp							
2000-03-06	223.5	226.4	220.5	223.9	98.245	11634200	NaN
2000-03-07	227.6	227.8	217.5	220.5	96.753	19735500	-1.519
2000-03-08	222.0	224.0	212.6	222.5	97.631	21756300	0.907
2000-03-09	222.4	231.0	218.2	230.0	100.921	15050700	3.371
2000-03-10	229.8	232.9	227.0	229.0	100.483	14875300	-0.435

Ahora que tenemos las variables: aapl, spy y qqq cada una con la tabla de datos correspondiente vamos a armar la concatenada

```
#Funcion concat:  
#axis=1 concatena horizontalmente  
#(es decir que busca coincidencia en el indice de la tabla y las intersecta)  
#axis=0 concatena verticalmente es decir pega una atras de otra  
tabla = pd.concat([aapl['change_pct'],spy['change_pct'],qqq['change_pct']],axis=1)  
tabla.columns=["AAPL","SPY","QQQ"]  
tabla.dropna().head()
```

	AAPL	SPY	QQQ
timestamp			
2000-03-07	-2.243326	-1.934273	-1.518552
2000-03-08	-0.708922	-0.125380	0.907052
2000-03-09	0.206262	2.922331	3.370767
2000-03-10	2.863250	-0.532405	-0.434793
2000-03-13	-3.532671	-1.092729	-2.620053

Como pueden ver es muy sencillo, ya tenemos de esta forma las 3 tablas previas juntadas en una sola

Obviamente hay infinidad de maneras diferentes de hacerlo, el principio usé lo mismo que veníamos haciendo y de ahí apliqué el concat(), pero veamos otras alternativas ya que de la forma anterior eran 15 líneas de código para armar la tabla final, pero imaginense que en lugar de 3 Activos tengo 10 o 100, se hace denso no?

Bueno, lo que hago en el siguiente ejemplo es armar la tabla exactamente de la misma forma que antes, pero llenando la lista de variaciones que uso en la función concat() adentro de un ciclo FOR, de esta manera lo que gano es que si necesito ahora 20 activos en lugar de 3, la única modificación es la primera línea agrego todos los activos que necesite y ya, no cambia nada del resto (obviamente tengo que tener los Excel en este caso)

Bueno, acá en lugar de 15 líneas usé solo 11, y si tuviera 10 activos, en lugar de 43 líneas serían 11, siempre serían 11 líneas con esta forma de escribir el código y además es más prolífico que la anterior

```

activos = ["AAPL","SPY","QQQ"]
listaVariaciones = []
for activo in activos:
    tabla = pd.read_excel(str(activo) + '.xlsx')
    tabla = tabla.sort_values('timestamp',ascending=True)
    tabla['change_pct'] = tabla['adjusted_close'].pct_change()*100
    tabla.set_index('timestamp',inplace=True)
    listaVariaciones.append(tabla['change_pct'])

tablaFinal = pd.concat(listaVariaciones, axis=1)
tablaFinal.columns=activos
tablaFinal.dropna().head()

```

	AAPL	SPY	QQQ
timestamp			
2000-03-07	-2.243326	-1.934273	-1.518552
2000-03-08	-0.708922	-0.125380	0.907052
2000-03-09	0.206262	2.922331	3.370767
2000-03-10	2.863250	-0.532405	-0.434793
2000-03-13	-3.532671	-1.092729	-2.620053

Bueno, ahora voy a usar algo sencillo pero peligroso, digo peligroso porque si bien en este ejemplo es válido y lo será en muchos casos, habrá casos en los que hay que tomar precauciones.. veamos

Voy a armar una tabla solo con las fechas y los precios ajustados de los 3 activos, y luego con esa tabla calculo las variaciones y listo

Cuál es la trampa? y claro, en este ejemplo los 3 arrancan y terminan en las mismas fechas pero si tuviera tablas de diferentes rangos de fechas este método podría hacer cosas inesperadas, veamos:

```

aapl = pd.read_excel('AAPL.xlsx')
spy = pd.read_excel('SPY.xlsx')
qqq = pd.read_excel('QQQ.xlsx')
todo = pd.concat([aapl['timestamp'],aapl['adjusted_close'],spy['adjusted_close'],qqq['adjusted_close']],axis=1)
todo = todo.set_index("timestamp").sort_values("timestamp")
res = todo.pct_change()*100
res.columns = ["AAPL","SPY","QQQ"]
res.dropna().head()

```

	AAPL	SPY	QQQ
timestamp			
2000-03-07	-2.243326	-1.934273	-1.518552
2000-03-08	-0.708922	-0.125380	0.907052
2000-03-09	0.206262	2.922331	3.370767
2000-03-10	2.863250	-0.532405	-0.434793
2000-03-13	-3.532671	-1.092729	-2.620053

Ok, ahora vamos a mejorar esto suponiendo que puede recibir decenas de activos, como en el ejemplo de resolución anterior usando un ciclo FOR:

```
activos = ["AAPL","SPY","QQQ"]
listaCierres = []
for activo in activos:
    data = pd.read_excel(str(activo) + '.xlsx')
    listaCierres.append(data['adjusted_close'])
    if activo == activos[0]:
        listaCierres.append(data['timestamp'])

todo = pd.concat(listaCierres, axis=1)
todo = todo.set_index("timestamp").sort_values("timestamp")
res = todo.pct_change()*100
res.columns = activos
res.dropna().head()
```

	AAPL	SPY	QQQ
timestamp			
2000-03-07	-2.243326	-1.934273	-1.518552
2000-03-08	-0.708922	-0.125380	0.907052
2000-03-09	0.206262	2.922331	3.370767
2000-03-10	2.863250	-0.532405	-0.434793
2000-03-13	-3.532671	-1.092729	-2.620053

En este último ejemplo vemos que aumentó la cantidad de líneas pero claro si ahora en lugar de 3 activos tuviéramos 100 este último código quedaría igual mientras que el anterior seguiría creciendo y quedaría hiper despropósito y "poco mantenible"

Guardamos la tabla con la que vamos a trabajar de ahora en mas

Bueno, como de ahora en más, vamos a trabajar mucho con esta tabla de rendimientos de estos 3 activos, lo que vamos a hacer es guardar dicha tabla en un EXCEL aparte así no tenemos que andar construyéndola cada vez que queramos guardar algo

Como ya vimos en el principio de este bloque para guardar un Excel usamos el método de pandas `to_excel()`

```
activos = ["AAPL","SPY","QQQ"]
listaVariaciones = []
for activo in activos:
    tabla = pd.read_excel(str(activo) + '.xlsx')
    tabla = tabla.sort_values('timestamp',ascending=True)
    tabla['change_pct'] = tabla['adjusted_close'].pct_change()*100
    tabla.set_index('timestamp',inplace=True)
    listaVariaciones.append(tabla['change_pct'])

tablaFinal = pd.concat(listaVariaciones, axis=1).dropna()
tablaFinal.columns=activos
tablaFinal.to_excel("AAPL_SPY_QQQ.xlsx")
```

Clip (Acotado)

Esta función "acota" los valores de una columna entre un mínimo y un máximo, es muy útil para descartar la incidencia de "outliers" o "datos aberrantes" o "valores de colas" o simplemente "errores" en una distribución

```
tabla = pd.read_excel('AAPL_SPY_QQQ.xlsx')
tabla.set_index("timestamp",inplace=True)
tabla.tail(8)
```

	AAPL	SPY	QQQ
timestamp			
2020-02-26	1.586365	-0.367823	0.515392
2020-02-27	-6.536819	-4.491172	-5.007391
2020-02-28	-0.058497	-0.420154	0.077806
2020-03-02	9.310067	4.330656	5.160350
2020-03-03	-3.175931	-2.863244	-3.206728
2020-03-04	4.638463	4.203304	4.172236
2020-03-05	-3.243707	-3.324171	-3.038218
2020-03-06	-1.328008	-1.653111	-1.687225

Supongamos que no queremos considerar los valores superiores a +8% o los inferiores a -8% y queremos "topearlos" en esos límites

```
tabla = pd.read_excel('AAPL_SPY_QQQ.xlsx')
tabla.set_index("timestamp", inplace=True)
tabla_acotada = tabla.clip(-8,8)
tabla_acotada.tail(8)
```

	AAPL	SPY	QQQ
timestamp			
2020-02-26	1.586365	-0.367823	0.515392
2020-02-27	-6.536819	-4.491172	-5.007391
2020-02-28	-0.058497	-0.420154	0.077806
2020-03-02	8.000000	4.330656	5.160350
2020-03-03	-3.175931	-2.863244	-3.206728
2020-03-04	4.638463	4.203304	4.172236
2020-03-05	-3.243707	-3.324171	-3.038218
2020-03-06	-1.328008	-1.653111	-1.687225

como vemos en el dato del 2 de marzo de 2020, que teníamos un 9.31, lo cambió por 8 que era nuestro "tope"

Funciones Estadísticas Básicas

Gracias a pandas tenemos a disposición de un clic las siguientes funciones estadísticas básicas:

- Valor máximo: max()
- Valor mínimo: min()
- Índice de valores mínimos y máximos: idxmin() e idxmax()
- Media: mean()
- Mediana: median()
- Producto: prod() o product()
- Suma: sum()
- Ranking: rank()
- Cantidad de valores únicos: nunique()

Atención, les pido encarecidamente que le presten atención a toda esa lista de funciones de Pandas porque en esa listita está el corazón de la eficiencia en el cálculo con python y dataframes, con esas funciones vamos a poder calcular casi todo lo que se nos ocurra, desde el indicador más típico al más complicado, y siempre va a haber seguramente mil maneras de calcular algo, pero con estas funciones les aseguro que el código quedará super limpio, entendible y lo mas importante, la ejecución va a ser eficiente que si recurren a bucles para calcular e iterar a mano algo que podría haberse hecho con estas funciones

Empecemos por los más sencillos max() y min()

Obviamente nos devuelve el máximo y mínimo de cada columna respectivamente Si queremos el máximo o mínimo de una columna concreta debemos indicarlo

```
tabla.max()
```

```
AAPL    13.905182  
SPY     13.960810  
6       16.841302  
dtype: float64
```

```
tabla.min()
```

```
AAPL   -51.870835  
SPY    -9.664497  
7      -8.955709  
dtype: float64
```

De la siguiente forma buscamos el máximo de una determinada columna:

```
tabla.AAPL.max()
```

```
13.90518246958838
```

¿Y si queremos saber no solo el máximo valor de variación de AAPL sino que fecha se dio? Bueno, para ello en lugar de max() usamos el método idxmax() que nos devuelve el índice del máximo

```
tabla.AAPL.idxmax() , tabla.AAPL.max()
```

```
(Timestamp('2008-10-13 00:00:00'), 13.90518246958838)
```

Y si queremos el máximo de una determinada fila, es decir para una fecha ¿cuál de los tres activos tuvo mayor variación y cuál fue esa variación?

En este ejemplo lo averiguamos para el 2 de marzo de 2020:

```
tabla.loc["2020-03-02"].idxmax() , tabla.loc["2020-03-02"].max()
```

```
('AAPL', 9.310067310506298)
```

En este ejemplo vemos cuáles fueron las fechas en que más cayó cada activo:

```
tabla.idxmin()
```

```
AAPL  2000-09-29  
SPY   2008-10-15  
8     2008-10-15  
dtype: datetime64[ns]
```

Cálculos estadísticos básicos

Media y mediana de cada serie: mean() y median() respectivamente

```
tabla.mean()
```

```
AAPL    0.118673  
SPY     0.029583  
9       0.029224  
dtype: float64
```

```
tabla.median()
```

```
AAPL    0.087605  
SPY     0.064885  
13      0.090677  
dtype: float64
```

Por las dudas por si a esta altura alguno no lo sabe, la media es el promedio simple, y la mediana es el valor para el cuál la mitad de los valores de la serie está por encima y la mitad por debajo del mismo, es decir la mediana es el valor que divide a la serie en dos mitades de la misma cantidad de datos (si la serie estuviera ordenada de menor a mayor)

Lo mismo que antes también podemos hacerlo para una columna (activo) o fila (fecha) determinada

```
tabla.AAPL.mean()
```

```
0.1186730297628902
```

```
tabla.loc["2020-03-02"].mean()
```

```
6.267024223344361
```

Función Rank()

Esta función nos devuelve el número de orden dentro de la serie (en lugar de devolver la serie ordenada devuelve los rankings)

Recomiendo fuertemente prestarle atención es una función que acepta varios argumentos, vamos a ver acá solo los más importantes

Arranquemos por un ejemplo básico sin pasarle argumentos al método rank() así como viene sería algo como esto:

```
tabla.rank().head()
```

	AAPL	SPY	QQQ
timestamp			
2000-03-07	602.0	238.0	633.0
2000-03-08	1522.0	1925.0	3955.0
2000-03-09	2674.0	4951.0	4907.0
2000-03-10	4560.0	1160.0	1493.0
2000-03-13	276.0	599.0	275.0

¿Y esos números que son?

Son los rankings ordenando la tabla en orden creciente, es decir que el valor que tenía la tabla para el 7 de marzo para AAPL fue la variación número 602 de todas qlas que tenía

Con un ordenado posterior puedo saber las fechas de los días mas bajistas:

```
tabla.rank().sort_values(by='AAPL', ascending=True).head(5)
```

	AAPL	SPY	QQQ
timestamp			
2000-09-29	1.0	719.0	200.0
2008-09-29	2.0	3.0	7.0
2001-07-18	3.0	1015.0	116.0
2000-12-06	4.0	351.0	117.0
2002-06-19	5.0	144.0	31.0

Como vemos ahí, al ordenar por AAPL, veo que tan bien o mal les fue a los otros esas mismas fechas

Rank(ascending=False)

Si usamos el argumento `ascending` (que por default viene como `True`) le indicamos si queremos que rankeen orden ascendente o descendente

Si quisieramos las fechas en que mayores alzas tuvo SPY podemos hacer:

```
tabla.rank(ascending=False).sort_values(by='SPY', ascending=True).head(5)
```

	AAPL	SPY	QQQ
timestamp			
2008-10-13	1.0	1.0	2.0
2008-10-28	29.0	2.0	4.0
2009-03-23	88.0	3.0	34.0
2008-11-24	4.0	4.0	28.0
2008-11-13	48.0	5.0	29.0

Rank(pct=True)

El argumento `pct` (que por default viene dado como `False`) nos indica si queremos que en lugar de un ranking numérico (de 1 a n, siendo n la cantidad de datos de la serie) nos devuelva un ranking percentil, esto es bastante más útil por lo general en análisis cuantitativo

```
tabla.rank(pct=True).head()
```

	AAPL	SPY	QQQ
timestamp			
2000-03-07	0.119634	0.047297	0.125795
2000-03-08	0.302464	0.382552	0.785970
2000-03-09	0.531399	0.983903	0.975159
2000-03-10	0.906200	0.230525	0.296701
2000-03-13	0.054849	0.119038	0.054650

y ¿qué significan esos números de la respuesta? bien analicémoslo:

Como no aclaramos el argumento `ascending` usa el valor default que es `True`, así que lo que hizo fue ordenar de menor a mayor y nos está diciendo que por ejemplo AAPL el 7 de marzo de 2000 tuvo un movimiento que de 1000 ruedas sería la número 119 (aprox) es decir que fue un día bastante malo para AAPL

Sin embargo, AAPL el 10 de marzo de 2000 nos está diciendo que estuvo dentro del mejor 10% ya que su percentil es mayor a 0.9

Obviamente que si en la misma instrucción hubiéramos configurado `ascending = False`, nos devuelve los percentiles pero ordenados de mayor a menor, es decir que esa tabla debería ser el resultado a hacer $1-X$ (siendo X la matriz anterior) Comprobémoslo:

```
tabla.rank(pct=True, ascending = False).head()
```

	APL	SPY	QQQ
timestamp			
2000-03-07	0.880564	0.952901	0.874404
2000-03-08	0.697734	0.617647	0.214229
2000-03-09	0.468800	0.016296	0.025040
2000-03-10	0.093998	0.769674	0.703498
2000-03-13	0.945350	0.881161	0.945548

Hay mas argumentos de esta función por ejemplo, `method` es el tratamiento que le da a los valores iguales, es decir supongamos que encuentra 10 valores exactamente iguales, ¿que valor de ranking les asigna a cada uno?

Excelente pregunta, bueno, por default viene `rank(method=average)` que como se imaginarán de los 10 valores calcula un promedio y les pone a los 10 el mismo ranking.. También se puede configurar como `rank(method=min, max, first)`, obviamente es fácil deducir que hará cada una de esas configuraciones, y si no se lo imaginan, prueben ustedes en sus compus

Función `nunique()`

Esta función como se imaginarán nos devuelve la cantidad de valores únicos de cada columna

```
tabla.nunique()
```

AAPL	5001
SPY	4999
9	4995
<code>dtype: int64</code>	

Si queremos los valores "repetidos" podemos hacer:

```
tabla.count() - tabla.nunique()
```

AAPL	31
SPY	33
9	37
<code>dtype: int64</code>	

Como la tabla tiene 6 decimales es raro encontrar valores repetidos pero si redondeamos la tabla veremos que es bastante mas común:

```
tabla_redondeada = tabla.round(2)
tabla_redondeada.nunique()
```

```
AAPL    1093
SPY     654
2       842
dtype: int64
```

Repaso de funciones estadísticas

Bueno, necesito hacer un parate en este momento para aclarar algunas cosas

Este libro es parte de una colección de libros de análisis quant de activos financieros, es esperable que cualquiera que se va a poner a aprender estos temas tenga una base de conocimientos de matemática y estadística al menos intermedio.

Dicho esto, asumiendo que la gran mayoría de los lectores de estas líneas tiene ese conocimiento, me voy a dirigir en los siguientes párrafos a quienes no lo tengan, es decir, que quienes tengan una buena base de matemática y estadística pueden empezar a saltar todas estas páginas porque se van a encontrar con que los temas están demasiado simplificados y justamente lo hice así para saldar un poco esa brecha ya que no es este el espacio para enseñar profundamente todos esos conceptos, pero al menos quería dejar una base resumida pero que sirva a quienes no tienen ese conocimiento para que puedan seguir el hilo de esta obra

Para quienes quieran profundizar temas específicos de estadística básica recomiendo el libro:

"Probabilidad y estadística para ingeniería y ciencias"
De Walpole-Myers



Desvío Estándar

Es algo así como la variación típica esperada, asumiendo que la distribución es normal, el desvió estándar es un valor que dentro de un +/- 1desvio me asegura el 68,27% de los valores dentro de ese rango, y dentro de +/- 2desvios me asegura que el 95,45% de los valores está dentro de ese rango

La fórmula del desvío estándar de una distribución es la siguiente:

Siendo:

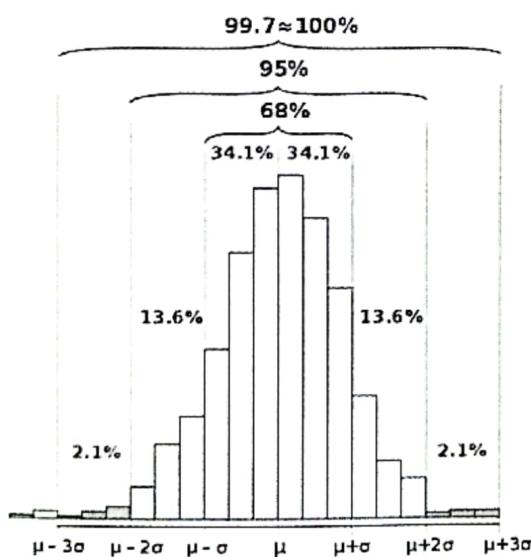
n = cantidad de valores

\bar{x} = promedio de los valores

x_i = cada valor de la serie

$$\sigma = \sqrt{\frac{1}{n} \cdot \sum_{i=1}^n (x_i - \mu)^2}$$

Rango	Probabilidad 1	Frecuencia Aproximada	Frecuencia aproximada para un evento diario
$\mu \pm 0.5\sigma$	38.29%	2 en 3	Cuatro o cinco veces a la semana
$\mu \pm \sigma$	68.27%	1 en 3	Dos veces a la semana
$\mu \pm 1.5\sigma$	86.64%	1 en 7	Semanalmente
$\mu \pm 2\sigma$	95.45%	1 en 22	Cada tres semanas
$\mu \pm 2.5\sigma$	98.76%	1 en 81	Trimestralmente
$\mu \pm 3\sigma$	99.73%	1 en 370	Anualmente
$\mu \pm 3.5\sigma$	99.95%	1 en 2149	Cada 6 años
$\mu \pm 4\sigma$	99.99%	1 en 15 787	Cada 43 años (dos veces en la vida)
$\mu \pm 4.5\sigma$	99.9993%	1 en 147 160	Cada 403 años (una vez en la edad moderna)
$\mu \pm 5\sigma$	99.9999%	1 en 1 744 278	Cada 4776 años (una vez en la historia documentada)
$\mu \pm 5.5\sigma$	99.999996%	1 en 26 330 254	Cada 72 090 años (3 veces en la historia del ser humano)
$\mu \pm 6\sigma$	99.9999998%	1 en 505 797 346	Cada 1.38 millones de años (2 veces en la historia del ser humano)
$\mu \pm 6.5\sigma$	99.99999992%	1 en 12 450 197 393	Cada 34 millones de años (2 veces desde extinción de dinosaurios)
$\mu \pm 7\sigma$	99.999999997%	1 en 390 682 215 445	Cada 1070 millones de años (4 veces en la historia de la Tierra)



Calculemos entonces el desvió estándar de las variaciones diarias de cada activo de nuestra tabla:

```
tabla.std()
```

AAPL	2.509999
SPY	1.193366
0	1.694009
dtype:	float64

¿Y se usa el desvío estándar de las variaciones de un activo en la práctica?

ufff, SI, y muchísimo más de lo que se imaginan, incluso a mi me parece una métrica más importante que el rendimiento mismo, lo van a escuchar normalmente como "volatilidad"

Una manera de expresar la volatilidad es en términos anualizados, dado que sirve para comparar con rendimientos anualizados también y entre ambos calcular ratios de riesgo como los que veremos a continuación

Y como se "anualiza" la volatilidad? sencillo, no voy a entrar en detalle del por qué, pero si les muestro la fórmula que es muy muy sencilla, se parte de la idea de que el "random walk" de los rendimientos diarios sigue los principios del movimiento browniano y a partir de esa base se calcula el desvío anualizado como el desvío diario multiplicado por la raíz cuadrada de la cantidad de ruedas de un año (en un activo que se negocia de lunes a viernes son feriados hablamos de más o menos 250 ruedas)

$$\sigma_n = \frac{\sigma}{\sqrt{n}}$$

Siendo "n" la cantidad de ruedas en un año, σ la volatilidad anualizada y σ_n la volatilidad diaria

Ahora ¿cómo interpretamos esos valores de desvío estándar diario?

Bien, primero calculemos los valores medios de esas variaciones

```
tabla.mean()
```

AAPL	0.118673
SPY	0.029583
12	0.029224
dtype:	float64

Entonces AAPL se mueve en promedio +0,1186% todas las ruedas de la serie
 Pero como tiene un desvío estandar de 2.5099% podemos afirmar que el 68,27% de las veces se va a mover entre -2,3913% y +2,6285%

De donde saco eso? simple de sumar 1 desvió estándar para cada lado de la media

Lo mismo si sumo 2desvios para cada lado estaré contemplando el rango para el 95,45% de las ruedas

¿tan fácil? momento, esto solo funciona en pocos desvios, no funciona para valores mas extremos

No se ilusionen, la distribución de los rendimientos diarios no es una distribución normal, es muy parecida y sirve para los eventos típicos pero se comporta muy diferente en eventos extremos

Para aquellos lectores argentinos que recuerden el evento bursátil del famoso 12 de Agosto de 2019, el día posterior a las elecciones PASO, ese día el movimiento del Merval fue equivalente a 16 sigmas, jaja, debería pasar un evento diario así cada tantos millones de años que no alcanza la historia del universo... obviamente que como decía antes esta contradicción no es que seamos tan especiales en esta época sino que claramente la distribución de rendimientos de un activo financiero no es una distribución normal, es algo muy parecido excepto en las colas (los valores extremos)

Entonces si no sigue una distribución normal ¿para que lo vamos a ver en este curso? bueno, como dije antes, dentro de los +/- 2 a 2.5 sigmas es casi un calco a una distribución normal las distribuciones de los rendimientos diarios de los activos financieros, fuera de ese rango no, pero como el 98% y pico de las ruedas se dan dentro de ese intervalo, este tipo de análisis nos va a ser muy útil, siempre teniendo en cuenta que ese casi 2% de las ruedas que dejamos afuera puede ser muy importante obviamente, para los interesados en este tipo de análisis que no tengan un buen conocimiento de estos temas, les recomiendo fuertemente leer los libros de Nassim Taleb ("¿Existe la suerte?" o "El cisne negro" y si tienen más tiempo y quieren algo más general: "Antifragil")

Ratios de Riesgo básicos

Ratio Sharpe

Hay infinidad de maneras de calcular este ratio, ya veremos más adelante cuando usemos data feed de APIs usos mas reales, por ahora a manera meramente didáctica vamos a explicarlo de forma simplificada

La idea principal del ratio sharpe es comparar el rendimiento contra el riesgo al que se está expuesto para ese rendimiento ya que todos buscamos altos rendimientos pero no a costa de estar expuestos a movimientos abismales (sobre todo para abajo, pero ese detalle lo veremos en el siguiente ratio el Sortino)

Entonces básicamente calculamos un cociente entre rendimiento neto (el del activo menos el de la tasa libre de riesgo) y la volatilidad (desvío estándar de los rendimientos)

$$SharpeRatio = \frac{r - r_f}{\sigma}$$

Siendo:

r = rendimiento del activo

r_f = rendimiento libre de riesgo

σ = desvío estándar o volatilidad del activo

Cabe aclarar que como tasa libre de riesgo se suele tomar la tasa del bono a 10 años del tesoro de EEUU, es relativo también, si estamos evaluando activos en pesos argentinos, se debería tomar la tasa libre de riesgo en esa moneda, jaja, si ya se libre de riesgo en pesos argentinos es un poco mucho pedir, pero ojo, no hablamos de riesgo cambiario ni riesgo inflacionario, sino libre de riesgo de baja nominal, por lo que si evaluamos activos en pesos argentinos bien podría tomarse tasa de caución corta en pesos o tasa de plazos fijos garantizados por poner algún ejemplo

A lo que voy con el concepto "libre de riesgo" es a que allí debe colocarse la "tasa natural" es decir la diferencia de valor del dinero en el tiempo piceada por el mercado, o sea lo que se obtiene sin hacer nada ni asumir riesgo especulativo

Ok, y entonces, ¿cómo calculamos este ratio?

Bueno, como les adelantaba hay miles de maneras de calcularlo, una es usando variaciones diarias, otra es anualizando otra es haciendo un cálculo "rolling" es decir medias móviles de los componentes de la fórmula o del resultado de la fórmula, en fin hay muchas, vamos a mostrar lo más básico porque como les dije esta es la presentación nomás del tema, luego profundizaremos más adelante

```
tabla = pd.read_excel('AAPL_SPY_QQQ.xlsx')
tabla.set_index("timestamp", inplace=True)

rendimientos = tabla.mean()
desvios = tabla.std()
tasaLibreRiesgo = 0.024 / 360

tablaRatios = pd.DataFrame(index=tabla.columns)
tablaRatios['sharpeDiario'] = (rendimientos - tasaLibreRiesgo)/desvios
tablaRatios['sharpeAnualizado'] = tablaRatios.sharpeDiario * 250**0.5

tablaRatios
```

	sharpeDiario	sharpeAnualizado
AAPL	0.047254	0.747144
SPY	0.024734	0.391073
QQQ	0.017212	0.272143

Ratio Sortino

El ratio sortino es lo mismo que el ratio sharpe con la particularidad que no toma en cuenta todas las variaciones sino solo las negativas para calcular el desvío

Cuál es la lógica subyacente de este ratio entonces? muy sencillo, la mayoría de la gente prefiere altos rendimientos siempre y cuando esto no suponga alta volatilidad sencillamente porque la mayoría de la gente sufre demasiada angustia al ver su posición caer muy fuerte, por lo tanto lo de "baja volatilidad" en realidad lo que significa es "baja volatilidad negativa" es decir a nadie le jode alta volatilidad positiva, es decir que de repente no pase nada pero de repente nuestro activo "se vuelve" lo que jode es que se "haga percha" es decir los movimientos bruscos hacia abajo son los que empeoran la ecuación rentabilidad retorno riesgo y no los movimientos bruscos hacia arriba

Bueno, dicho esto, vamos a aclarar que en realidad no es que importan los rendimientos menores a CERO, sino los menores a un umbral máximo de pérdida soportada, se suele tomar CERO pero para un valor anualizado podría usarse tranquilamente un -10% es decir que contabilizaríamos en ese caso las variaciones diarias equivalentes a -10%/250 aprox

Para calcular el ratio sortino no podemos hacerlo con los 3 activos juntos como lo hicimos en el ratio sharpe porque si filtramos los valores negativos de AAPL nos devolverá una tabla donde SPY tiene fechas con valores positivos, y así con los otros, debemos calcularlos por separado, veamos un ejemplo

```
tabla = pd.read_excel('AAPL_SPY_QQQ.xlsx')
tabla.set_index("timestamp", inplace=True)
rendimientos = tabla.mean() tasaLibreRiesgo
= 0.024 / 360

activos = ["AAPL", "SPY", "QQQ"]
sortinos = []
for activo in activos:
    filtro = tabla.AAPL.loc[tabla.AAPL < 0]
    desvioNeg = filtro.std()
    sortinos.append((rendimientos[activo] - tasaLibreRiesgo) / desvioNeg)

tablaRatios = pd.DataFrame(index=tabla.columns)
tablaRatios['sortinoDiario'] = sortinos
tablaRatios['sortinoAnualizado'] = tablaRatios.sortinoDiario * 250**0.5
tablaRatios
```

	sortinoDiario	sortinoAnualizado
AAPL	0.059896	0.947044
SPY	0.014906	0.235680
QQQ	0.014724	0.232811

Y por último veamos una comparativa de los 2 ratios diarios y anualizados

```
tabla = pd.read_excel('AAPL_SPY_QQQ.xlsx')
tabla.set_index("timestamp", inplace=True)

rendimientos = tabla.mean()
desvios = tabla.std()
tasaLibreRiesgo = 0.024 / 360

tablaRatios = pd.DataFrame(index=tabla.columns)
tablaRatios['sharpeDiario'] = (rendimientos - tasaLibreRiesgo)/desvios
tablaRatios['sharpeAnualizado'] = tablaRatios.sharpeDiario * 250**0.5

activos = ["AAPL", "SPY", "QQQ"]
sortinos = []
for activo in activos:
    filtro = tabla.AAPL.loc[tabla.AAPL < 0]
    desvioNeg = filtro.std()
    sortinos.append((rendimientos[activo] - tasaLibreRiesgo) / desvioNeg)

tablaRatios['sortinoDiario'] = sortinos
tablaRatios['sortinoAnualizado'] = tablaRatios.sortinoDiario * 250**0.5
tablaRatios.round(2)
```

	sharpeDiario	sharpeAnualizado	sortinoDiario	sortinoAnualizado
AAPL	0.05	0.75	0.06	0.95
SPY	0.02	0.39	0.01	0.24
QQQ	0.02	0.27	0.01	0.23

Analicemos un poco esa última tabla, vemos que por ejemplo AAPL presenta un ratio sortino mejor que su propio ratio sharpe, esto significa que su ratio sharpe en parte está incidido por altas volatilidades positivas, sin embargo si miramos SPY vemos que su ratio sortino es menor que su ratio sharpe, con lo cual para SPY podemos decir que las volatilidades negativas pesan más en el ratio sharpe que en el caso de AAPL

¿y para qué me sirve analizar estos ratios?

Bien, el punto es el siguiente, si uno es como la gran mayoría de personas que sufre cuando su cartera tiene pérdidas significativas, vamos a tender a preferir "sacrificar rendimiento" o conformarnos con menos rendimiento potencial, a cambio de recortar esas bajas ·bruscas, porque simplemente nuestra aversión al riesgo nos marca algún límite en algún lugar

Dicho esto, lo que vamos a buscar es maximizar el ratio Sharpe o Sortino, es decir a la misma volatilidad maximizar el retorno o dicho al revés, al mismo retorno minimizar la volatilidad.. Ahora bien, ¿cuál volatilidad queremos minimizar? muy sencillo la negativa, por lo tanto el ratio sortino es más razonable para mejorar el bienestar o conformidad de un inversor respecto de su relación riesgo/retorno

Obviamente no todo es tan sencillo, hay autores que se basan en que si un determinado activo tiene un sesgo hacia la volatilidad negativa mayor que a la positiva es porque desarrolló una determinada tendencia, con lo cual afirman que hay una alta probabilidad que revierta en algún momento esa relación o sesgo de volatilidades positivas y negativas haciendo que nuestro análisis de solo la volatilidad negativa no sea correcto, es discutible como todo, depende de muchos factores que no vienen al caso porque no termino mas el libro este asi que lo corto acá, ya retomaremos con esto de las tendencias y las regresiones en volatilidades y sus signos mas adelante, lo prometo

Error estandar

El **Error del desvío estándar**, que se calcula como:

$$\frac{\sigma}{\sqrt{n}}$$

Siendo "n" el tamaño de la muestra,

Es el error estimado del desvío estándar de la muestra respecto al desvío estándar de la población

```
tabla.sem()
```

AAPL	0.035384
SPY	0.016823
5	0.023881
dtype:	float64

Obviamente como nuestra muestra es muy grande (mas de 5000 datos) el error estandar es muy bajo

Varianza

La **varianza** de cada columna que es el cuadrado del desvío estándar σ^2 se calcula con el método: var()

```
tabla.var()
```

AAPL	6.300093
SPY	1.424123
#	2.869668
dtype:	float64

Covarianza

¿y esto con que se come?

La covarianza nos da una idea de la variación conjunta de dos variables respecto a sus medias.

¿Para qué nos puede servir?

Para saber si existe una dependencia entre las variables (es decir si está relacionado el movimiento de una respecto al de la otra)

A mayor covarianza de una variable respecto de otras, mayor dependencia o correlación existirá con ella

¿Cuál es la covarianza entre dos series iguales?

Estos son los valores de las diagonales del cuadro y son ni más ni menos que la varianza de cada serie (el cuadrado de su desvío estándar)

```
tabla.cov().round(2)
```

	AAPL	SPY	QQQ
AAPL	6.30	1.57	2.65
SPY	1.57	1.42	1.69
QQQ	2.65	1.69	2.87

De la tabla anterior deducimos que AAPL está más relacionada con AAPL (una obviedad) y luego está más ligada al QQQ que al SPY (dado que el ETF QQQ es de acciones tecnológicas como AAPL)

Correlación

Se pueden usar los 3 métodos científicamente más utilizados para correlacionar

- Pearson (Default, ideal para distribuciones normales)
- Spearman (Ideal para distribuciones con valores muy anormales, +/- 5sigma o mas ya que correlaciona por rangos)
- Kendall (Es una correlación más estricta, ojo que consume más recursos porque su formulación es bastante más compleja que las anteriores)

¿Qué ventaja tienen los coeficientes de correlación respecto de la covarianza?

La principal ventaja de los coeficientes de correlación respecto de la covarianza, es que son independientes de las escalas de valores de las series ya que la imagen de estos coeficientes es el intervalo (-1,1)

- Donde -1 significa que son variables exactamente opuestas es decir que siempre que una suba la otra baja,
- 0 significa que no tienen ninguna correlación,
- y 1 que están totalmente correlacionadas, es decir que cada vez que una sube la otra sube en proporciones iguales respecto de sus escalas

Esta característica que las hace independientes de sus escalas, hace que los coeficientes de correlación sean mucho más útiles para comprar peras con manzanas, ver por ejemplo que si estudio las covarianzas pareciera que hay más relación entre AAPL y QQQ que entre QQQ y SPY, cuando claramente está más correlacionado el Nasdaq al S&P que a AAPL en sí, esa confusión se genera en las covarianzas cuando se comparan correlaciones entre activos con varianzas altas (Acciones) con activos con varianzas bajas (índices)

Veamos cómo nos dan los tres métodos entre AAPL SPY y QQQ:

```
tabla.corr(method="pearson").round(2)
```

	AAPL	SPY	QQQ
AAPL	1.00	0.53	0.62
SPY	0.53	1.00	0.84
QQQ	0.62	0.84	1.00

```
tabla.corr(method="spearman").round(2)
```

	AAPL	SPY	QQQ
AAPL	1.00	0.55	0.66
SPY	0.55	1.00	0.86
QQQ	0.66	0.86	1.00

```
tabla.corr(method="kendall").round(2)
```

	APL	SPY	QQQ
AAPL	1.0	0.40	0.50
SPY	0.4	1.00	0.69
QQQ	0.5	0.69	1.00

¿Y para que quiero saber los coeficientes de correlación entre diferentes acciones e índices?

Ya lo veremos, pero claramente para diversificar y mejorar los ratios de riesgo/retorno de una cartera

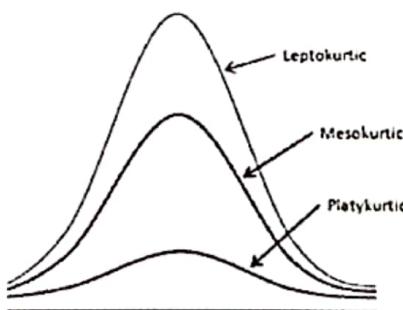
Si tengo en una cartera activos de correlación negativa entre sí, seguramente performará mucho mejor la cartera a igual riesgo que otra cartera con correlaciones entre los activos muy alta cercanas a 1 todas

Más funciones Estadísticas

Kurtosis:

La Curtosis nos da una idea de la forma, mientras más alta, más valores cerca de la media y más gruesas las colas:

- Leptocúrtica, Curtosis > 3 (más apuntada y con colas más gruesas que la normal)
- Platicúrtica, Curtosis < 3 (menos apuntada y con colas menos gruesas que la norma)
- Mesocúrtica, Curtosis = 3 (tiene una distribución normal)



Lo calculamos en python:

```
tabla.kurtosis()
```

```
AAPL    40.134582
SPY     11.128320
QQQ     7.599452
dtype: float64
```

Recordemos que para calcular todo esto estamos trabajando con los métodos de la librería Pandas, ya que nuestra variable "tabla" es un DataFrame construido en Pandas

Sesgo

El sesgo nos da una idea de la forma
O una posición relativa de la media respecto a la mediana

A mayor skew más corrida la media hacia la derecha de la mediana y viceversa

```
tabla.skew()
```

```
AAPL    -1.721671
SPY     0.132564
QQQ     0.289223
dtype: float64
```

Cuantiles

Los cuantiles son una generalización de los cuartiles, quintiles, deciles, percentiles, etc..
En números:

- Los cuartiles serían cuantiles 0.25, 0.50 y 0.75
- Es decir el primer cuartil que es el cuantil 0,25 es el primer cuarto de la población ordenado de menor a mayor
- Los quintiles son los cuantiles 0.2, 0.4, 0.6 y 0.8
- Los deciles son los cuantiles 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8 y 0.9 Los percentiles son los cuantiles 0.01, 0.02, 0.03 ... 0.99

O sea que el quantile q=0.5 es el valor para el cual el 50% de la población está por debajo del mismo

```
tabla.quantile(q = 0.5)
```

```
AAPL    0.087605
SPY     0.064885
QQQ     0.090677
Name: 0.5, dtype: float64
```

Y para que corno sirve esto? Bueno, imaginemos que quiero saber que caída de un activo tiene menos del 1% de probabilidad de que suceda (Esto siempre estadísticamente hablando)

```
tabla.quantile(0.01)
```

```
AAPL    -6.327301
SPY     -3.318685
QQQ     -4.852816
Name: 0.01, dtype: float64
```

Es decir que una caída de AAPL diaria de -6.3273% o peor tiene un 1% de probabilidad de que suceda

Gráficos con Pandas

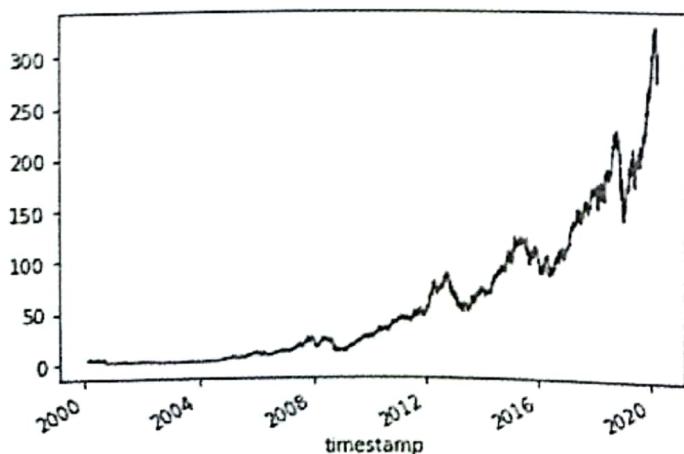
Claramente pandas no es la librería para realizar gráficos, ya veremos más adelante la librería matplotlib que es un paquete específicamente diseñado para graficar todo tipo de serie de datos, pero por ahora para representar gráficamente de modo extremadamente sencillo una serie vamos a ver las funciones de plotting de la librería pandas (que dicho sea de paso usa de backend para realizarlos los métodos de la librería matplotlib)

Como les digo más adelante vamos a ver bien en detalle esa librería, acá solo les dejo algunas de las opciones más típicas solo para mostrarles algo rápido y en pocas líneas

Empecemos por lo más sencillo, un gráfico lineal, de los precios

```
data = pd.read_excel('AAPL.xlsx')
data = data.sort_values(by='timestamp', ascending=True)
data.set_index("timestamp", inplace=True)
data['variacion'] = data['adjusted_close'].pct_change()*100

#Ahora si lo graficamos:
data.adjusted_close.plot()
```



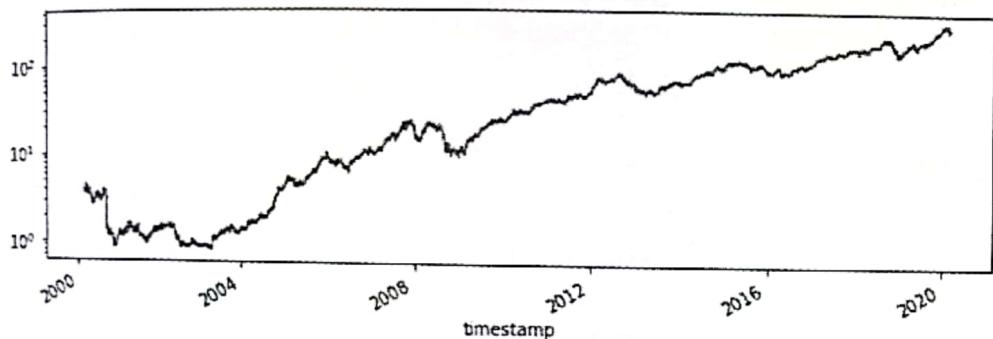
Primer parámetro **logy** por default es False, y obviamente me indica si queremos escala logarítmica en el eje Y

Del mismo modo está el parámetro **logx** para el eje X
Con: `figsize=(pulgadasX,pulgadasY)`

Le pasamos el tamaño de salida que queremos

```
data = pd.read_excel('AAPL.xlsx')
data = data.sort_values(by='timestamp', ascending=True)
data.set_index("timestamp", inplace=True)
data['variacion'] = data['adjusted_close'].pct_change()*100

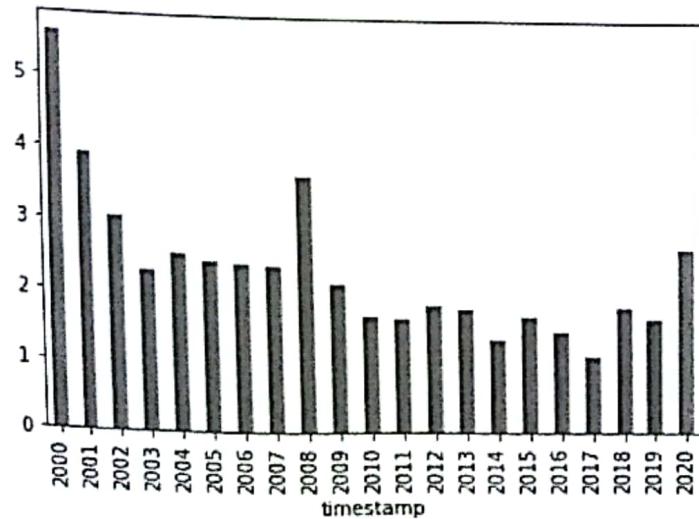
data.adjusted_close.plot(logy=True, figsize=(10,3))
```



Si quisiera un gráfico de barras

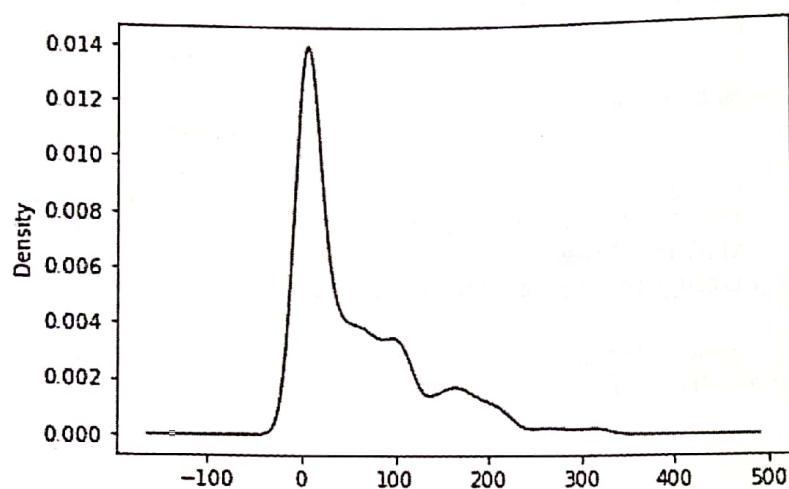
```
import datetime as dt
data = pd.read_excel('AAPL.xlsx')
data = data.sort_values('timestamp', ascending=True)
data['variacion'] = data['adjusted_close'].pct_change()*100
agg = pd.DataFrame()

agg['Volatilidad'] = data.variacion.groupby(data.timestamp.dt.year).std()
agg.Volatilidad.plot(kind="bar")
```



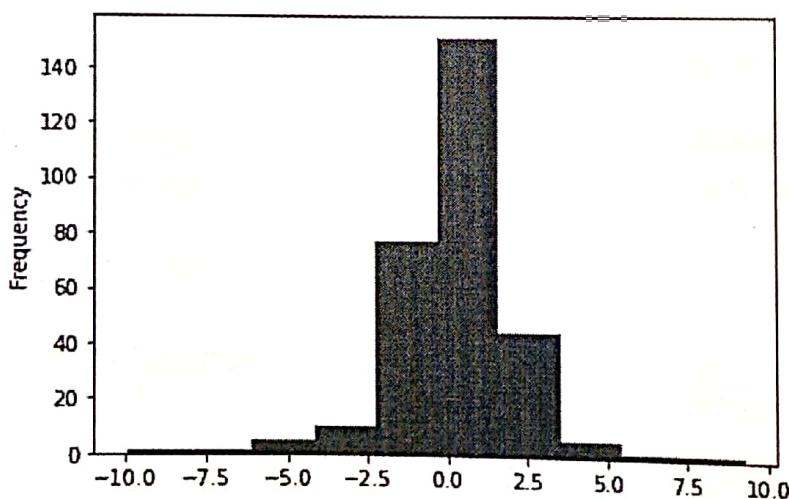
Un gráfico de densidad (Probabilidad de ocurrencia)

```
data.adjusted_close.plot(kind="kde")
```



O bien un histograma

```
data.loc[data.index>"2019-01-01"].variacion.plot(kind="hist")
```



Ejercicios Propuestos

1 - Con los conocimientos vistos en este módulo construir una tabla (partiendo de los datos de cierres diarios) que contenga las bandas de bollinger para cada fecha. Las bandas de Bollinger son dos bandas de precios una superior y otra inferior, que son la media móvil de "X" ruedas +/- "Y" desvíos estándar

Definir para ello una variable llamada "ruedas" y otras llamada "desvíos" al inicio como parámetros de las bandas a construir

1b- Guardar la tabla en el archivo "bollinger.xlsx" para usarlo en ejercicios posteriores

2 - Partiendo del ejercicio anterior, armar una tabla que cuente para cada año, las ruedas en las que el cierre haya estado por encima de la banda de bollinger superior o por debajo de la inferior y mostrar los resultados en un gráfico de barras

3 - Partiendo de la tabla de las bandas de bollinger de AAPL, armar una tabla que muestre una columna con la variación forward 5 ruedas adelante después de que el precio ajustado haya estado por encima de la banda de bollinger superior y calcular la media de esa columna

Reflexión: Si bien es extremadamente simplista el análisis ¿no se va pareciendo esto a un backtesting acerca de si hay una tendencia a la corrección en el corto plazo cuando se superan las bandas de bollinger? ¿Viendo esto que le dirían a alguien que cuando rompe la banda de bollinger hacia arriba hace trading a la baja esperando corrección?

4 - Armar un script similar al del ejercicio anterior pero que en lugar de retronarme la media de los rendimientos forward solo de 5ruedas luego de comprar cuando se supere la banda de bollinger, que me retorne un gráfico de barras que me muestre ese rendimiento forward (por rueda) para 1, 2, 3 y así hasta 20 ruedas forward en el mismo gráfico

Ayuda: Definir primero una lista vacía de resultados e ir llenándola dentro de un ciclo FOR con la media de lo que calculamos en el ejercicio anterior, y luego de terminar el ciclo armar un dataframe con la lista graficarlo

5- Armar una tabla con todas las variaciones diarias ajustadas en 2013 de los siguientes ADRs:

BBAR, BMA, CRES, EDN, GGAL, PAM, TEO, TGS, YPF

- a- Guardar el resultado en un archivo ARDs.xlsx
- b- Leer ese archivo y setearle la fecha como índice
- c- A partir de este archivo armar otra tabla con los valores de rendimiento acumulado y volatilidad anualizada
- d- Calcular el sharpe de 2013 para ellos c/freeRisk del 3% y decidir cual performó mejor entre EDN e YPF (justificar)
- e- ¿Y entre BBAR, BMA y TGS? ¿Cuál hubieras recomendado para una cartera?
- f- Graficar en un chart de barras las volatilidades de los ADRs

Para los datos de precios de esos ADRs buscar en la carpeta ADRs de: <http://bit.ly/39Am64T> (<http://bit.ly/39Am64T>)

6- Armar una tabla con las variaciones de los ADRs del Ejercicio 5 pero para todo el historial y

- a- Guardarlo en el archivo ARDs_historial.xlsx
- b- Leer ese archivo y hacer un grafico de barras donde cada barra indique la variación negativa necesaria para que su probabilidad de ocurrencia sea menos al 1%
- c- Idem al punto b, pero para retornos positivos y que su probabilidad de ocurrencia sea menor al 0,1%
- d- Mostrar una tabla con las ultimas 5 ruedas de los ADRs y la probabilidad de ocurrencia de una variación superior a la dada
- e- Mostrar un gráfico de barras que muestre la probabilidad % de cada ADR de caer más del 5% en una sola rueda
- f- Mostrar una tabla de correlaciones entre los ADRs
- g- Mostrar para cada activo su par menos correlacionado
- h- Mostrar para cada activo su par más correlacionado (excepto si mismo)

```

#-----#
# Rta Ejercicio 1 #
#-----#

ruedas = 20
desvios = 2

data = pd.read_excel('AAPL.xlsx')
data = data.sort_values(by='timestamp', ascending=True)
data.set_index('timestamp', inplace=True)
data.index.name='fecha'
data['variacion'] = round(data['adjusted_close'].pct_change() * 100,2)
data['desvio'] = data.adjusted_close.rolling(ruedas).std()
data['sma'] = data.adjusted_close.rolling(ruedas).mean()
data['bollinger_sup'] = data['sma'] + desvios*data['desvio']
data['bollinger_inf'] = data['sma'] - desvios*data['desvio']
data = data.dropna().round(4)

bol = pd.DataFrame(index=data.index)
bol['adjusted_close']=data['adjusted_close']
bol['sma']=data['sma']
bol['bollinger_sup']=data['bollinger_sup']
bol['bollinger_inf']=data['bollinger_inf']

bol.head()

```

	adjusted_close	sma	bollinger_sup	bollinger_inf
fecha				

fecha	adjusted_close	sma	bollinger_sup	bollinger_inf
2000-03-31	4.21	3.99	4.55	3.43
2000-04-03	4.13	4.00	4.56	3.45
2000-04-04	3.95	4.01	4.56	3.46
2000-04-05	4.04	4.02	4.56	3.48
2000-04-06	3.88	4.03	4.56	3.49

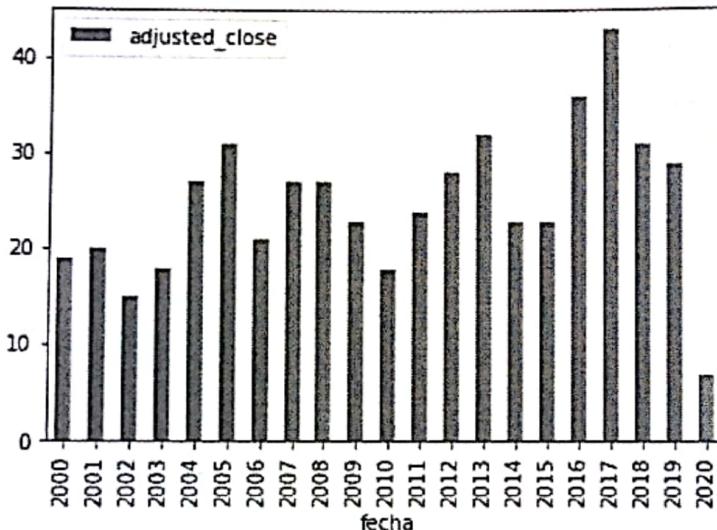
```

#-----#
# Rta Ejercicio 1b #
#-----#

bol.to_excel("bollinger.xlsx")

```

```
#-----#
# Rta Ejercicio 2 #
#-----#
data = pd.read_excel('bollinger.xlsx')
data_filtrada = data[
    (data.adjusted_close > data.bollinger_sup) |
    (data.adjusted_close < data.bollinger_inf)]
data_filtrada.set_index("fecha", inplace=True)
conteo = data_filtrada.adjusted_close.groupby(data_filtrada.index.year)
conteo = conteo.count().to_frame()
conteo.plot.bar()
<matplotlib.axes._subplots.AxesSubplot at 0x1b5151db308>
```



```
#-----#
# Rta Ejercicio 3 #
#-----#
data = pd.read_excel('bollinger.xlsx')
data.set_index("fecha", inplace=True)
data['precio_forward'] = data.adjusted_close.shift(-5)
data['variacion_forward'] = (data.precio_forward / data.adjusted_close - 1)*100
data_filtrada = data[data.adjusted_close > data.bollinger_sup]
data_filtrada.variacion_forward.mean()
```

0.992398713287691

```
#-----#
# Rta Ejercicio 4 #
#-----#

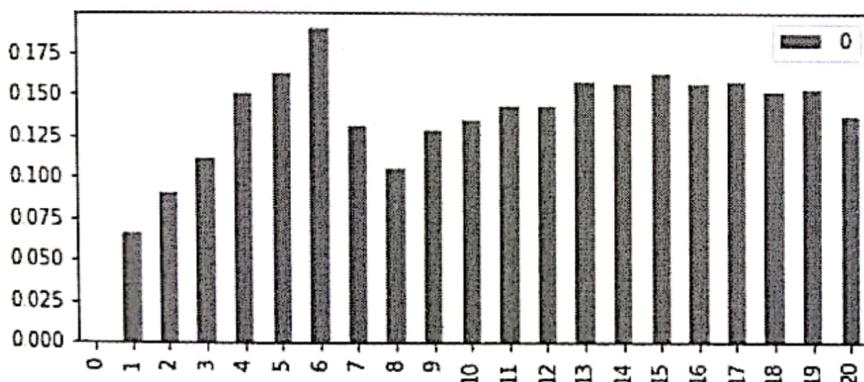
data = pd.read_excel('bollinger.xlsx')
data.set_index("fecha", inplace=True)
data = data[data.index < "2009-01-01"]

results = []

for i in range (0,21):
    data['precio_forward'] = data.adjusted_close.shift(-i)
    data['variacion_forward'] = (data.precio_forward / data.adjusted_close - 1
)*100
    data_filtrada = data[data.adjusted_close > data.bollinger_sup]
    results.append(data_filtrada.variacion_forward.mean()/(i+1))

graf = pd.DataFrame(results)
graf.plot(kind="bar", figsize=(7,3))
```

<matplotlib.axes._subplots.AxesSubplot at 0x1b517f49188>



```
#-----#
# Rta Ejercicio 5a #
#-----#

activos = ["BBAR", "BMA", "CRESY", "EDN", "GGAL", "PAM", "TEO", "TGS", "YPF"]
listaVariaciones = []
for activo in activos:
    tabla = pd.read_excel('ADRs/' + str(activo) + '.xlsx')
    tabla = tabla.sort_values('timestamp', ascending=True)
    tabla['change_pct'] = tabla['adjusted_close'].pct_change()*100
    tabla.set_index('timestamp', inplace=True)
    listaVariaciones.append(tabla['change_pct'])

ret = pd.concat(listaVariaciones, axis=1)
ret.columns=activos
ret = ret.loc[(ret.index >= "2013-01-01") & (ret.index <= "2013-12-31")]
ret.dropna().to_excel("ADRs_2013.xlsx")
```

```
#-----#
# Rta Ejercicio 5b #
#-----#
tabla = pd.read_excel("ADRs_2013.xlsx")
tabla.set_index("timestamp", inplace=True)
tabla.head().round(2)
```

	BBAR	BMA	CRESY	EDN	GGAL	PAM	TEO	TGS	YPF
timestamp									
2013-01-02	4.18	5.02	5.29	2.56	6.65	2.03	6.59	0.00	2.68
2013-01-03	-0.57	-1.99	1.26	0.00	-1.27	1.99	1.90	3.98	-0.40
2013-01-04	-1.92	-2.30	0.79	0.00	-1.00	4.19	0.32	0.00	-0.07
2013-01-07	1.57	2.41	0.45	0.00	-1.45	1.61	1.21	0.00	0.54
2013-01-08	-1.54	-2.46	-5.35	-0.50	-1.65	-1.32	-1.59	1.64	5.02

```
#-----#
# Rta Ejercicio 5c #
#-----#
tabla = pd.read_excel("ADRs_2013.xlsx")
tabla.set_index("timestamp", inplace=True)

tablaFinal = pd.DataFrame(index=tabla.columns)
tablaFinal['retornoAcum'] = (((tabla/100+1).prod()-1)*100)
tablaFinal['volatilidad'] = tabla.std() * (len(tabla)**0.5)

tablaFinal
```

	retornoAcum	volatilidad
BBAR	38.369636	50.578205
BMA	33.792095	43.260035
CRESY	24.830521	33.820285
EDN	159.487179	89.964986
GGAL	58.007813	47.226990
PAM	52.325581	49.632494
TEO	57.270592	41.318576
TGS	37.220344	39.075911
YPF	127.751886	43.632829

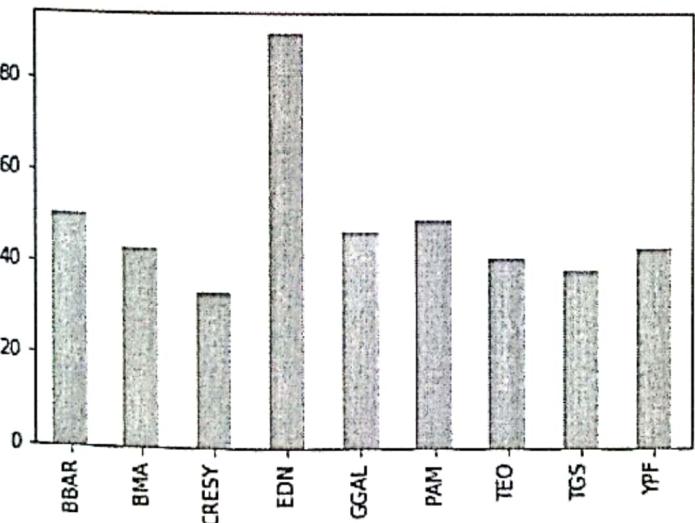
```
#-----#
# Rta Ejercicio 5 d,e #
#-----#
tablaFinal['sharpe'] = (tablaFinal.retornoAcum -3) / tablaFinal.volatilidad
tablaFinal
```

	retornoAcum	volatilidad	sharpe
BBAR	38.369636	50.578205	0.699306
BMA	33.792095	43.260035	0.711791
CRESY	24.830521	33.820285	0.645486
EDN	159.487179	89.964986	1.739423
GGAL	58.007813	47.226990	1.164754
PAM	52.325581	49.632494	0.993816
TEO	57.270592	41.318576	1.313467
TGS	37.220344	39.075911	0.875740
YPF	127.751886	43.632829	2.859129

```
#-----#
# Rta Ejercicio 5f #
#-----#
tabla = pd.read_excel('ADRs.xlsx')
tabla.set_index("timestamp", inplace=True)

desvios = tabla.std() * 250**0.5
desvios.plot(kind="bar")
```

<matplotlib.axes._subplots.AxesSubplot at 0x1b51b8eff08>

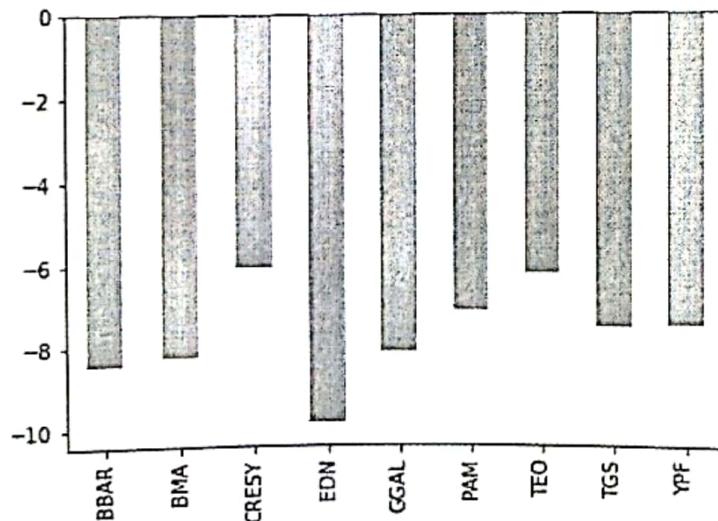


```
#-----#
# Rta Ejercicio 6a #
#-----#
activos = ["BBAR", "BMA", "CRESY", "EDN", "GGAL", "PAM", "TEO", "TGS", "YPF"]
listaVariaciones = []
for activo in activos:
    tabla = pd.read_excel('ADRs/' + str(activo) + '.xlsx')
    tabla = tabla.sort_values('timestamp', ascending=True)
    tabla['change_pct'] = tabla['adjusted_close'].pct_change()*100
    tabla.set_index('timestamp', inplace=True)
    listaVariaciones.append(tabla['change_pct'])

tablaFinal = pd.concat(listaVariaciones, axis=1)
tablaFinal.columns=activos
tablaFinal.dropna().to_excel("ADRs_historial.xlsx")
```

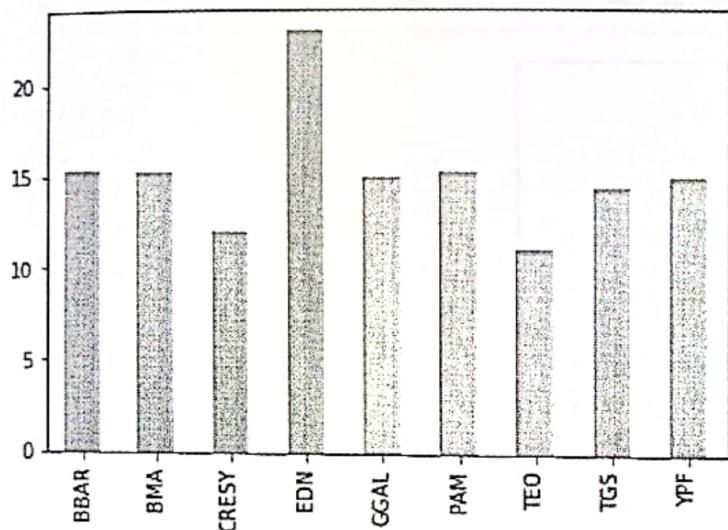
```
#-----#
# Rta Ejercicio 6b #
#-----#
tabla = pd.read_excel("ADRs_historial.xlsx")
tabla.set_index("timestamp", inplace=True)
tabla.quantile(0.01).plot(kind="bar")
```

<matplotlib.axes._subplots.AxesSubplot at 0x1b51baba348>



```
#-----#
# Rta Ejercicio 6c #
#-----#
tabla = pd.read_excel("ADRs_historial.xlsx")
tabla.set_index("timestamp", inplace=True)
tabla.quantile(0.999).plot(kind="bar")
```

<matplotlib.axes._subplots.AxesSubplot at 0x1b548b1ff48>



```
#-----#
# Rta Ejercicio 6d #
#-----#
tabla = pd.read_excel("ADRs_historial.xlsx")
tabla.set_index("timestamp", inplace=True)
res = tabla.rank(pct=True, ascending=False).tail()*100
res.round(2)
#tabla.tail()
```

timestamp	BBAR	BMA	CRESY	EDN	GGAL	PAM	TEO	TGS	YPF	
2020-03-30	14.48	14.40		11.68	85.25	53.18	16.11	0.34	59.59	3.60
2020-03-31	0.99	5.12		0.38	3.83	6.44	0.87	13.57	0.57	9.02
2020-04-01	84.57	87.91		94.05	93.90	91.09	97.95	97.50	87.68	86.43
2020-04-02	21.80	11.68		96.78	5.31	3.34	33.17	55.27	34.50	1.44
2020-04-03	48.96	16.19		90.37	92.99	23.77	18.95	82.71	4.28	48.73

```
#-----#
# Rta Ejercicio 6e #
#-----#

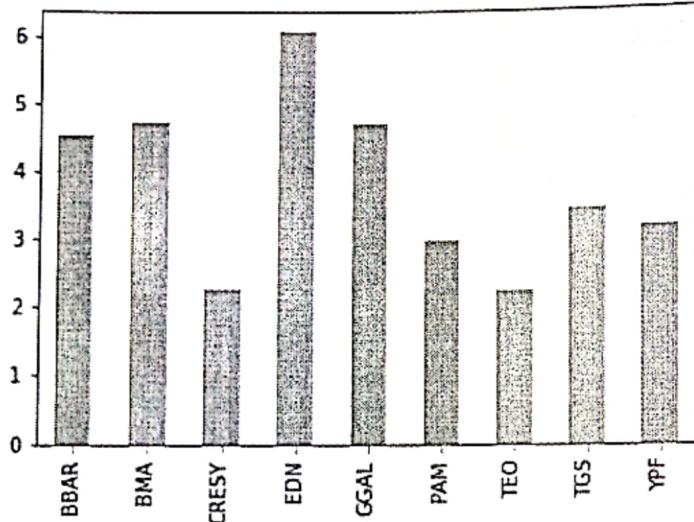

|      |     |       |     |      |     |     |     |     |
|------|-----|-------|-----|------|-----|-----|-----|-----|
| BBAR | BMA | CRESY | EDN | GGAL | PAM | TEO | TGS | YPF |
| 4.5  | 4.7 | 2.2   | 6.0 | 4.7  | 3.0 | 2.2 | 3.5 | 3.2 |


```

tabla = pd.read_excel("ADRs_historial.xlsx")
 tabla.set_index("timestamp", inplace=True)

for column in tabla.columns:
 tabla.loc["Valor Fijo -10%", column] = -5

res = tabla.rank(pct=True).round(4)*100
 res.loc["Valor Fijo -10%"].plot(kind="bar")



```
#-----#
# Rta Ejercicio 6f #
#-----#


|      |      |       |      |      |      |      |      |      |
|------|------|-------|------|------|------|------|------|------|
| BBAR | BMA  | CRESY | EDN  | GGAL | PAM  | TEO  | TGS  | YPF  |
| 1.00 | 0.79 | 0.50  | 0.53 | 0.81 | 0.65 | 0.57 | 0.52 | 0.51 |
| 0.79 | 1.00 | 0.51  | 0.52 | 0.81 | 0.64 | 0.57 | 0.53 | 0.52 |
| 0.50 | 0.51 | 1.00  | 0.40 | 0.51 | 0.47 | 0.44 | 0.42 | 0.41 |
| 0.53 | 0.52 | 0.40  | 1.00 | 0.53 | 0.60 | 0.41 | 0.44 | 0.40 |
| 0.81 | 0.81 | 0.51  | 0.53 | 1.00 | 0.65 | 0.57 | 0.53 | 0.54 |
| 0.65 | 0.64 | 0.47  | 0.60 | 0.65 | 1.00 | 0.48 | 0.58 | 0.48 |
| 0.57 | 0.57 | 0.44  | 0.41 | 0.57 | 0.48 | 1.00 | 0.40 | 0.44 |
| 0.52 | 0.53 | 0.42  | 0.44 | 0.53 | 0.58 | 0.40 | 1.00 | 0.46 |
| 0.51 | 0.52 | 0.41  | 0.40 | 0.54 | 0.48 | 0.44 | 0.46 | 1.00 |


```

tabla = pd.read_excel("ADRs_historial.xlsx")
 tabla.set_index("timestamp", inplace=True)
 correlaciones = tabla.corr().round(2)
 correlaciones

	BBAR	BMA	CRESY	EDN	GGAL	PAM	TEO	TGS	YPF
BBAR	1.00	0.79	0.50	0.53	0.81	0.65	0.57	0.52	0.51
BMA	0.79	1.00	0.51	0.52	0.81	0.64	0.57	0.53	0.52
CRESY	0.50	0.51	1.00	0.40	0.51	0.47	0.44	0.42	0.41
EDN	0.53	0.52	0.40	1.00	0.53	0.60	0.41	0.44	0.40
GGAL	0.81	0.81	0.51	0.53	1.00	0.65	0.57	0.53	0.54
PAM	0.65	0.64	0.47	0.60	0.65	1.00	0.48	0.58	0.48
TEO	0.57	0.57	0.44	0.41	0.57	0.48	1.00	0.40	0.44
TGS	0.52	0.53	0.42	0.44	0.53	0.58	0.40	1.00	0.46
YPF	0.51	0.52	0.41	0.40	0.54	0.48	0.44	0.46	1.00

```
#-----#
# Rta Ejercicio 6g #
#-----#
tabla = pd.read_excel("ADRs_historial.xlsx")
tabla.set_index("timestamp", inplace=True)
correlaciones = tabla.corr().round(2)
correlaciones.idxmin()
```

```
BBAR      CRESY
BMA       CRESY
CRESY     EDN
EDN       CRESY
GGAL      CRESY
PAM       CRESY
TEO       TGS
TGS       TEO
YPF       EDN
dtype: object
```

```
#-----#
# Rta Ejercicio 6h #
#-----#
tabla = pd.read_excel("ADRs_historial.xlsx")
tabla.set_index("timestamp", inplace=True)
correlaciones = tabla.corr().round(2)
correlaciones.replace(1,0, inplace=True)
correlaciones.idxmax()
```

```
BBAR      GGAL
BMA       GGAL
CRESY     BMA
EDN       PAM
GGAL      BBAR
PAM       BBAR
TEO       BBAR
TGS       PAM
YPF       GGAL
dtype: object
```

Títulos de esta colección

A mayo de 2020, los tres primeros tomos están ya editados y en proceso de impresión, la idea es un nuevo lanzamiento por mes hasta completar toda la colección.

- t_[0] Primeros Scripts: Introducción a Python y entornos de desarrollo recomendados. Estructuras de datos y variables. Estructuras lógicas básicas, ciclos y estructuras lógicas más complejas, concatenadas y anidadas
- t_[1] DataFrames: Uso intensivo de la librería Pandas, estructuras de datos matriciales en formato de tablas, importación y exportación de datos con Excel y Python. Uso de funciones estadísticas con Pandas
- t_[2] Matplotlib: Gráficas en Python, gráficas de líneas, gráficos financieros de velas, gráficas de columnas, columnas apiladas, columnas en 3D, anillos, gráficos estadísticos, histogramas y diagramas de cajas, introducción a modelización de distribuciones con scipy
- t_[3] APIs de Market Data: Uso de funciones, JSON, requests http, APIs de datos de acciones, precios, fundamentales, indicadores, bonos, FX, opciones, futuros y datos económicos.
- t_[4] APIs de Conexión a Mercados: APIs para acceso al mercado en Argentina API de Rofex, API de Invertir Online. API de Alpaca para inversiones en USA. Paper trading, sandbox o cuentas simuladas. APIs de Exchange cryptos para mercados spot, futuros y opciones. HitBTC, FTX, Deribit, Binance etc.
- t_[5] SDKs y Librerías: SDKs de conexión a APIs financieras, librerías de dataFeed, de colocación de órdenes, de análisis de datos financieros y estadísticos (Análisis de tendencias, indicadores técnicos, análisis de portafolios, riesgo, etc)
- t_[6] BBDD: Estructuras de bases de datos relacionales, uso de MySQL, sentencias, armado de funciones para CRUDs. Mantenimiento de Bases de datos automatizadas
- t_[7] Backtestings: Ejecución offline y online de un backtesting. Guardado de datos en BBDD, consulta y actualización de backtestings. Métricas de un backtesting
- t_[8] Screenings: Screenings en tiempo real y programados. Sistemas de alertas de screenings. Ejemplos de screenings de ratios de AF y de señales de AT. Screenings de ratios de análisis Quant.
- t_[9] Portabilidad de estrategias: Estudio cruzado de estrategias backtesteadas en diferentes mercados, activos, grupos de activos, segmentación por afinidades, screenings y retesteos de portabilidad por diferentes agrupamientos
- t_[10] Bots de trading: Ejemplos de diferentes estructuras de bots de trading, bots de balanceo de carteras y seguimiento de índices. Bots especulativos de scalping y swing trading. Bots de arbitrajes y arbitrajes estadísticos. Bots de coberturas de posiciones abiertas especulativa (reemplazo de stopLoss por cobertura con opciones). Ingeniería básica y costos operativos y de setup de un bot.
- t_[11] Inteligencia artificial en la industria financiera. Aplicaciones a estrategias de inversión. Algoritmos de aprendizaje supervisado y no supervisado. Regresiones y clusterización. Correlación y predicción de valores y eventos
- t_[12] Montecarlo y métodos modernos para medición de riesgos, simulación de eventos y posibilidades. Modelización de frontera eficiente de riesgo/retorno con simulación. Uso de las simulaciones seteo de escenarios

- t(13) Opciones a fondo con python, screenings con miles de activos con todas sus cadenas de opciones, pricing de primas por simulación y contraste contra modelos como el de Black&Scholes, griegas y algos de IA en regresiones de volatilidad para estrategias delta-neutral, long y short vega y vega-neutral.
- t(14) Ejemplos de implementaciones de bots reales con sistemas de control mixtos, gatillo de compra y de cierre de posición por leads de distintos tipos. Manejo dinámico de volumen de posición, sistemas activos y pasivos, bots de hedging, reporting real time etc
- t(15) Papers Quant y sus implementaciones en python